RWTH AACHEN UNIVERSITY

# A Pattern-Based Software Framework for Computer Aided Jazz Improvisation

Diploma Thesis at the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Jonathan Klein

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Wolfgang Prinz

Registration date: Oct 26th, 2004
Submission date: May 24th, 2005

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, May 24th, 2005

# Abstract

Improvisation in music is a complex creative process and the most native way of musical performance, but became secondary in the Western world with the prevalence of sheet music. Despite the high degree of freedom that improvisation offers, there are still rules to be followed, especially when playing together with others. Jazz music provides a foundation of well-established patterns that allow musicians to improvise in a group. However, playing freely under those constraints demands a great deal of technical ability and experience from the performer, which usually takes years of practice to achieve. Improvisation is hard to learn and a challenge even to the serious jazz musician, who may know most of the patterns, but has not fully internalized all of them. In addition, thinking about rules generally inhibits the creative process.

With computers being able to represent and apply patterns to given inputs, it seems natural to have them assist performers by taking some of the thinking off their backs, so they can focus more on being creative. In order to do so, *coJIVE*, the system we propose, implements essential patterns of jazz. Like in a standard jazz combo setting, the user is presented a *lead sheet*, which indicates harmonic progressions as chord symbols. An automatic accompaniment emulates a bass player and drummer. The player's input, which is served via a MIDI keyboard, infrared batons, or other device, is interpreted and modified to conform to musical rules. The extent of the corrections can be adjusted to match the user's own level of expertise. *coJIVE* provides multi-user support by adapting popular schemes like "trading fours", where players take turns that last four bars each.

This thesis describes the back-end framework that models the musical knowledge of the coJIVE system. It should be read together with the companion thesis by Jan Buchholz [2005] that describes the front-end and user interface of coJIVE.

# Kurzdarstellung

Improvisation in der Musik ist ein komplexer kreativer Vorgang und die ursprünglichste musikalische Ausdrucksform, trat jedoch mit der Durchsetzung von notierter Musik in den Hintergrund. Trotz des hohen Grades an Freiheit, den die Improvisation bietet, sind Regeln zu befolgen, insbesondere im Zusammenspiel mit anderen Musikern. Jazz bietet Musikern ein Fundament breit anerkannter Spielregeln zur koordinierten Improvisation in der Gruppe. Unter diesen Beschränkungen trotzdem frei zu spielen, verlangt jedoch ein hohes Maß an technischer Fertigkeit und Erfahrung, die in der Regel jahrelanges Üben erfordert. Improvisation ist schwer zu erlernen und ist eine Herausforderung auch für ernsthafte Jazzmusiker, die womöglich die meisten der gängigen Muster kennen, sie jedoch vermutlich nicht vollständig internalisiert haben. Darüberhinaus bremst jedes Nachdenken über Regeln die Kreativität.

Da Rechner in der Lage sind, Muster darzustellen und auf Eingaben anzuwenden, erscheint es nur natürlich, diese Fähigkeit auszunutzen, um Menschen beim Spielen zu unterstützen. Ein Teil der notwendigen Denkarbeit kann dem Spieler abgenommen werden, um mehr Raum für Kreativität zu schaffen. Mit diesem Ziel setzt coJIVE charakteristische Muster aus dem Jazz um. Wie in einer normalen Bandsituation wird dem Benutzer ein Leadsheet präsentiert, das die Harmoniefolgen von Jazzstücken in Form von Akkordsymbolen darstellt. Eine automatische Begleitung liefert Bass und Schlagzeug. Die Eingaben des Spielers, die über ein MIDI-Keyboard, Infrarot-Schlegel oder andere Eingabegeräte geliefert werden, werden interpretiert und verändert, so dass sie musikalischen Regeln entsprechen. Das Ausmaß dieser Korrekturen ist so regelbar, dass es den Fähigkeiten des Spielers entspricht. coJIVE unterstützt mehrere Benutzer gleichzeitig, indem es sich an beliebte Spielvarianten wie z.B. das “Trading Fours” anlehnt, bei dem Spieler sich mit viertaktigen Soli abwechseln.

Diese Diplomarbeit beschreibt das Back-End-Framework, welches das musikalische Wissen des coJIVE-Systems modelliert. Sie sollte zusammen mit der Partnerarbeit von Jan Buchholz [2005] gelesen werden, die das Front-End und die Benutzerschnittstelle von coJIVE behandelt.

# Acknowledgments

I thank my family, especially my parents, Robert and Diethild Klein, for supporting me throughout my whole life. I am also very grateful for the emotional support from my friends Martin Roemer, Angela Braun, Frank Stiegler, and many others. Thanks go to Stefan Michalke and Sabine Kühlich for their musical coaching and encouragement.

Many thanks to my supervisor, Eric Lee, for his help and advice throughout the development of this work, and also for his after-work Tai-Chi lessons, and Prof. Dr. Jan Borchers for the thesis topic itself, the friendly atmosphere at the Media Computing Group, and the legendary P7 Movie Nights. Thanks to my project partner, Jan Buchholz, for our good cooperation. Thanks to Stefan Werner for keeping the machines at our workplace running, and to Nils Beck for spending his free time on the design of the cover page. Thanks to everybody at the Media Computing Group for being such helpful and enjoyable folk.

# Contents

# List of Figures

# List of Tables

# 1. Introduction

## 1.1 Motivation

In jazz music, unlimited creativity and freedom are crucial. The ability to improvise is the most essential skill needed to engage in this kind of music. The most prevalent form of jazz performance is the small group performance, where groups of usually three or more musicians improvise over a given tune. All the musicians need to agree upon is a roughly sketched form of the piece, which is specified in a more or less standardized way: the so-called *lead sheet*. It consists of a notation of the main melody and has symbols written above the staves to indicate harmonies.

A typical small group performance, for example at an open jazz session, is ideally a very structured process: As long as mainstream jazz is played (as opposed to Free Jazz, for example), Musicians agree upon a "standard", one of several hundred pieces that have made it into the common repertoire of the community. By convention, the basic structure of the performance is almost always the same: Someone plays the main theme while the others accompany, then everyone gets to play their solos. Optionally, the band can agree to "trade fours" after the solos, which means taking turns in four-bar solos, interleaved by unaccompanied four-bar drum solos. Solos are usually one or more complete song choruses in length. After all the solos and the optional trading of fours, the main theme is repeated to conclude the performance. Details such as the tempo, the rhythm, the key and the musician who plays the theme are usually agreed upon in advance, while other aspects of the performance such as the order of solos, the ending, trading fours etc. are often negotiated during the performance via eye contact, body language or shouting. Figure 1.1 shows the minimal trio setting with piano, drums and bass.

The establishment of rules in contemporary music proceeds similarly to the formation of rules in natural sciences: Observations are distilled into rules, which can describe, but never prescribe, what is possible. In fact, jazz has developed throughout the past 100 years by constant breaking of rules. Each decade yielded a new style of jazz that polarized the audience and music critics, because some musician or group had done something that was not supposed to be done according to established conventions.

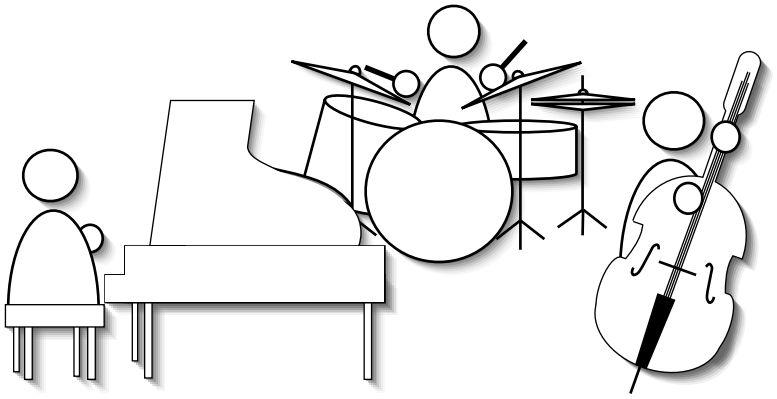


Figure 1.1: A Jazz trio: piano, drums and bass.

In order to engage in improvisation, musicians need a considerable amount of skills, which include good command of the instrument, theory knowledge, trained ears and rhythmic ability. Jazz musicians learn these things and try to apply them consciously when practicing at home. In an actual performance however, things happen way too fast to leave the players with enough brain capacity for extensive reasoning. They hear in their heads what they want to play and try to reify their ideas by reflexes. Only what has been practiced for a long time (or a long time ago) will safely make it into live performances.

Especially in the early stages of learning an instrument or learning jazz, it is hard to focus on creativity and communication, which is essential for good improvisation. This is where technology can support hobby musicians and non-musicians who are still in the early learning process: If the set of applicable rules is known, it can be used to guide humans' playing into the right direction. Where a player is overwhelmed by rapidly changing harmonies, leaving him no time to adapt and find the right notes to play, he or she can have wrong notes corrected by the computer. What is more, the collaborative aspect of jazz music, such as negotiation of solo rounds, can be supported by the machine through different feedback mechanisms. Correction can be performed in real-time as long as rhythm remains untouched: Unfortunately, time works only in one direction, so corrections could only be made towards the future, i.e., "early" inputs could have delayed results in the output. However, this behavior would probably break causality and interfere with the player's rhythmic feel.

## 1.2 The coJIVE Project

In order to provide the kind of support described above, a system called coJIVE (**Co**llaborative **J**azz **I**mprovisation **E**nvironment) was designed and implemented. It plays a bass and drum accompaniment and allows several players to improvise, providing a typical jazz session environment. Players can use a MIDI (Musical Instruments Digital Interface) keyboard or play two infrared batons like a xylophone. "Bad" notes are either corrected or not even offered, depending on the type of input device. Musical theory knowledge is used to automatically classify the fitness of notes.

## 1.3 Project Organization

The coJIVE project is divided into two parts, which are henceforth called "front-end" and "back-end". The front-end is the interactive side of coJIVE, which was developed by Jan Buchholz and is covered in [Buchholz, 2005]. The back-end, covered by this thesis, uses knowledge from music theory to classify notes in a way that allows the front-end to correct bad notes. The following subsections outline the scopes of the two modules, which were developed in a collaborative effort.

### 1.3.1 The Front-End

- Supporting the improvisation experience, e.g., by correcting "bad" notes
- Adjusting the degree of support based on the user's level of expertise
- Supporting different input devices such as MIDI keyboard and infrared batons
- Collaboration support for multiple players
- User-centered, iterative design process with user studies

### 1.3.2 The Back-End

- Designing and implementing data structures to handle musical knowledge
- Using theoretical knowledge to classify notes in harmonic context
- Devising a data interchange format for lead sheets

## 1.4 Structure of This Document

The following chapters of this thesis are organized as follows: Chapter 2 explains some basic musical terms and concepts needed to understand the analysis process described in this work. This chapter can be skipped by readers who are familiar with musical theory and chord progression analysis. Chapter 3 describes other research that has been conducted in related areas like automatic chord progression analysis and jazz improvisation systems. In chapter 4, the architecture and design of the back-end system is explained, as well as the pattern-based analysis concept implemented in this system. Chapter 5 presents some implementation details of critical system components. Chapter 6 discusses the results yielded by the analysis process. Conclusions and potential future improvements to the current system are addressed in chapter 7.

# 2. Musical Theory

This chapter provides some theoretical foundations of the work presented in this thesis. In the following sections, a short explanation of musical concepts including pitch classes, intervals, scales and chords will be given. Chord progressions are of special importance to this work as they provide the harmonic grid for every jazz performance.

## 2.1 Pitch Classes

Western tonal music basically knows twelve distinct pitch classes, i.e., an octave, which is the range from a frequency $f$ to $2f$, is divided into twelve steps called *semitones*, each of which is approximately[1] $\sqrt[12]{2}$ times the frequency of the preceding one. This is at least true for keyboard instruments, where pitches are discrete, as opposed to strings or voice, which have a continuous pitch range. Pitch classes are named using latin letters ranging from A to G, plus one or more optional *accidentals* ♯ ("sharp") or ♭ ("flat"), where a ♯ shifts the pitch up by one semitone, and a ♭ shifts down accordingly. More than two accidentals are rarely necessary, but generally possible. The seven notes without accidental are called *natural notes*. Successive natural notes are either one or two semitones apart. The natural note series C, D, E, F, G, A, B constitutes the C major scale. The term "pitch class" is also referred to as "note", although an actual note in a musical score encodes more information than just the pitch class.

### 2.1.1 Enharmonic Spelling of Notes

Each pitch class has multiple representatives. For example, a C♯ belongs to the same pitch class as a D♭ or B♯♯, since they all sound the same. The assignment of a name to a given pitch, termed "enharmonic spelling", depends on the situation in which it occurs. Theoreticians are often very strict about the correct spelling of a note, since it encodes not only the pitch, but also to some degree the function of a note in its harmonic context. On the other hand, score authors sometimes choose to spell notes contrary to theoretical demands. This can occur when the harmonic function of a

[1] For aesthetic and practical reasons, instrument tunings are not completely regular.

note is ambiguous (e.g., in modulations), when re-spelling improves readability, or when tonality is so unclear or irrelevant that spelling becomes completely arbitrary.

## 2.2 Intervals

Intervals measure the distance between two notes. Their denotation is not based on actual semitone distance: An interval name has two components, the more significant of which is the interval degree. The degree is based on the distance of the two natural versions of the notes in question. Degrees are counted one-based, i.e., the interval between a note and itself is called a "first". Since the degree alone would only allow distance measuring in certain subsets of notes, the interval needs to be further refined. This is accomplished by using one of the prefixes *perfect, major, minor, diminished* or *augmented* to state which alterations are performed on the "natural" intervals. The applicability of a prefix depends on the interval degree[2]. The terms *minor* and *major* apply to second, third, sixth and seventh intervals, while the first, fourth and fifth intervals are assumed to be *perfect* if not explicitly stated otherwise. When interval sizes go beyond the terms *minor*, *major* or *perfect*, they are called either *augmented* or *diminished*, depending on whether they are altered up or down. The naming conventions for an interval also apply to its transpositions by whole octaves, i.e., a ninth can have the same prefixes as a second, a tenth is treated like a third, etc.

### 2.2.1 Enharmonic Spelling of Intervals

Obviously, interval naming is also dependent on enharmonic spelling, since the name of the interval is derived from the distance of natural notes. For example, F♯ and G♭ are two different names for the same pitch class, and so the intervals from C to either of the two notes are named differently. The interval C → F♯ is called an *augmented fourth*, while C → G♭ is a *diminished fifth*. This distinction becomes particularly important when one tries to derive sets of playable notes (→ scales, section 2.3) from the presence or absence of altered notes in a harmonic situation.

### 2.2.2 Abbreviations

As interval names are rather lengthy, musicians write (and often also speak) them in shortened form, using arabic numbering for the interval degree and characters for abbreviation of alterations. Downwards alterations are abbreviated by a ♭ sign. Sometimes *minor* is distinguished from *diminished* by using 'min', 'm' or '-'. In this document, we will stay with the ♭ sign to prevent any confusion. A ♯ sign is used to denote any upwards alteration. Unless otherwise stated, *major* or *perfect* is implicitly assumed, with the exception of the seventh interval: The minor seventh, in the jazz and Blues context, occurs more frequently than its major counterpart, which is why a seventh interval without any prefix is often assumed to be minor. To eliminate this source of confusion, the prefix 'maj' should be used to clarify that the major seventh is meant, and a 'min' or '-' prefix or, as it will be done here, a ♭ sign should be employed to signify the minor seventh. See Table 2.1 for abbreviations of the most important intervals.

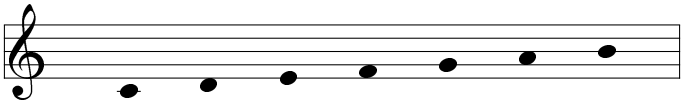
[2]This phenomenon can be attributed to the harmonic meanings that are commonly associated with different intervals.

| interval | abbrev. | semitones | interval | abbrev. | semitones |
|---|---|---|---|---|---|
| perfect first | 1 | 0 | perfect eighth | 8 | 12 |
| minor second | ♭2 | 1 | minor ninth | ♭9 | 13 |
| major second | 2 | 2 | major ninth | 9 | 14 |
| | | | augmented ninth | ♯9 | 15 |
| minor third | ♭3 | 3 | | | |
| major third | 3 | 4 | | | |
| perfect fourth | 4 | 5 | perfect eleventh | 11 | 17 |
| augmented fourth | ♯4 | 6 | augmented eleventh | ♯11 | 18 |
| diminished fifth | ♭5 | 6 | | | |
| perfect fifth | 5 | 7 | | | |
| augmented fifth | ♯5 | 8 | | | |
| minor sixth | ♭6 | 8 | minor thirteenth | ♭13 | 20 |
| major sixth | 6 | 9 | major thirteenth | 13 | 21 |
| minor seventh | ♭7 | 10 | | | |
| major seventh | maj7 | 11 | | | |

Table 2.1: Abbreviations of important intervals. Intervals that are exactly an octave apart are placed in the same row.

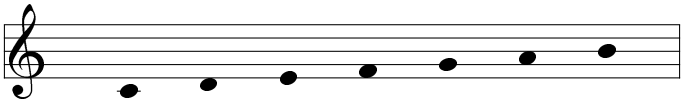
## 2.3 Scales

A scale is a set of notes, usually written in ascending or descending order. They provide closed systems from which melodies and harmonies are formed over large passages of musical pieces. Such a closed system is often referred to as *tonality* or a *key*. The best-known example of a scale is the C major scale, which is constituted by the white keys on the piano.



c d e f g a b

Since the character of a scale does not depend on absolute pitches, but rather on the relations between them, it can be transposed into any key by displacing all notes by the same interval. In fact, it makes more sense to view a scale as a set of intervals, either stepwise (the distance from one note to the next), or relative to a single point of reference, commonly called *root note*. The former perspective may be helpful when trying to play through a scale, but the latter one is more relevant for analysis tasks. Hence, scales are often specified as a series of interval abbreviations. In the case of C major, this reads simply:



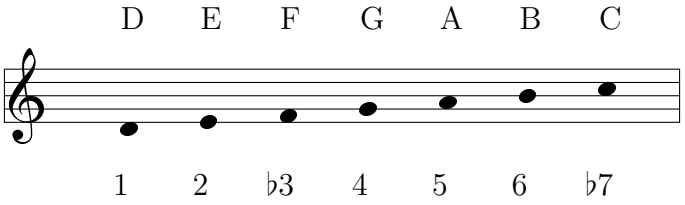
1 2 3 4 5 6 7

Although the interval-based view of scales should always be kept in mind, for illustrative purposes it is common to just give an example in some random key, mostly in C or whichever key requires the smallest number of accidentals.

There are hundreds of possible scales[3], but already a few are enough to generate reasonable improvisations. The most popular scales are called major , harmonic minor, melodic minor ascending[4], diminished and whole-tone scales. The notation of these scales in C is shown in Figure 2.1.

### 2.3.1 Modes

By enumerating the notes of the C major scale starting from a note other than C, let's say D, we obtain a different *mode* of the C major scale, which is called *D dorian* or *second mode of C major*. Analogously, the third mode of C major starts at E and is called *E phrygian*. Relative to this new root note, the intervallic representation changes:




The "regular" C major scale (the one that starts with C) is called *C ionian*. The major scale has seven modes altogether, where each has its unique character, but also shares properties with other modes. The most outstanding of these common properties is the quality of major or minor sound. This quality is solely determined by the size of the third interval (w.r.t. the root) in a mode. Modes 2, 3, 6 and 7 of the major scale have a minor third above their root note and have a "minor sound". Modes 1, 4 and 5 have a major third and therefore "sound major". In order to make the minor character of the song clear to the listener, the minor third should appear somewhere in the melody or harmony.

Other intervals represent other aspects of the sound of scale modes, but not quite as obvious. Because of the properties that different modes of different scales share, modes are often named after similar modes of the major scale. Table 2.2 lists the most important modes of different scales along with their common names.

### 2.3.2 Tonality and the Circle of Fifths

Tonality, or simply a *key* can be seen as a closed system of harmonies. Keys are usually either major or minor, and classically this refers to the major scale or the natural minor scale. Transposing the major scale into keys other than C requires the use of accidentals at certain points. Moving upwards or downwards in intervals of perfect fifths modifies only one accidental at a time: G major, which is a perfect fifth above C, requires one accidental for the note F♯. D major (a fifth above G) requires two sharps, one for F♯ and one for C♯. Likewise, F major, which is a fifth below C, has one flat sign at B♭. Since the natural minor scale is simply the sixth mode of the major scale, A minor has the same accidentals as C major, B minor corresponds to

[3]There are $\frac{}{127=792}$ possibilities to select seven notes out of an octave, and a scale is not even required to have exactly seven notes.

[4]In classical theory, the ascending melodic minor scale differs from the descending one. On descent, it is equal to the natural minor scale (aeolian or sixth mode of major). Thus, the term "ascending" is often added in the literature, but will be omitted in the rest of this work.

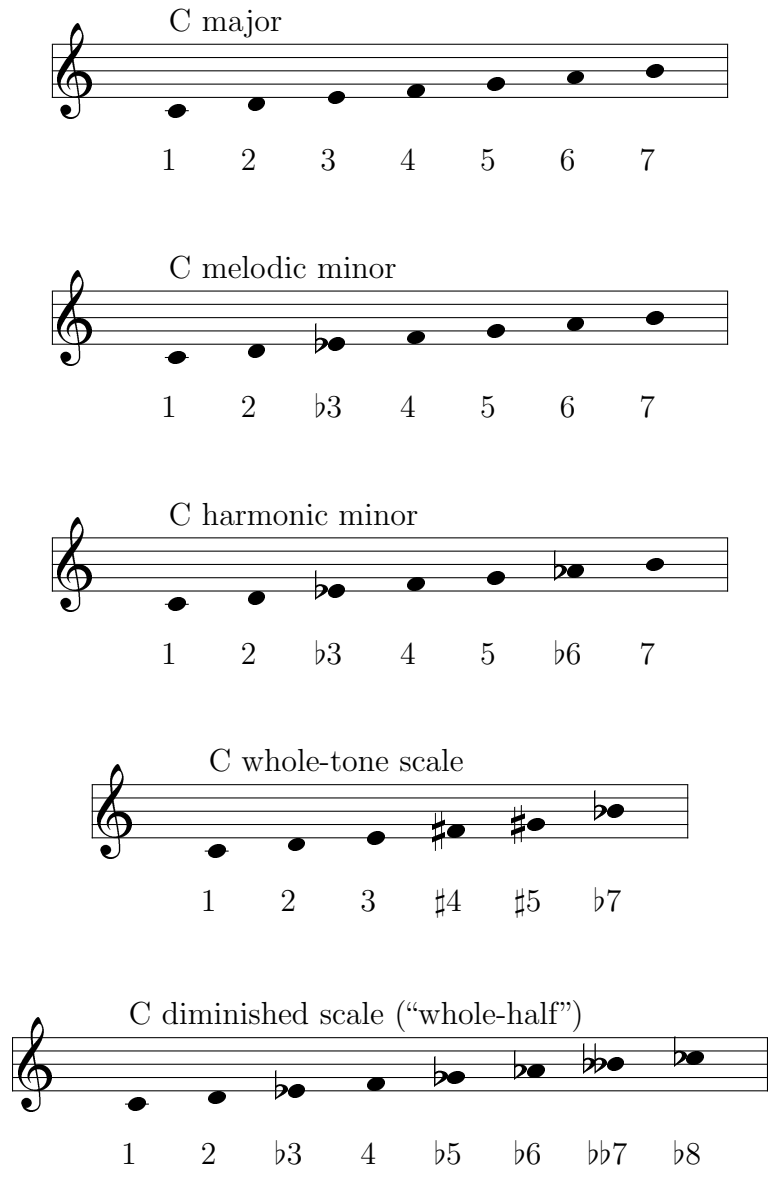
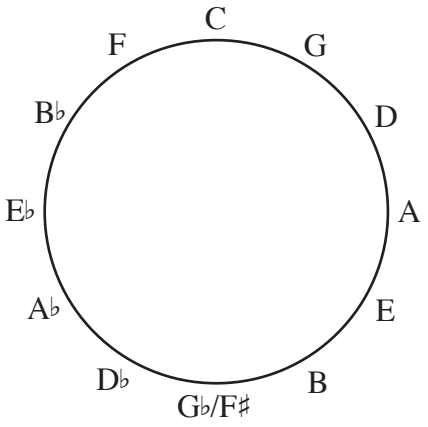



Figure 2.1: Some of the most important scales used by jazz musicians. The diminished scale is built of alternating whole-tone and half-tone steps. The whole-tone scale consists only of whole-tone steps. The ♭7 and the ♯6 fall on the same note, but its function is mostly that of a minor seventh. Musicians use the above scales in different modes (see subsection 2.3.1). Major and the melodic/harmonic minor scales have seven distinct modes each, while symmetric scales have less: the whole-half scale has only two unique modes, and the whole-tone scale has only one.

| Mode | Major | Melodic Minor | Harmonic Minor | Diminished |
|---|---|---|---|---|
| I | ionian | minor-major | harmonic minor | whole-half diminished |
| II | dorian | phrygian ♯6 | | half-whole diminished |
| III | phrygian | lydian ♯5 | | |
| IV | lydian | mixo ♯11 | | |
| V | mixolydian | mixo ♭13 | HM5 / mixo ♭9♭13 | |
| VI | aeolian | locrian 9 | | |
| VII | locrian | altered | | |

Table 2.2: Modes of the most important scales. The names of melodic minor modes are mostly derived from major modes. Less important modes have been omitted. The notion of modes is obsolete for the whole-tone scale, which is perfectly symmetric and has only one mode. This is why it is not listed in this table.

D major and so on. In order to visualize those relations between major and minor keys, musicians use an image called the *circle of fifths*. It is called a circle because the movement by fifths is actually a circular traversal of all 12 pitch classes (although an enharmonic re-spelling has to be made to close the circle). In clockwise direction the circle of fifths reads C, G, D, A, E, B, F♯, and in counter-clockwise direction it reads C, F, B♭, E♭, A♭, D♭, G♭. Respelling occurs at F♯ and G♭ in order to obtain a closed circle.




## 2.4 Chords

A set of simultaneously triggered notes is called a chord. Each combination of intervals in a chord creates different degrees of tension. Moreover, a chord implies one or more tonal centers, harmonic points of rest that are (often subconsciously) anticipated by the listener.

### 2.4.1 Relation between Chords and Scales

There are two perspectives on the relation between scales and chords. One way to look at it is to say that chords originate from scales. Starting with a particular mode of a given scale, a chord can be constructed by stacking every other scale note on top of each other. Many music styles, like classical or pop, often use chords built

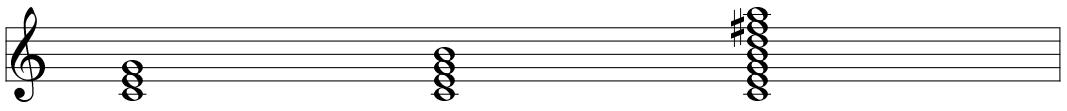



Figure 2.2: Some notated examples for chords. The chord on the right consists of all seven notes of the C lydian scale.

from only three notes ("triads"), while jazz chords have usually at least four notes ("seventh chords"). Since multiple scales yield the same chord, the mapping between scales and chords is generally ambiguous. However, the more notes a chord uses, the more the set of potential scales of origin is narrowed down. Some chords even use up a whole seven-note scale and thereby fix a scale completely. Some examples are given in Figure 2.2.

From the opposite perspective, a scale or mode can be said to originate from a chord. The most important notes in the chord, namely the ones that form first, third, fifth and seventh intervals to the root, are called "chord notes", while the ninth, eleventh and thirteenth are termed *options*. Chord notes determine very fundamental properties of the chord which cannot trivially be changed. Options can produce additional tension but are more situation-dependent. By selecting options, one basically fills the large gaps between chord notes.

### 2.4.2 Lead Sheets

Jazz musicians rarely use explicitly notated material. Instead, they mostly rely on a score that contains only one notated staff that describes the main melody (the theme) of the tune, decorated by combinations of letters, accidentals and numbers that give hints on which harmonies to put underneath the melody. This task of harmonization is one of the liberties of a jazz musician, who is usually able to read and analyze these seemingly cryptic superscripts called *chord changes*. Such scores are called "lead sheets" and can be found in collections of jazz standards called "Fake Books" or "Real Books". One of the most widespread books in the jazz community is [Sher, 1988]. An example of a lead sheet is shown in Figure 2.3.

### 2.4.3 Chord Symbols ("Changes")

Jazz and pop musicians use chord symbols to simplify the written representation of chords. These chord symbols, commonly referred to as "changes", are less detailed than standard notation, which makes them easy to read, naturally supporting improvisation.

There are several conventions for chord spelling, but none of them is completely intuitive or straightforward. Chord symbols can be arbitrarily general or specific. In general, a chord symbol consists of the name of a root note, followed by quality information (whether the chord is minor, diminished, augmented...), followed by options and/or alteration information. An example for a rather complex chord symbol that specifies options would be:

$$G^{7/9/13}$$

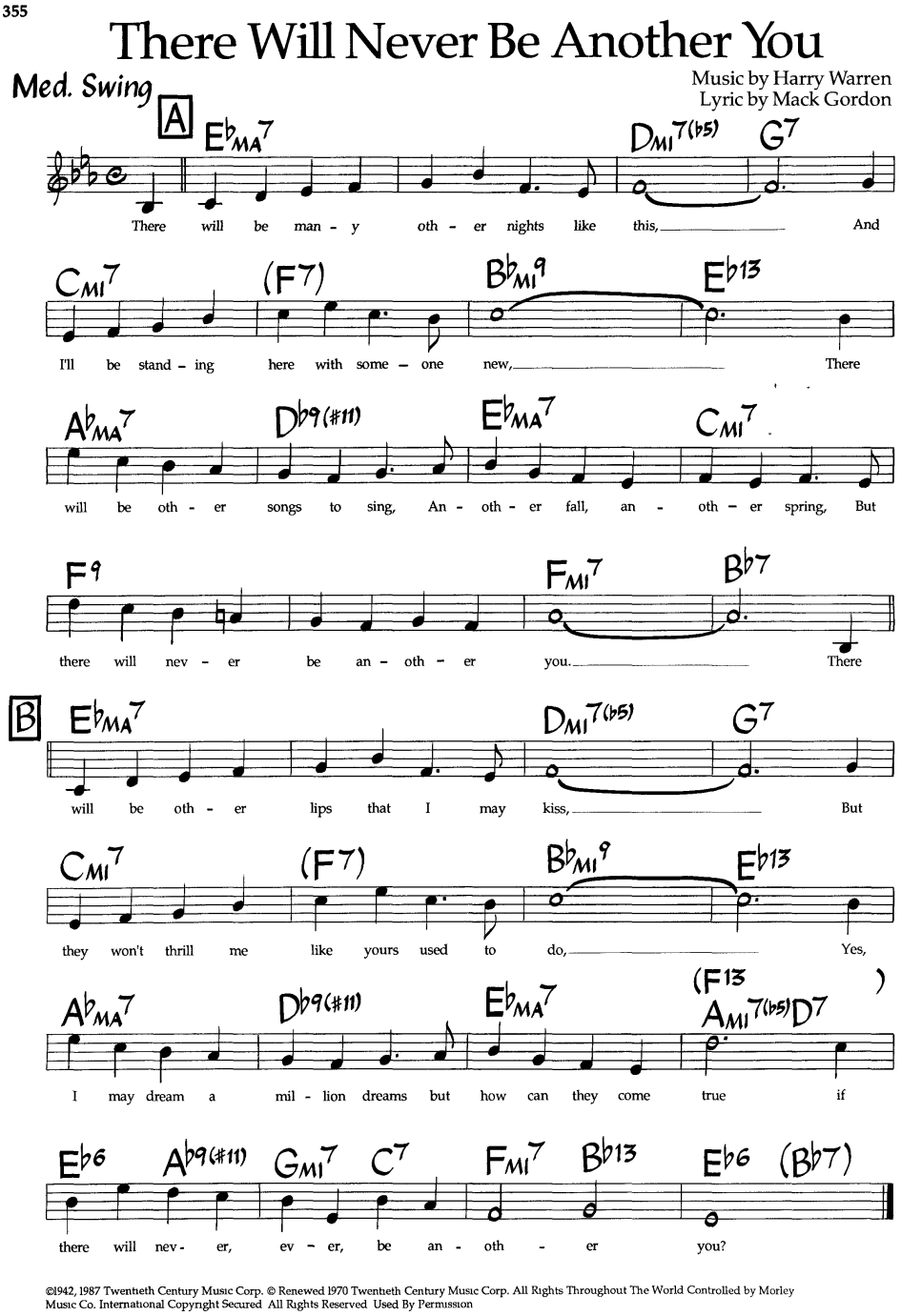



Figure 2.3: Lead sheet of the song "There Will Never Be Another You", taken from The New Real Book [Sher, 1988]. The notation gives the main melody, and the symbols above the staves indicate harmony. The tempo mark on the upper left indicates an approximate speed recommendation ("Medium") and the type of rhythm to be used ("Swing"). The boxed letters A and B are rehearsal marks that outline the song structure. The key signature in the first row (three ♭s) indicates the key of the tune (E♭ major), and the time signature following it states "c" for *common time*, which is synonymous with 4/4.

which specifies a G major chord (G, B, D) with a minor seventh (F), also called *G dominant seventh chord*. Additionally, it has a major ninth (A) and a major thirteenth (E) as options. Options can be altered up and/or down (the possibilities vary for each option). Here is an example for a chord symbol with altered options:

$$G^{\mathbf{7\flat 9\flat 13}}$$

This describes a G dominant seventh chord (G, B, D, F), a minor ninth (A♭) and a minor thirteenth (E♭).

| Symbol | Intervals | Name/Description |
|---|---|---|
| $X^{maj7}$ | 3, 5, 7 | major seventh chord |
| $X^{7}$, $X^{9}$, $X^{13}$ | 3, ♭7 [, 9[, 13]] | dominant seventh chord |
| $X^{7+}$, $X^{7\sharp 5}$ | 3, ♯5, ♭7 | augmented (dominant) seventh chord |
| $X^{-7}$, $X^{-9}$, $X^{-11}$ | ♭3, 5, ♭7 | minor seventh chord |
| $X^{-7\flat 5}$, $X^{\varnothing 7}$ | ♭3, ♭5, ♭7 | half diminished seventh chord |
| $X^{\circ 7}$ | ♭3, ♭5, ♭♭7 | (fully) diminished seventh chord |

Table 2.3: Table of common chord symbols for different general qualities. The middle column lists the intervals that determine the chord's quality. Note that the perfect fifth is not listed in the dominant chord, as it is not needed for this chord to function. Altered options (♭9, ♯9, ♯11, ♭13, ...) can be appended to the chord symbols to indicate a deviation from the standard.

#### 2.4.3.1 Abbreviating Abbreviations

Chord symbols can get even more concise. Very common changes have special names which make them even easier to read, at least by trained people. Here are some examples (replace the X by any note name):

**$X^{alt.}$:** short for $X^{7\flat 9\sharp 9\sharp 11\flat 13}$ and called "X altered", because the chord implements all possible alterations of options.

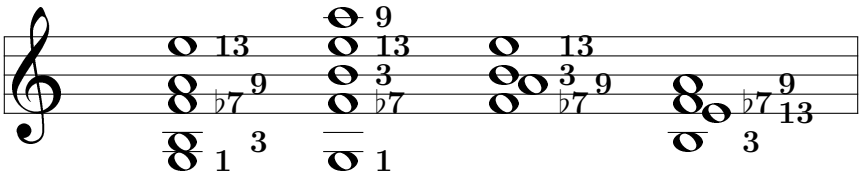
**$X^{7+}$:** a dominant seventh with an augmented fifth[5], whose regular spelling is $X^{\sharp 5}$.

**$X^{9}$, $X^{13}$** When only one option, e.g., 13, is specified, this actually means "extend the dominant seventh chord using all possible (unaltered) options up to the 13th". It should be mentioned that the perfect 11th, forming a minor ninth with the third, is very dissonant on dominant seventh chords and is implicitly excluded from the set of possible options. Thus, $G^{13}$ has actually the same meaning as $G^{7/9/13}$.

**$X^{\varnothing 7}$** This is short for a half-diminished seventh chord, usually spelled $X^{-7\flat 5}$.

All options of higher degree than the highest one in the symbol can potentially be added by the musician. If only, let's say, a $G^7$ is specified, the musician can add the appropriate options, the selection of which he or she perform (consciously or subconsciously) using some sort of context analysis (and also personal preference). Table 2.3 shows a list of common chord symbols.

[5] [Pachet, 1993] describes the '+' as an invitation to add whichever crazy alteration comes to mind, but in fact the '+' has the very defined meaning of an augmented fifth.




a) b) c) d)

Figure 2.4: Voicing examples for a $G^{13}$ chord. The numbers next to the notes denote the intervals to the root G. Examples a) and b) are root voicings , c) and d) are rootless.

### 2.4.4 Voicings

While a chord symbol just tells which notes can be used, it does not give any clues about their arrangement. A particular arrangement of notes is called a *voicing*. There are many possibilities to actually implement a chord symbol on a polyphonic instrument such as piano or guitar, and to find and practice those that sound good is a life-long pursuit of many jazz musicians. In standard combo settings, it is common that the piano plays voicings that do not contain the root note of the chord, since the bass will play the root anyway. That way the pianist has one more free finger for adding an option. Figure 2.4 shows a few voicings for $G^{13}$.

## 2.5 Harmony Analysis

When listening to a tune with its underlying chord progression, our ears tend to register a tonal center (or tonality), a point of rest where we want the harmony to resolve to, at most positions within the tune. For each chord change, there is a number of typical, expected continuations, and rather often this expectation is met. Places where this is not the case are often modulations—places in which the tonal center shifts.

The task of harmony analysis consists of finding the tonal centers and determining the functions of chords within their respective tonality. The jazz musician's tool of choice for that purpose is called *Roman Numeral Analysis*, which is described below.

### 2.5.1 Harmonic Functions

Tonality itself is perceived in terms of scales. In Western music, the most relevant scales are the major scale, the melodic and harmonic minor scales. Especially in jazz, also the diminished scale plays an important role, since it is often used as an interesting substitute for other, more conventional scales.

Chords are derived from these scales. Building seventh chords, the most basic chords in jazz, from the notes of a major scale by stacking thirds on top of each other generates seven chords, with differently sized third, fifth and seventh intervals. However, some of these chords are of the same type, as can be seen in Figure 2.5. Stacking thirds on the root note results in a chord with a major third, a perfect fifth and a major seventh. The same type of chord results when stacking thirds on the fourth note of the scale. In fact, the chords become unique once also option notes of the chords are filled in, i.e., when two more thirds are stacked on top of the original

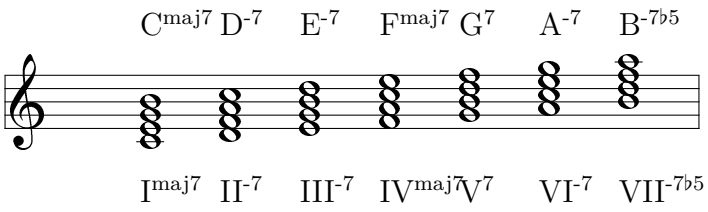



Figure 2.5: Seventh chords derived from the C major scale. Within the tonality of C major, these chords are assigned Roman Numerals.

chords. In theory, each of these chords has a certain harmonic function, i.e., a set of probable successor chords, potential substitutions etc. These functions are denoted by Roman Numerals, specifying the degree of the scale that the chord stems from.

## 2.5.2 Roman Numeral Analysis

In Roman Numeral Analysis, one tries to derive each chord's Roman Numeral by looking at the context in which it occurs.

### 2.5.2.1 Patterns

Some sequences of chords are widely used and well-studied, so if the neighborhood of one chord change is identified as one of these common patterns, theory can suggest certain scales. Probably the best known and most frequently used pattern is the *II-V-I progression*. One example for a II-V-I is $D^{-7}$ $G^{7}$ $C^{maj7}$. The root of each of these chords drops down to its successor's by a perfect fifth, leading to a II-V-I relation of the roots. The notes implied by the chord symbols (D,F,A,C)-(G,B,D,F)-(C,E,G,B) constitute the complete C major scale, which suggests that the sequence is to be interpreted in the context of that scale. Even the II-V part is enough to suggest the tonality of the I to the listener – the actual resolution is not necessary. There is also a version of II-V-I in minor tonality, and it can even be used to move over from major to minor or vice versa. In fact, the II-V-I is only the end-piece of a larger cadenza that reads I-IV-VII-III-VI-II-V-I-.... This cadenza consists only of dropping fifths and sounds very logical and predictable.

Because jazz music is often so complex, beginners are quickly overwhelmed with the amount of material they are confronted with. Although serious musicians should familiarize themselves with each tune they play, maybe even perform a complete formal analysis, it is sufficient in most cases to master a few common chord patterns in order to deliver an acceptable improvisation. There are typical scale choices for these established patterns which can be memorized. Some of the most important patterns, taken from [Jungbluth, 1981], can be found in Table 2.4.

An entertaining example for improvisations over unknown tunes can be found in the legendary recording of "Giant Steps" ([Coltrane, 1960]): The piano solo is very short (in comparison to John Coltrane's 15 choruses of solo), and at least to the more experienced listener, it sounds like patchwork. There are rumors saying that pianist Tommy Flanagan received the lead sheet via mail, and it had no metronome indications on it. He had expected the tune to be in ballad (slow) tempo. In fact, "Giant Steps" is an up-tempo tune played at about 290 beats per minute. It consists only of V-I and II-V-I patterns, modulating through three different keys. As can be heard on the recording, Flanagan seems to have had a really hard time improvising

something meaningful over the fast changes, but was skilled enough to come up with some fill material that did at least not sound wrong. His knowledge of II-V-I patterns was probably very helpful there.

#### 2.5.2.2 Tritone Substitution

Dominant chords (most frequently occurring on degree V) can often be substituted by a dominant chord on the note which is a tritone (diminished fifth or augmented fourth) away from the original root. For example, a $C^7$ could be replaced by a $G\flat^7$. The reason for that lies in the third and seventh interval, the size of which determines the chord's dominant property. In a dominant chord, the third and seventh form an interval of a tritone. Hence, shifting the third and seventh by one tritone only switches the order of the two notes. The seventh and third of a dominant chord are the third and seventh of its tritone substitute. Tritone substitution leads to patterns like $II^{-7}$-$\flat II$-$I^{maj7}$, where the V is replaced by the $\flat II$.

#### 2.5.2.3 Reharmonization

Tritone substitution is just one of many methods to replace chords by other chords. In the Be-Bop era (mainly the 1940s), musicians were tired of the old Swing tunes of the past decades. They reworked them harmonically to make them more interesting. A pioneer in that matter was saxophone player Charlie Parker, who often heavily reharmonized the standard 12-bar Blues form, for example in his "Blues for Alice". Charlie Parker's piece "Donna Lee" is a reharmonization of the old Swing tune "Indiana", where the theme was replaced and the changes modified, but the original piece still remains perceivable. John Coltrane reharmonized Miles Davis's "Tune Up" by replacing all the II-V-I progressions by a longer progression modulating through three keys, the result being the piece "Countdown". The II-V-I substitute is known as *Coltrane Changes* since. Such reharmonization adds harmonic diversity to the tunes, but also increases complexity to its analysis, because harmonic functions become more subtle and less predictable.

| | Pattern | Scale(s) | Description |
|---|---|---|---|
| 1) | $II^{-7}$<br>$V^{7}$ | dorian<br>mixolydian | Most common chord progression, normally resolved into $I^{maj}$. Resolution is often omitted. |
| 2) | $II^{-7}$<br>$V^{7alt.}$ | dorian<br>altered | Variant of 1), with possible resolution to I minor. |
| 3) | $II^{-7}$<br>$V^{7\flat9}$ | dorian<br>mixo♯11♭9 | Another variant of 1). Notes in the melody determine which one of the three is applicable. |
| 4) | $II^{-7\flat5}$<br>$V^{7\flat9}$ | locrian/locrian 9<br>HM5 | The regular II-V cadenza in minor.<br>Locrian 9 is used in exceptional cases. |
| 5) | $II^{-7\flat5}$<br>$V^{7alt.}$ | locrian/locrian 9<br>altered | Variant of 4), sometimes the $V^{7alt.}$ is used to move from a minor to a major key. |
| 6) | $II^{-7}$<br>$\flat II^{7\sharp11}$ | dorian<br>mixo♯11 | Major II-V pattern with V substituted by ♭II.<br>Advantage: Bass line descends in semitones. |
| 7) | $II^{-7\flat5}$<br>$\flat II^{7\sharp11}$ | locrian<br>mixo♯11 | The same as 6), but in minor tonality. |
| 8) | $VI^{-7}$<br>$II^{-7}$<br>$V^{7}$<br>$I^{maj7}$ | aeolian<br>dorian<br>mixolydian<br>ionian | "1625" cadenza that became famous through Gershwin's "I Got Rhythm". |
| 9) | VI-II-V-I | | Rotated version of 8), where the $VI^{-7}$ is often substituted by one of the following alternatives:<br><br>$\sharp I^{\circ7}$ whole-half<br>$\sharp I^{-7}$ dorian<br>$\flat III^{\circ7}$ whole-half<br>$\flat III^{-7}$ dorian<br>$\flat III^{7}$ mixo♯11<br>$III^{-7}$ dorian<br>$II^{7}$ mixo♯11<br>$VI^{7}$ HM5/altered |
| 10) | $V^{7}$<br>$\curvearrowright V^{7}$<br>$\curvearrowright \ldots$<br>$\curvearrowright V^{7}$<br>$\curvearrowright I$ | mixo♯11<br>altered<br>...<br>altered<br>... | Dominant chain following the circle of fifths (descending). |

Table 2.4: The most commonly occurring chord patterns along with their most suitable scale assignments. This list was adapted from [Jungbluth, 1981].

# 3. Related Work

This chapter presents some of the research that has been conducted in the area of jazz improvisation with computers. The first section deals with representation of musical data. Several approaches to automatic analysis of chord progressions are discussed in the second section. The last section presents some interesting solo generation systems, the basic ideas of which could be adapted to support, rather than simulate, human creativity. For the work at hand, mainly the algorithmic realization of musical theory is relevant. For the interactivity perspective on existing systems for computer aided and computer generated improvisation, see [Buchholz, 2005].

## 3.1 Musical Data Structures

### 3.1.1 Object-Oriented Representation of Musical Data in MusES

Pachet [1993] proposes a set of classes that represent the basic entities needed to solve musical problems, such as harmony analysis. These classes include data structures for note names and intervals, scales and chords, all of which preserve and correctly handle enharmonic spelling information. The classes are implemented in Smalltalk.

The system relies on the assumption that the maximum number of flat or sharp signs of a note is 2, which yields five possible types of notes: natural, sharp, flat, double sharp and double flat. There are seven natural notes which a note can be based on, so the total number of possible notes is 35. The initial assumption implies that major or minor keys outside the circle of fifths, such as G♯ major, do not exist. In fact, this is not exactly true: Pieces written in G♯ major exist, and due to local key changes within musical pieces such keys can be reached.

#### 3.1.1.1 Representation of Notes

Based on the above assumption, MusES derives five subclasses from the superclass `Note`, namely `NaturalNote`, `FlatNote`, `SharpNote`, `DoubleFlatNote` and `DoubleSharpNote`. These subclasses are instantiated at initialization time to yield the 35 valid note objects. Basic note relations are set in the form of pointers: Each

natural note is linked to its successor and predecessor, and each note is linked to the objects representing its sharpened and flattened versions. The five subclasses slightly differ in the services they provide. For example, a `DoubleFlat` object does not provide a method to retrieve its flattened version, since it does not exist. The advantage of this structure is that no note needs to be instantiated after the initialization step, since everything is handled via references. The disadvantage is that it is obviously not sufficient in order to represent all possible notes, as notes with three or more sharps or flats are theoretically possible.

#### 3.1.1.2 Representation of Intervals

The interval representation, which was also used for the coJIVE back-end, is quite straightforward: an integer denotes the interval degree (second, fifth, seventh, ...), and another integer denotes the actual size of the interval in semitones. This information is sufficient to compute the exact full name of an interval (e.g., "augmented fourth"). The `Interval` class provides services like adding or subtracting interval instances and notes, or finding the reverse interval.

#### 3.1.1.3 Representation of Scales and Chords

Scales in MusES consist of a root note and a set of intervals. The actual notes in the scale can be computed on the fly, which is provided as a method in the `Scale` class. Chords are stored as a root note and a list of string symbols representing the chord's structure. The `Chord` class, like `Scale`, provides the computation of the actual notes as a method. It also supports the inverse operation, i.e., calculating the structure string from a given set of notes. Some other interesting services are provided, such as calculating all possible chords from a list of notes, either assuming that the list contains the root note, or without that assumption (which results in a larger set of candidate chords). `Chord` can also return a set of plausible root notes or generate chords from a `Scale` object, by stacking thirds.

#### 3.1.1.4 Basic Harmony Analysis

[Pachet, 1993] also contains an example on how to use MusES for harmony analysis. A "harmony analysis" is defined as a scale and a degree (a Roman numeral). A `HarmonicAnalysis` class is defined, aggregating these two pieces of information as instance variables. The methods provided by this class can find a set of possible scale/degree pairs for a given chord. Methods of that kind were also implemented in the coJIVE back-end during the evaluation of the optimization approach in [Choi, 2004] (see also: 3.2.4, 6.3).

## 3.2 Chord Sequence Analysis and Scale Suggestion

Several methods have been developed to perform analysis on chord progressions. While most of these methods did not have the intention of selecting scales or notes for improvisation, they delivered interesting ideas that helped developing a solution. Some of these methods are described next.

### 3.2.1 Modeling Expectation and Surprise in Chord Sequences

[Pachet, 1999] describes an effort to determine the degree of surprise in chord sequences using data compression methods. The symbols that are compressed are not isolated chord symbols, but rather changes between subsequent chords, which is a transposition-invariant representation. The changes are displayed as tuples of adjacent chords, transposed so that the first symbol starts with a C. For example, the sequences $D^7$–$G^{maj7}$ and $E^7$–$A^{maj7}$ both become $C^7$–$F^{maj7}$ in that representation.

From a corpus of jazz tunes, which are transformed in the way described above, a Lempel-Ziv (LZ) compression tree is built. Such a tree records new patterns as they occur and is mostly used for sequence prediction and compression. Since such a tree basically represents a dictionary of expected patterns, Pachet defines a likelihood measure used to classify chord changes in sequences by their degree of expectation or surprise. Furthermore, he extracts a grammar of chord production rules from the LZ tree. The algorithm, working only on the symbol level and possessing no explicit musical knowledge, discovered well-known rules on its own, e.g., the preparation of a dominant by a minor chord rooted a perfect fourth below (as in a $II^{-7}$–$V^7$ pattern).

### 3.2.2 Analysis of Chord Progressions: "Is Solar a Blues?"

In [Pachet, 2000], a system is described that performs a hierarchical harmony analysis on chord progressions. It detects so-called *shapes*, such as II–V progressions or turnarounds, in a hierarchical way, i.e., larger shapes depend on the detection of smaller shapes. In order to speed up the analysis and make it unambiguous, there are also rules to "forget" shapes, for example when one shape subsumes the other. Larger matches are preferred over smaller ones.

Each shape is assigned a scale and a degree during the analysis. Theoretically, this information could be used for scale selection. However, the tonality of a large detected structure does not always suggest a good or interesting scale. Pachet himself states that the system is not meant to select scales for improvisation.

The system can successfully identify Charlie Parker's "Blues for Alice", a particularly complex augmentation of the original blues form, as a blues and correctly determines its key. Pachet also demonstrates that his system fails to identify the tune "Solar" by Miles Davis as a blues, due to some very obscure chord choices in the final turnaround, breaking the blues structure. Musicians can hear past this "flaw" and would still say that "Solar" is a blues.

Unfortunately, little information is given about the order in which the set of rules is applied to the input chord sequences. During the development of the coJIVE backend, Pachet's ideas were a used as a starting point. However, since the goal of coJIVE was not the identification of global structure, but merely making a local selection of suitable notes, the complete hierarchical analysis was not re-implemented.

### 3.2.3 Scale Determination with RhyMe

RhyMe [Nishimoto et al., 1998] performs musical analysis according to a knowledge base that contains a particular jazz theory. Using this knowledge base, from each chord change a single scale is derived, which is then mapped to a seven-key input device. The notes are not mapped onto the keys according to their pitch, but by

harmonic function. In other words, there is one key for each degree of a diatonic scale. Unfortunately, there is no detailed explanation available on how the actual chord analysis proceeds.

### 3.2.4 Chord Analysis as Optimization

Andrew Choi maintains a Mac OS X programming weblog, where he made several entries describing his development of a tool for automatic bass line generation. He also refers to MusES [Pachet, 1993] and re-implemented large portions of it in C++. Finally, he refrained from the analytical approach and devised a simple minimization scheme [Choi, 2004] that chooses scales based on minimal change of pitches. A dynamic programming approach is used to efficiently minimize a global scale distance measure. Furthermore, the system incorporates a single rule that accounts for II–V progressions, which are among the most common progressions in jazz music.

## 3.3 Solo Generation

Although the automatic generation of solos is not the goal of coJIVE, the projects going into that direction are relevant to the design of coJIVE. Statistical and AI methods used for the generation of automatic sequences could be used to improve the quality of the music made with coJIVE in the future, and the back-end was designed with the integration of such methods in mind. The following section describes some of these solo generation approaches.

### 3.3.1 Solo Generation using Neural Nets

Judy Franklin [2001] has developed a system called CHIME, the core component of which is a recurrent multi-layer neural network. The net consists of one input layer, one hidden layer and one output layer. The outputs are fed back into the inputs to provide some concept of temporal sequence. The network is trained in two phases, namely an offline phase using static training data and an online phase in which a human player trades solos with the net.

#### 3.3.1.1 Phase 1: Supervised Learning

The system is trained to replay 3 Sonny Rollins tunes. To accomplish that, a supervised learning method, namely a recurrent back-propagation algorithm, is used. The net possesses about 50 hidden units, 26 outputs and 29 inputs in this phase. 24 inputs and outputs are for the chromatic notes of two octaves. Two more inputs/outputs are needed for rests and “new note” triggering. The remaining three inputs are context inputs that tell the system which song it is currently learning. After training phase 1, the system can already generate novel note sequences when given different context values than the ones used during the training.

#### 3.3.1.2 Phase 2: Reinforcement learning

After phase 1, the system can already trade fours with human players. Additionally, it is fed the player’s solo, in order to be able to relate to his particular style. For reinforcement learning, the system is extended by another 26 note/rest/note-on inputs (for the player’s input) and 50 more hidden units. The critic that provides the

reinforcement value (reward or punishment) consists of an evaluation function, rating the system's performance using an algorithm called SSR and some rather simple musical rules. The rule set which was initially used looks as follows [Franklin, 2001]:

“

1. *Given a chord, playing notes in the corresponding scale is good (except as noted in rule 3).*
2. *Given a dominant seventh chord (typical for jazz) adding the 9, flat 9, sharp 9, #11, 13 or flat 13 is very good, unless one has already been played within the last 2 beats.*
3. *Given a dominant seventh chord, playing the natural 4 is bad.*
4. *Given a dominant seventh chord, playing the natural 7 is very bad.*
5. *Resting is bad if the total amount of resting time so far is over half of the playing time.*

”

Using the above rules, the net turned out to grow huge weights or just settle on the same note. Consequently, among other improvements to the algorithm, the rules were revised, which resulted in the following set of rules:

“

1. *Any note in the scale associated with the underlying chord is ok (except as noted in rule 3).*
2. *Given a dominant seventh chord, adding the 9, flat 9, sharp 9, #11, 13 or flat 13 is very good. But, if the same hip note is played 2 time increments in a row, it is a little bad. If any 2 of these hip notes is played more than 2 times in a row, it is a little bad.*
3. *If the chord is a dominant seventh chord (typical for jazz), a natural 4 th note is bad.*
4. *If the chord is a dominant seventh chord, a natural 7 th is very bad.*
5. *A rest is good unless it is held for more than 2 16th notes and then it is very bad.*
6. *Any note played longer than 1 beat (4 16th notes) is very bad.*

” [sic]

### 3.3.2 Solo Generation using Statistical Models

In order to generate solos, statistical models of a human player's style can be built. Likelihood computations on the model and possibly some seed input can then be used to generate new note sequences in the style that the model was trained. Two interesting systems implementing such an approach are introduced next.

#### 3.3.2.1 BoB

The project described in [Thom, 2000] aims at developing an interactive music companion called "BoB" ("Band Out of the Box") that is able to trade fours with a player, not just by applying some fixed set of musical rules, but by building a probabilistic model of the player's style. Such a model is built from "offline learned knowledge" (OLK), which is gathered in an offline warm-up session in which the player improvises over some harmony. In the online phase, BoB processes the four bars of the player's solo part to generate its own musical answer, using the model to calculate the likelihoods of different generated pitch sequences. Out of these sequences, BoB chooses the one whose likelihood matches the likelihood of the human player's sequence most closely.

**Data representation**

Pitch and rhythmical data is stored in VLTs (variable length trees). The structure of the tree reflects the rhythmic structure of the sequences. All internal nodes have two or three children, and the leaves contain pitch information. Leaf ordering determines the temporal order of notes.

**Solo generation**

During the online phase, the following steps are executed:

1. Create a mutated copy of the player's VLTs (one for each bar).
2. Generate new pitch sequences by random walks from pitch to pitch, with transition probabilities drawn from the OLK model. Also compute the overall likelihood of the generated sequences during the walk.
3. Compare the likelihoods to the likelihood of the player's VLT and choose the closest match.

#### 3.3.2.2 The Continuator

In [Pachet, 2002], a system is described that can continue solos based on a previously trained model. A human player plays an input sequence, and as soon as he stops playing, the system continues in the style that it was trained for. The Continuator uses Markov Chains, a statistical method for sequence analysis, to model the styles of its teachers.

**Variable-Order Markov Chains**

Markov Chains can be viewed as probabilistic state machines in which states and transitions have probabilities assigned. Transition probabilities can be conditional: They can depend on a path of zero or more previous states. The length of that path is called the *order* of the Markov Chain. The probabilities in a Markov Chain are usually acquired by counting state transitions in training sequences from real-world data. Markov Chains can be used to calculate the probability of given sequences, or to predict continuations of sequences. While models with $m$ states and a low, fixed order $n$ can be stored in a matrix with $m^n \cdot (m+1)$ probability entries, this is not feasible

for higher orders. Furthermore, in the particular application of modeling melodies, not all sequences have the same length. This is why variable-order Markov chains are employed, which can be represented by a data structure called a *prefix tree*. The nodes of prefix trees reference states, while no two siblings reference the same state (i.e., the branching factor is bounded by the number of possible states). Each path through the tree starting at the root reflects a sequence with nonzero probability. Each node carries a counter which is incremented every time a traversal crosses the node during training. Given a previously trained prefix tree, sequences (prefixes) can then be traced from the root by following the nodes referencing the symbols in the sequence, and the counters can be used to calculate the sequence probability or find possible continuations. In order to be able to evaluate subsequences of the strings in the training data, each suffix of each training sequence must be added to the tree, e.g., if the sequence AABA is in the training set, then AABA, ABA, BA and A must be recorded. Figure 3.1 shows an example of a prefix tree along with sample calculations.

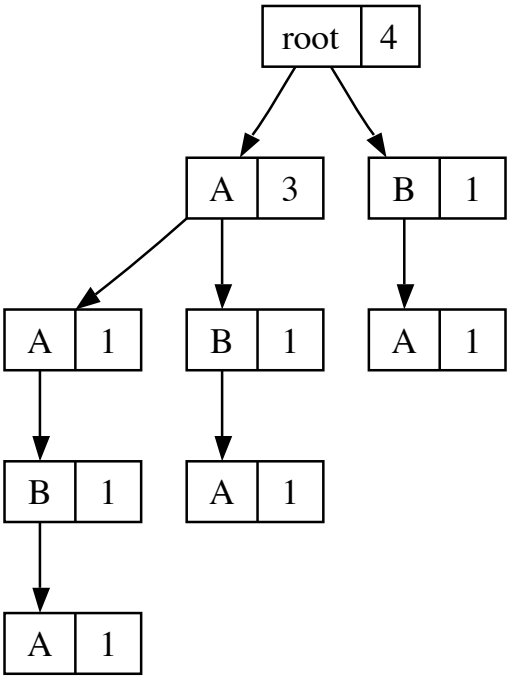



Figure 3.1: A prefix tree built from the sequence AABA and all of its suffixes. According to this model, the probability of an A appearing in an input stream is $P(A) = 3/4$. The probability of an A coming just after an A is $P(A|A) = 1/3$, and the probability of B following A is $P(B|A) = 1/3$. The fact that the sequence might as well end after an A accounts for the remaining third of probability. Given the conditional probabilities derived from the tree, the total probability of a given sequence or its continuations can be determined.

### Hierarchical Markov Chains

Given a sequence of previously played notes, the Continuator tries to find a continuation of that sequence. In order to do so, it uses variable-order Markov Chains at different levels of detail. On the finest possible level, states are (pitch, duration,

volume)-tuples. At a very coarse level, whole pitch regions serve as states for the model. Building and using several models on these different detail levels significantly increases the chance that the system finds a continuation for any seed sequence played by the human player.

## 3.4 Summary

In this chapter, the object-oriented representation of musical data described in [Pachet, 1993] has been presented. Unfortunately, this framework is implemented in Smalltalk and does thus not address as wide an audience as it could. Moreover, the subclassing of the `Note` superclass into five subtypes seems unnecessarily complex and involves a limitation of note names to 35 possibilities, which is also not necessary.

The approaches to chord sequence analysis presented above are interesting — especially [Pachet, 2000] served as a starting point for development — but those approaches are either not specifically designed for scale selection, or they are not explained in sufficient detail, such as [Nishimoto et al., 1998]. The optimization approach by Choi [2004] was the only reproducible approach intended for scale selection, but showed serious deficiencies, as discussed in chapter 6.

The solo generation techniques from the CHIME system [Franklin, 2001], BoB [Thom, 2000] and the Continuator [Pachet, 2002] show how statistical models and neural networks can be used to determine suitable improvised melodies. Such techniques could be adapted in the future to support and improve human improvisation.

# 4. Architecture and Design

The following chapter describes the architecture of the back-end and the design choices made along the way. In the first section, the main goals of the framework are formulated. The subsequent sections describe the actual design of different aspects of the framework.

## 4.1 Goals

In order to provide an overview of the tasks that were faced during the development of coJIVE, the following section outlines the main goals of the back-end framework.

### 4.1.1 Note Classification

The main task of coJIVE's back-end module is to classify notes within their harmonic context, i.e., to tell playable notes and unplayable notes apart. The front-end uses this information to map input from different kinds of input devices to output notes.

#### 4.1.1.1 Harmony Analysis

Notes are classified using analysis methods from jazz theory, used by many musicians to determine scales and notes that fit chords. Since chords cannot be viewed as isolated units, but have relations to each other, and also because chord symbols are often stated in a way that does not directly imply all the possible notes, automating the analysis process is a non-trivial task. Typical chord patterns found in most jazz tunes are used to determine larger harmonic contexts and infer sets of playable notes.

This work can be said to stick closely to the Berklee jazz theory, which is well-established among jazz musicians, although it is far from absolute and considered a bit out-dated for the current jazz world. Jazz music defines itself through improvisation and trying out new things — it is constantly evolving. Thus, it seems impossible to fully capture all the possibilities in a theoretical model. However, when limiting the scope to a particular subset of jazz, it is possible to express many phenomena in theoretical terms. For the work at hand, some theoretical information was drawn from [Jungbluth, 1981], which was one of the first books to import the Berklee theory to German-speaking countries.

#### 4.1.1.2 Musical Data Structures

As a prerequisite for performing harmony analysis, suitable data structures must be provided that handle the relevant musical information.

### 4.1.2 Selection of Voicings

In order to play more than just single notes, it is also necessary to have a method of translating chord symbols into actual chords. Chord symbols do not contain information about the arrangement of voices within a chord, which is typically possible in many different ways. This task also depends on the results of harmony analysis, as chord symbols only specify a subset of possible notes, and limiting the chord voices to these few notes usually does not sound very interesting.

### 4.1.3 Song Management

The task of acquiring and managing the song information was delegated to the back-end module. Songs are loaded, preprocessed and stored within the framework for analysis, and are later relayed to the front-end as they are requested.

#### 4.1.3.1 Lead Sheet

Lead sheet information (especially chord symbol data) is not stored in common music file formats. A file format had to be developed to store chord symbols along with their onset times. Other song information such as tempo and meter should be included in that format. Since both the back-end and the front-end were a work in progress, the format had to be extensible and human-readable, so more information could easily be added by hand without breaking old behavior.

#### 4.1.3.2 Theme and Bass Line

In almost every jazz performance, the main theme is played in the beginning and in the end of the tune. Trained musicians have the theme running through their heads during their solos. For coJIVE, which confronts untrained users with jazz, it seemed a good idea to adopt the convention of theme playing, because it provides them with initial ideas for their improvisations. Since good bass lines are also essential to the inspiration of other musicians in a jazz combo, and the main focus of this work did not lie on the generation of these bass lines (although some basic scheme was implemented in coJIVE, see [Buchholz, 2005]), the choice was made to use predefined bass lines.

Advantages for the interactive experience put aside, melody and bass information can potentially contribute to the accuracy of the analysis results. Consequently, a means of loading this information had to be provided.

## 4.2 Musical Data Structures

In order to process the musical information on a high level, special data structures are required to represent this information in an intuitive way. Many concepts from [Pachet, 1993] were adopted and in part reimplemented in C++. The data structures can be divided into two general classes of types: The first class includes atomic types, which represent very elementary items like notes and intervals. The second class encompasses container types such as chords and scales, which generally store sets of atoms, but differ in detail.

### 4.2.1 Atomic Types

The fundamental units used in harmony theory are named notes, pitch classes and intervals. Three classes were designed to represent these units and the typical operations between them:

- `CJPitch`
- `CJNote`
- `CJInterval`

#### 4.2.1.1 Pitch Classes

The twelve unique pitch classes are represented by the class `CJPitch`, which contains some basic methods for incrementing and decrementing the pitch by integer numbers of semitones. These additions and subtractions are taken modulo 12, so that only valid pitch classes can result from these operations.

#### 4.2.1.2 Named Notes

Named notes, i.e., octave-independent enharmonic spellings of pitch classes, are represented by objects of the class `CJNote`. It provides some basic methods for their manipulation, such as sharpening/flattening, simplifying the spelling to have one accidental at most, retrieving the natural note that a note is derived from (e.g., G for G♭ or G♯), and calculating the note's pitch class, which is returned in form of a `CJPitch` object.

In addition to these unary operations, some more advanced arithmetic operations interacting with intervals are defined. The relationship between notes and intervals is similar to the relation between points and vectors: An addition operator can be defined between notes and intervals, and another one between intervals, while addition of two notes does not make sense at all. Let $N$ be the set of (named) notes and $I$ the set of intervals, then the following holds:

$$\begin{aligned} n \in N, i \in I \quad &\Rightarrow \quad (n+i) \in N \\ i_1, i_2 \in I \quad &\Rightarrow \quad (i_1 + i_2) \in I. \end{aligned}$$

Hence, it is natural to define an addition operator "+" between notes and intervals. However, subtraction of two notes is not as well-defined: Since note names are not associated with an octave, it cannot be told which one of two notes is higher than the other, let alone how many octaves they are apart. A subtraction operator would require an order of pitch classes, which is not well-defined (a C is not necessarily lower than a D). Therefore such an operator would be unnatural.

The two methods `isIdentical()` and `isPitchEqual()` allow comparison of notes on two different levels. While `isIdentical()` checks whether two notes have exactly the same name, `isPitchEqual()` only checks for matching pitch classes.

In order to deal with MIDI data which does not carry information about enharmonic spelling, a method was defined that allows the construction of a `CJNote` from a `CJPitch`. Since it assigns arbitrary enharmonic spelling, this is not theoretically correct, and the method might be removed or replaced at a later stage of development. Currently the method is used to gain some melody information from MIDI tracks.

#### 4.2.1.3 Intervals

Objects of the class `CJInterval` represent enharmonically spelled intervals. Like `CJNote`, this class provides unary and binary operators and methods for manipulation and retrieval of the actual semitone distance represented by the interval.

Intervals smaller than an octave have an inverse with respect to addition: Transposing a note up by a perfect fifth and transposing it down by a perfect fourth, for example, are equivalent in the context of octave-independent note names. If intervals larger than an octave are inverted, the inverse is not unique any more, and re-inversion will not yield the original interval. It can still be defined, as long as the intervals are used only for arithmetic operations on note names (as opposed to octave-dependent notes). A method `reverse()` was defined to perform this kind of inversion, as well as a negation operator overload that does the same.

Analogously to the addition of notes and intervals, addition of intervals was defined as a "+" operator. Subtraction is still a problem, since subtracting a larger interval from a smaller one would result in a negative interval, which is not supported so far. For example, a major second minus a perfect fifth would be a perfect fourth downwards. Although only upwards intervals are supported, a subtraction operation was defined that simply takes the reverse of the second operator and adds it to the first. For example:

```
  MajorSecond - PerfectFifth
= MajorSecond + PerfectFifth.reverse()
= MajorSecond + PerfectFourth
= PerfectFifth
```

Obviously, the result (a perfect fifth) is not the correct solution (a perfect fourth downwards). However, adding this incorrect difference to a note yields the same result as the correct difference would, since only pitch classes, not real pitches are used. Since the result of `reverse()` is always non-negative and only defined for intervals smaller than an octave, this operation is somewhat obscure and does not yield the correct difference of two intervals. Thus, the operator is currently still present in the class as a convenience function.

### 4.2.2 Container Types

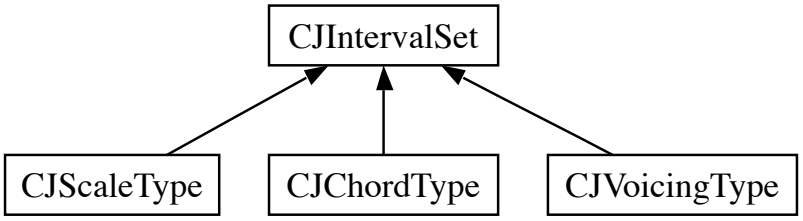
All the larger structures such as chords and scales are generally sets of intervals or sets of notes. Since set operations on either kind of sets are identical among all of their specializations, two superclasses were defined that provide these methods:

- `CJIntervalSet`
- `CJNoteSet`

#### 4.2.2.1 Sets of Intervals

The class `CJIntervalSet` stores an ordered set of intervals and provides some basic set operations, such as adding or removing elements, joining with other interval sets, and checking for the presence of specific elements. The set is ordered by ascending semitone distances. The semitone distance is also the key for this container, so it is not possible to add two different intervals that describe the same actual distance. This makes sense in most cases, since no scale, chord or voicing would ever contain two differently named intervals of the same length in semitones. In addition to the set operations, `CJIntervalSet` provides a method `buildOnRoot()`, which populates a note set with the notes resulting from the addition of each interval in the original set to a given root note.

Interval sets describe the properties of scales and chords independently from actual pitches. Consequently, the following three classes were derived from `CJIntervalSet`:




Objects of all of these classes can be constructed from simple text strings that describe the structure of a particular scale, chord or voicing, respectively. The format of these strings is different for each of these classes, since standard notation is different for each. The classes also differ in the additional domain-specific methods they provide.

#### 4.2.2.2 Scale Types

Objects of the class `CJScaleType` describe the structures of scales. Their main constructor parses text strings consisting of degree numbers 1 through 8 with arbitrarily many accidental prefixes. As an example, the structure of the major scale can be constructed from the string `"1234567"`, and the string `"12b34567"` yields the melodic minor scale structure. The most important scale types (see Figure 2.1) are already defined as static members of the `CJScaleType` class for convenience.

The only purpose of the `CJScaleType` class is the construction of actual scale objects (see 4.2.2.6), so it provides no additional methods.

#### 4.2.2.3 Chord Types

An object of the class `CJChordType` describes the structure of a chord, independent from its root note. The strings parsed by the constructor are typical text notations of chord symbols with the root note omitted, e.g., `"maj7 #11"` generates a major seventh chord (containing a major third, a perfect fifth and a major seventh) with an augmented eleventh interval added.

In the course of harmony analysis, information about the general harmonic qualities of chord types is often required. To simplify this task, methods such as `isMajor()`, `isMinor()` or `isDiminished()` encapsulate the queries for the presence of the intervals defining those qualities.

#### 4.2.2.4 Voicing Types

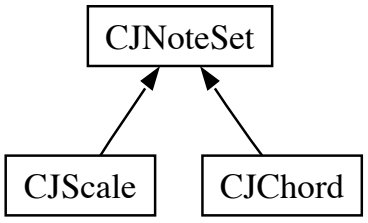
Objects of the `CJVoicingType` class describe the structure of voicings. The text strings used for construction consist of space-separated degree numbers (integers greater than 1), with optional prefixed accidentals to signify deviation from the major degree. For example, a voicing type that describes a root voicing of a major seventh chord would be constructed from the string `"1 3 5 7"`, a dominant seventh voicing standing on the minor seventh and having an added 9th and 13th is described by `"b7 9 10 13"`.

The purpose of `CJVoicingType` is the construction of actual voicing objects (see 4.2.2.8). Additionally, a helper method `buildOnInteger()` is provided, which builds a voicing on top of a MIDI root note.

#### 4.2.2.5 Sets of Notes

The class `CJNoteSet` stores an ordered set of notes and contains some methods for basic set operations, such as adding or removing elements, joining with other interval sets and checking for the presence of specific elements. Additionally, methods for set inclusion testing and set difference computation are provided. The set is ordered by ascending pitch. The pitch is also the key for this container, so it is not possible to add two different notes that have the same pitch. Analogously to the `CJInterval` class, this makes sense, since no chord or scale would contain the same pitch in more than one spelling. The methods `hasPitch()` and `hasNote()` provide means of checking for the presence of a note at different levels of precision.

`CJNoteSet`s are mainly used to describe which notes are contained in a chord or scale. Actual instances of scales and chords are represented by classes derived from `CJNoteSet`:




Objects of these classes can be constructed from a root note and their respective type. These two classes are used for harmony analysis and scale selection.

Making the distinction between a scale/chord type and an actual instance of such a type was necessary, because the pure “set of pitch classes” representation lacks ordering information. Especially chords and voicings could not be stored as a set of pitch classes only, since the order of intervals determines their sound characteristics. For example, an A note occurring in a C major chord (C,E,G) could be seen either as the sixth or as the thirteenth interval with respect to C, and those two have fundamentally different functions. The thirteenth, located above the seventh, produces tension (because it forms an additional minor seventh interval with the seventh), while the sixth, forming rather “harmless” intervals with the other chord notes (major 2nd with G, major 4th with E), produces relaxation and the impression of a harmonic conclusion. Whenever information of that kind needs to be accessed, the type object is the place to look for it.

#### 4.2.2.6 Scales

A scale, as mentioned before, is an actual instantiation of a scale type in form of a pitch-ordered set of notes. `CJScale` objects are generated on the fly to determine a set of playable notes in the harmony analysis module. Besides the constructor and getter methods for the root note and type object, the `CJScale` class adds no functionality to its superclass `CJNoteSet`.

#### 4.2.2.7 Chords

The class `CJChord` represents actual instantiations of chord types with root notes. Objects can be constructed in three different ways: by specifying a root note and a chord type object, by specifying a root note and a symbol string describing the type (e.g., `"7 b9 #11"`), or by simply passing a complete chord symbol (e.g., `"C 7 b9 #11"`). The latter two methods create a type object from scratch, while the first just copies the one specified. The chord class exposes the quality determination methods (such as `isMinor()`, `isDiminished()`) as the chord type class and simply relays them to the internal `CJChordType` object. Additionally, it provides a method to retrieve the symbol of the chord, which is used for text output.

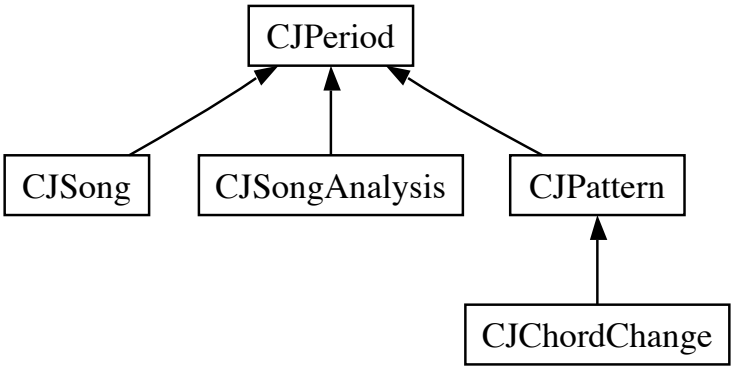
Finally, a method used during the evaluation of the minimization approach from [Choi, 2004] is provided: `findPossibleScales()` generates, from a given set of scale types, all scales that contain the root note and match the pitch classes of all other chord notes. Per default, the set of types is the set of all scale types known to the system.

#### 4.2.2.8 Voicings

Unlike `CJScale` or `CJChord`, the class `CJVoicing` is not a set of `CJNote`s. It only stores the root note and the voicing type it was created from. The class exposes a method `getIntegerPitches()` which fills an array of integers with the lowest possible MIDI notes that describe the voicing. This functionality is needed by the front-end, which selects the octave and adds the according MIDI note value to all the pitches in the array.

## 4.3 Temporal Data Structures

Since chord progressions and songs are temporal sequences of symbols, data structures for time and temporal objects are required. `CJTime` represents times or durations in form of a bar/beat/tick representation. It provides arithmetic operations such as addition and subtraction of times. The `CJPeriod` class stores a start time and a duration and also a list of sub-periods that the period is composed of. `CJPeriod` serves as a superclass for objects that have temporal extents. The inheritance from `CJPeriod` looks as follows:




### 4.3.1 Pattern Objects

In the first version of the back-end, for each recognized chord pattern, an object is created that is assigned the temporal extents of the sequence of chord changes which it matches. The classes of these objects are derived from the abstract class `CJPattern`, which provides an interface for querying votes on playable notes at a particular time within the pattern's time span.

### 4.3.2 Chord Changes

The framework defines chord changes to be a combination of a chord object and a period of time that it spans in the song. Also, a chord change can serve note suggestions where the analysis process was not successful in finding a scale - notes in the chord are generally playable. This is why the `CJChordChange` class inherits from `CJPattern`, which provides the interface for the note querying routine. `CJChordChange` aggregates a chord object and, for coding convenience, forwards those of `CJChord`'s methods that are relevant to the analysis process.

### 4.3.3 Songs

For this project, the most important aspect of a song is the sequence of chord changes, which is why `CJSong` inherits from `CJPeriod` and stores the chord change objects in the collection maintained by this superclass. The song class exposes a method that can load a lead sheet from disk, populating the internal collection with chord changes. If MIDI files for a theme and a bass line are specified in the lead sheet file, these are also loaded automatically. A song object also contains some general song properties, such as the name, meter and tempo of the song.

### 4.3.4 Analyses of Songs

The `CJSongAnalysis` class serves as a container for recognized patterns. Its constructor is fed a song object and performs the harmony analysis immediately. At a later point, the analysis object can be queried for note suggestions at a particular song position, which are gathered from the stored patterns. Note suggestions are returned in the form of histograms or probability distributions over all possible pitches (see 4.5.1).

## 4.4 Song Management

The central `CJFramework` object maintains a state, which includes a pointer to the current song (a `CJSong` object). Once the framework is requested to load another song, the old song is deleted from memory. This behavior has been implemented for ease of use through the front-end, which also has only one current song. Of course, other users of the framework can load and keep as many songs in memory as desired, if they do not go through the central framework object.

### 4.4.1 Lead Sheet

The lead sheet consists of several chunks of data, the most important one being the sequence of chord changes.

#### 4.4.1.1 XML File Format

An XML (eXtensible Markup Language) file format for the chord sequence was defined. XML has the advantages of being human-readable (which is useful during development and debugging) extensible and downward-compatible, since defining new tags is always possible and does not break old loading routines. A lead sheet file contains the following information:

- song title
- tempo mark (beats per minute)
- time signature (meter)
- style tag (this could influence the style of accompaniment or the analysis process in future versions of the program)
- sequence of chord changes with onsets and/or durations
- optional links to theme and bass files

### 4.4.2 Theme and Bass Line

The theme melody and the bass line are needed by the front-end for accompaniment and theme playback. In the back-end, some provisions have been made to utilize melody data to improve analysis and note classification. The system acquires theme and bass data from two separate MIDI files which are referenced in the lead sheet file. The free MIDI I/O library *libjdkmidi* is used for MIDI parsing.

The decision fell on the MIDI format, despite the fact that MIDI has a serious drawback: its support for enharmonic spelling is very bad. Nevertheless, it is a widespread format – many free tunes can be obtained on the web, and files can be easily created using freeware tools such as *Rosegarden*.

In the second version of the framework, the MIDI track of the melody is traversed to determine which melody notes are played over each chord change. This information is associated with the `CJChordChange` objects in the `CJSongAnalysis`.

Currently, the MIDI multi-track objects generated by *libjdkmidi* are not converted to a special internal format, but exposed as-is through getter methods in `CJSong`.

## 4.5 Note Classification

A major goal of this work is the classification of notes through harmony analysis. Given the harmonic grid of the song and the user input, the system tries to find notes that are suitable in each situation within an improvisation. In the beginning of the project, the desired output of the algorithm was defined to be a probability distribution over the pitch classes. Since this output format cannot properly model melodic lines crossing multiple octaves, it was later changed to a distribution over the 128 possible MIDI pitches.

### 4.5.1 Why Probabilities?

Strictly speaking, notes can never be separated into right or wrong. At best, a system trying to analyze a chord structure can make suggestions on what is possible or more probable than something else. Probabilities seemed to be the appropriate output format for note classification algorithm. Probabilities are very flexible and generic for this application, since they allow both exact and fuzzy statements to be made about note playability.

The front-end part of the project aimed at supporting different sorts of input devices, some of which are inherently fuzzy, i.e., the user cannot aim at exact positions due to the nature of the device. Probabilities support this kind of behavior, as opposed to binary statements about playability of notes.

Another aspect advocating the decision of using probabilities is coJIVE's support for different levels of expertise. Ideally, a less obvious note has a lower probability assigned than others and is unlikely to be intended by a novice player. Experienced players tend to use the less probable notes to connect the obvious ones. The front-end can use the probability distribution to further classify notes according to the different levels of proficiency.

Furthermore, some systems exist that build statistical models out of real performances to either analyze melodies or predict melodic continuations based on previously played note sequences (e.g., the Continuator, [Pachet, 2002]). The probability format allows the easy integration of such approaches into the framework. Due to the requirement of gathering large amounts of real-world data from professional musicians in order to generate such models, these approaches were beyond the scope of this work.

### 4.5.2 Pattern-Based Harmony Analysis

The analysis process is based on chord patterns commonly appearing in jazz standards. Recognizing patterns means understanding the context and function of chords, which is why professional jazz musicians are usually trained in music theory. In every version of our software, patterns are recognized to infer scales or singles notes that can or cannot be played over the individual chords in such patterns. The two versions of the software differ in how they make the final decision on the probability of each note.

### 4.5.3 Note Classification using Additive Probabilities

The original idea for the harmony analysis algorithm was based on the fact that no analysis can never be considered an absolute authority for choosing the right notes, and that it is often ambiguous in its results. The amount of chord sequences that become patterns is constantly growing, and so a pattern database can never cover all of the possible cases. The first approach that was tried should deal with ambiguous cases in a way that allows at least to rate the probability of different notes: When different patterns were recognized covering the same chord change, each pattern should contribute to the total probability distribution. The idea was to build a histogram over the 12 pitch classes, counting the votes on notes from the different patterns, and then normalizing the result to yield a probability distribution in the classical sense.

#### 4.5.3.1 Pattern Recognition

In the first phase of the analysis, a set of rules is applied at each position in the chord sequence of the song. Whenever a rule matches, a `CJPattern`-derived object is created and recorded in the `CJSongAnalysis`. A rule class is derived from a superclass `CJRule`. The rules that were implemented check consecutive chords for their individual harmonic properties, as well as for the movement of the root note. If the sequence meets the criteria of the rule, an object is created and returned to the calling analysis object, which then records it. Additionally, the chord changes themselves are treated as recognized patterns, which guarantees at least some information about notes where pattern matching fails. As a side effect, the probabilities of chord notes are raised above those of mere scale notes, which is a reasonable behavior — musicians usually construct their melodies around the chord notes, using them as resting points. Figure 4.1 shows a sample output of this scheme.

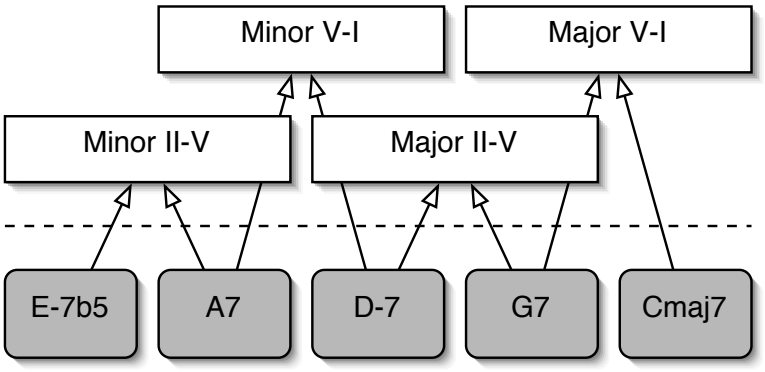



Figure 4.1: A sample pattern recognition output, where multiple patterns partly overlap and share chord changes.

#### 4.5.3.2 Probability Generation

Probabilities are generated during the playback of the song. The front-end requests the probabilities, typically when the song position changes to the next chord. The song analysis is searched for recognized patterns that cover the current position, and each pattern is queried for its votes on pitch classes. These votes are counted in a

histogram, which is then normalized and returned to the caller. Using this scheme, notes that are favored by more patterns receive a higher rating. Figure 4.2 illustrates this concept.

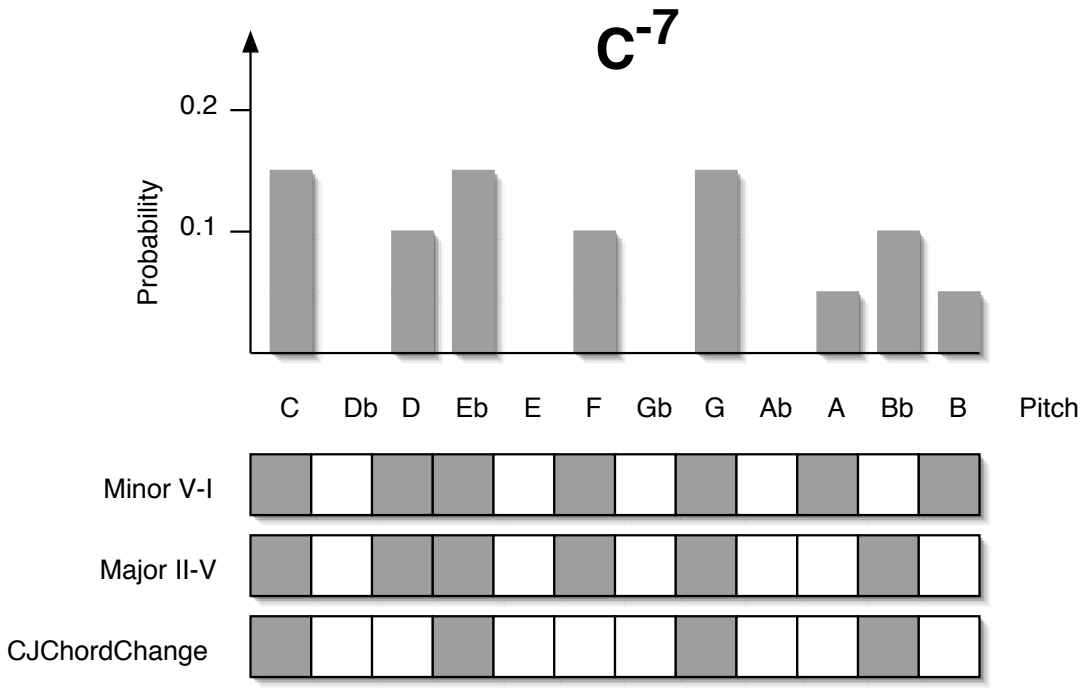



Figure 4.2: The concept of additive probabilities. This example shows how pitches are rated in a situation where a $C^{-7}$ chord is found to have the function of a II and a I at the same time. The contribution of the chord change itself raises the probability of chord notes above the others.

### 4.5.4 Note Classification using Interdependent Patterns

Due to problems with the first version of the analysis module, another approach was tried, using a more strict form of analysis. This proceeds in two phases: First, patterns are recognized to label each chord with a roman numeral. In the second phase, suitable notes are determined based on the previously assigned roman numeral.

#### 4.5.4.1 Phase 1: Pattern-Based Roman Numeral Assignment

When performing harmony analysis, it is an established technique to first look for candidate tonic chords, which are major (usually non-dominant) or minor chords. Such a candidate is often prepared by a preceding dominant $V$ chord or substitutes (e.g., $\flat II^7$ or $\flat VII^7$). A dominant or major seventh chord can be identified as a $IV^7$ or $IV^{maj7}$ if it follows a tonic and its root is a fourth away from the tonic's root. Similarly, chords on other degrees can be identified by confirming them through predecessor or successor chords. In order to escape the very predictable progression of dropping fifths (I–IV–VII–III–VI–II–V–I–...), chords are often substituted by others, e.g., using tritone substitution, or the current key is changed. Since such considerations seem to rarely require looking back or ahead by more than one symbol, a valid roman numeral assignment can mostly be obtained by recognizing very small, correctly prioritized patterns. In most cases, a pattern length of two is sufficient to

correctly identify all roman numerals of a jazz standard. The list of most common patterns in [Jungbluth, 1981] (see Table 2.4) was used as a reference.

As each rule is applied to each position in the lead sheet, chord changes are labeled with the roman numerals implied by the recognized patterns. The look-ahead wraps around at the end of the piece, since most jazz pieces are written in a circular manner to allow arbitrarily long group improvisations. The so-called "turn-around", a sequence which occurs often at the end of a tune, can provide important information about the very first chord of the piece and vice versa.

Recording the recognized patterns in a data structure is no longer required. Each chord has exactly one roman numeral assigned, which can be overwritten at any time during the analysis. It depends on the behavior of the pattern whether or not it overwrites existing labels. Since many patterns depend on prior recognition of other patterns, the whole set of rules is repeatedly applied to the chord sequence until no more labeling occurs. Efficiency seems to be no issue with this approach, since the possible number of chord changes in a typical 32-bar song is bounded. Of course, termination of this algorithm is only guaranteed if patterns are implemented carefully, i.e., circular dependencies of patterns must be avoided.

The roman numeral labels assigned are actually references to objects whose classes implement the interface `CJRomanNumeral`. The framework uses this interface to query roman numerals for suggestions on playable notes.

#### 4.5.4.2 Phase 2: Scale Determination

For any roman numeral, there is a relatively small number of possible scale assignments. Thus, in the second phase, a suitable scale is determined based on the assigned roman numeral. Additional rules and constraints are used to resolve any remaining ambiguities.

#### 4.5.4.3 Probability Calculation

The calculation of probabilities in this version proceeds similarly to the one in the previous version. A simple histogram is created from the note suggestions that the roman numerals deliver, which is then normalized into a probability distribution over the twelve pitch classes.

### 4.5.5 Input-Dependent Probability Modulation

As trumpet player Miles Davis put it: "There are no wrong notes". This statement is often cited by jazz musicians, not only to defend bad solos. It is a fact that any note can be played, and even if the musician did not mean to play a note in the first place, there is always a way to turn that note into something meaningful. Furthermore, playing "inside" (i.e., scale notes) only often does not sound very interesting. One way to account for this fact is allowing "outside" notes, but enforcing their immediate resolution. This idea is similar to the one implemented in the reinforcement learning algorithm in [Franklin, 2001], where so-called "hip" notes (alterations like ♭9 or ♯11) are rewarded by the critic, unless one has been played within the last two bars. "Hip" notes are allowed only once in a while.

At the current stage of development, the system supports playing chromatic scales (i.e., playing up or down semitone-wise). The result from the harmony analysis is

extended to an initial probability distribution over the 128 MIDI notes. If one of the more probable notes (those with a higher probability than 1/128) is played, the probabilities of neighboring improbable notes are raised for the next probability query from the front-end. If an improbable note is played, the initial distribution remains unchanged. In that case, the front-end will automatically enforce the next note to be one of higher probability.

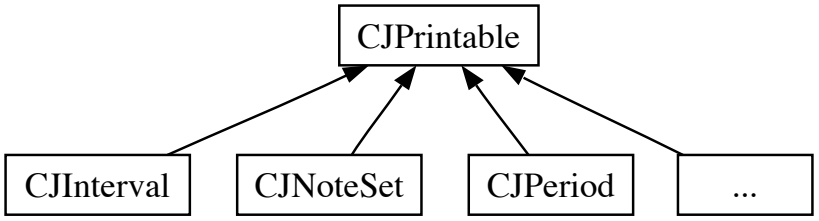
Currently, only the last output note is taken into account for probability modulation, but larger histories are possible. More sophisticated means of melody guidance (e.g., using a database of melodic patterns) could be implemented in a similar fashion in future framework versions.

## 4.6 Voicing Selection

In order to select suitable voicings as they are requested by the framework, a rather simple scheme is used. A list of predefined voicings is traversed, and each voicing type is built onto the current root note. Resulting voicings are cross-checked with the set of playable notes. If a voicing consists only of notes from this set, it is selected for output. Voicings are sorted by ascending lowest pitch, since the front-end assumes the first voicing to contain the root note. In order to get the more probable voicings closer to the top of the list returned to the front-end, the selected voicings are also sorted by ascending mismatch with the notes of the current chord.

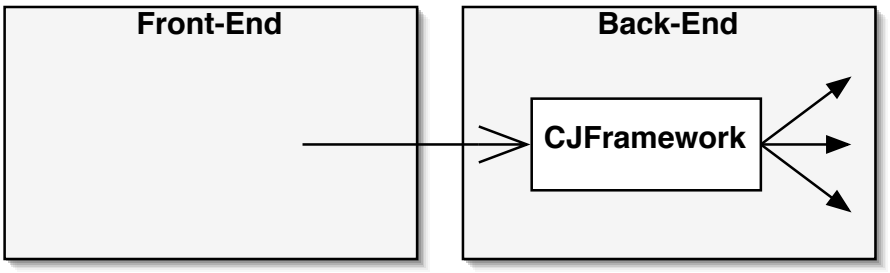
## 4.7 Text Output

The only visual output of the back-end is debug output on the console. In order to simplify this text output, each class that stores relevant information should be able to output itself to the console. Classes of this kind (actually most classes) are derived from the interface `CJPrintable`. This interface prescribes a method `printTo()` to be implemented by subclasses. It is meant to be overridden at each inheritance level to gradually refine the output of the base class method.




## 4.8 Central Framework Class

All the high-level services described above, such as song loading and note classification, are made accessible through a central class named `CJFramework`. This class serves as an interface between the back-end and the front-end, hiding the back-end's internal processes from the front-end, which creates a `CJFramework` instance once on startup and uses it for all further queries to the back-end.




The framework class maintains an internal state that consists of the current song, the current analysis and the input-independent note distribution. Furthermore, probability distributions and note histories of multiple players are stored. The latter are created on the fly as the front-end asks for the distribution of a new player.

# 5. Implementation

## 5.1 Environment

The following section describes the hardware and software components that were used during the development of coJIVE, and it explains why some of these components were chosen over others.

### 5.1.1 Hardware Environment

coJIVE runs and was developed on Apple PowerMac G5 dual processor machines with 2 GHz CPU clock frequency. Some back-end development was done on a standard PC with an Athlon Thunderbird 1 GHz processor. The complete system was tested using input devices such as a Buchla Lightning II infrared baton system [Buchla, 1995], an M-Audio Radium49 MIDI keyboard with 49 unweighted keys, a Viscount VIVA 76-key stage piano [Viscount, 2003] with half-weighted keys and a fully-weighted 88-key digital piano from Casio. Input devices were connected through an M-Audio USB MIDIsport 2x2 interface. For additional visual feedback, the Teleo prototyping tools were used to control external LEDs from the application.

### 5.1.2 Software Environment

Both parts of the software were developed under Mac OS X and Apple's Xcode integrated development environment using gcc 3.3. The back-end was also partly written under Debian Linux, KDE and the KDevelop IDE.

The coJIVE front-end is a native Mac OS X application written in Objective-C. It makes use of Apple's Cocoa GUI framework and the CoreMIDI operating system component. Audible MIDI output was achieved using an additional tool called SimpleSynth, which relays MIDI events to the internal software synthesizer of Mac OS X. However, coJIVE can send its MIDI output to any software which registers as a MIDI destination to the OS. For more information, see [Buchholz, 2005].

While Objective-C was best suitable for coJIVE's interactive front-end (due to the fact that Cocoa is written in Objective-C), the back-end was implemented in C++.

The reasons for this decision include the overall efficiency and portability of C++, the excellent support of portable, type-safe and efficient container classes through the C++ Standard Template Library (STL), and the compact and easy-to-use file and string I/O routines in the iostream library. Objective-C is not as widespread, nor are its container classes type-safe. Since Objective-C binds method calls at runtime, whereas the C++ templates in the STL are possibly even in-lined, C++ is the better choice for algorithmic tasks that use containers. Mixing C++ and Objective-C is generally possible, although not exactly clean. The current version of the back-end is compiled into a Cocoa Framework, which simplifies linking and exporting header files to the front-end. However, the framework files can easily be linked into a BSD static or dynamic library, which was successfully tested in the Linux environment. Using KDevelop [Various Authors, 1999–2005] as development environment, it is easy to create a portable source distribution which can be built with standard Unix tools like autoconf, automake and make. Additional third-party libraries used are TinyXML [Thomason, 2000–2005], a free object-oriented non-validating XML parser, and libjdkmidi [Koftinoff, 2004], an open-source MIDI file parsing library. Browseable HTML source code documentation was generated using doxygen [van Heesch, 1997–2005].

## 5.2 Musical Data Structures

The following section describes implementation details of the basic musical data structures used for the analysis engine. Particularly the calculations between notes and intervals involve non-trivial arithmetic, which will be explained in the following paragraphs.

### 5.2.1 CJInterval

The `CJInterval` class represents positive intervals between named notes. It stores a degree number ($\geq 1$) and the actual interval size in semitones.

#### 5.2.1.1 Name Determination

An interval can calculate its textual name consisting of one of the prefixes `"Perfect"`, `"Major"`, `"Minor"`, `"Diminished"` and `"Augmented"`, and the interval degree. The interval degree is output as an arabic number, not as an ordinal number word, since intervals of arbitrary size should be supported. For example, the text name of a perfect fifth interval would be `"Perfect 5"`.

An ID of the prefix, which is of the enumerated type `CJInterval::Accidental`, is determined by the `accidental()` method. This method uses a one-based modulus-7 computation, which reduces the interval degree to a value between 1 and 7, and then distinguishes intervals for which different prefixes are applicable in a `switch`-statement. A nested `switch`-statement determines the actual prefix based on the difference of the interval's actual semitone distance and the semitone distance of the major/perfect interval of the same degree (modulus 8). Intervals whose semitone distance differs so much from the Major/Perfect version of the interval that they would go beyond the terms "augmented" or "diminished" are classified as "undefined".

### 5.2.2 CJNote

The `CJNote` class stores a note name in the form of a natural note name and an integer value counting the number of accidentals. The natural note name is a class-scoped enumerated type `CJNote::NaturalID` with seven possible values, namely C, D, E, F, G, A and B. The accidental value is positive for sharp signs and negative for flat signs. An accidental of zero means that the note is natural. For convenience and readability, accidental values from -2 to +2 are predefined in an enumerated type `CJNote::Accidental`, using the names `Flat`, `Sharp`, `DoubleFlat` and `DoubleSharp`. A note object can be constructed in several ways:

- from a natural note ID and an accidental value, e.g., `CJNote note(CJNote::E, Flat)`
- from an existing note object, using the copy constructor
- from a note name in text form, e.g., `"Dbb"` or `"G#"`

#### 5.2.2.1 Pitch Calculation

A `CJNote` can return its own pitch class via the `CJNote::pitch()` method. It looks up the pitch class ID of the natural note in a private static array `pitchOfNaturalNote[]`, constructs a `CJPitch` object from this ID, and adds the accidental value using `CJPitch`'s overloaded addition operator.

#### 5.2.2.2 Note-Interval Arithmetic

`CJNote` overloads the addition and subtraction operators `+=` and `-=` addition and subtraction operators, as well as their constant versions `+` and `-`. The actual computation is implemented in the `+=` operator, which is used by all the other operators. The operator method adds the interval's degree (minus one) to the natural note ID modulus 7 in order to reach the new natural note. By this operation, the note travels a certain number of semitones up. To reach the desired pitch, which is determined by the semitone size of the interval, the accidental is corrected by the difference between the traveled pitch distance and the semitone count of the interval. Using this scheme, enharmonic spelling is preserved, so that a C♯ plus an augmented eleventh results in an F♯♯ (instead of, say, a G).

## 5.3 Lead Sheet Files

The XML file format for lead sheets is still a work in progress. The following list describes the current assignment of XML tags:

**<song>** The top-level element of the XML file. It has the attributes **name**, **theme** and **bass** describing the song title, and the paths to files containing the theme and bass line.

**<style>** A tag to describe the style of the tune. This might be handy for future versions of coJIVE. Currently, only the **id** attribute is parsed and stored in memory, but is not used.

**<meter>** The time signature of the song. Currently, this is only parsed once within the <song> environment, i.e., the time signature is global and cannot be changed in the middle of a song. Attributes parsed are **count** and **unit**, which receive integer values and stand for the numerator and the denominator of the time signature.

**<tempo>** Specifies the tempo of the song. Currently global for each song, like <meter>. The only argument parsed is **bpm**, which receives an integer value specifying the tempo in beats per minute. A beat is generally considered the unit given by the denominator of the time signature (<meter>), e.g., 100 bpm in $\frac{n}{2}$ time means 100 half notes or 200 quarter notes per minute.

**<leadsheet>** This tag encloses the sequence of chord changes. It does not know any attributes.

**<cc>** A chord change. Text enclosed by <cc> and </cc> is the chord symbol. The **beats** attribute is an integer specifying the duration of the chord in beats. Attributes other than **beats** might be added in the future to increase temporal resolution.

A typical lead sheet file looks as follows:

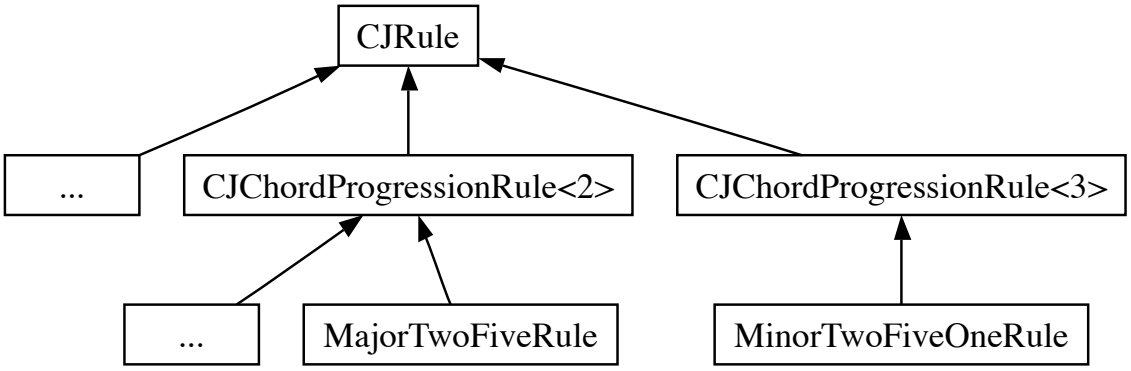
```
<song name="All The Things You Are"
    theme="AllTheThings.mid" bass="AllYourBass.mid">
    <style id="swing"/>
    <meter count="4" unit="4"/>
    <tempo bpm="160"/>
    <leadsheet>
        <cc beats="4">F -7</cc>
        <cc beats="4">Bb -7</cc>
        <cc beats="4">Eb 7</cc>
        <cc beats="4">Ab maj7</cc>
        <cc beats="4">Db maj7</cc>
        <cc beats="4">G 7</cc>
        <cc beats="8">C maj7</cc>
        ...
    </leadsheet>
</song>
```

The `<cc>` tags define chord changes, and their attribute `beats` specifies their duration in beats. At the current stage of development, we do not deal with specialties such as chord-less periods or chord onsets on half-beat times, so specifying only the duration in beats with each chord (instead of exact onset times) is sufficient to describe the temporal structure of a chord sequence. The open-source library *TinyXML* is used to parse the XML files. All parsing code is currently located in the private `readLeadSheet()` method in `CJSong`, which is called by the public `readFromXML()` method. `readFromXML()` also calls `readTheme()` and `readBass()`, which load the according data from MIDI files. Additionally, the method `storeMelodyNotes()` is used to parse the theme track and store melody notes in the chord change objects over which they occur.

## 5.4 The First Analysis Engine

The following section deals with central components of the first version of the analysis engine. It explains in detail how rules are matched, and how this matching is used to classify notes in terms of their fitness for underlying harmonies.

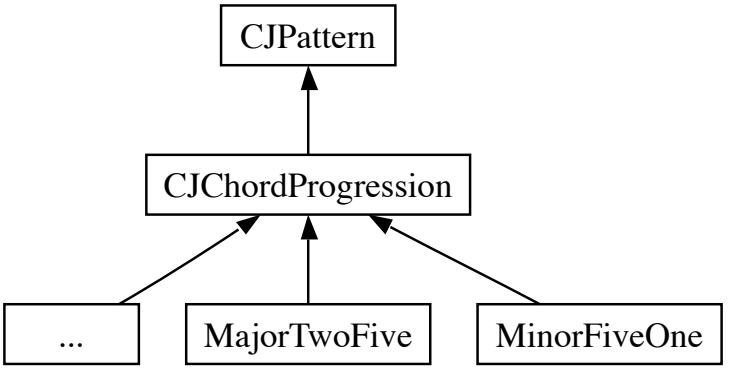
### 5.4.1 Rule Matching




For rules that deal with chord progressions (which are in fact the only rules that have been formulated so far), an intermediate class template `CJChordProgressionRule<n>` was implemented to facilitate access to the part of the temporal collection that should be matched. It overrides the general `apply` method of `CJRule`, in which it upcasts n successive entries in the song's `CJPeriodCollection` from `CJPeriod` pointers to `CJChordChange` pointers, which are then passed to an overloaded virtual method `apply()`. Actual rule implementations inherit from `CJChordProgressionRule<n>` and implement its overloaded `apply` method. A typical rule is shown in Figure 5.1.

### 5.4.2 Note Classification

The probability distribution over the 12 pitch classes is computed as it is queried. To find the distribution at a time $t$, all `CJPattern` objects (created in the rule matching pass) overlapping $t$ are queried for their suggestions on playable notes using the virtual method `getNotes(...)`. These votes are collected in a histogram and then normalized to yield a probability distribution.



```
static class MajorFiveOneRule : public CJChordProgressionRule<2>
{
public:
    MajorFiveOneRule() : CJChordProgressionRule<2>("V7-Imaj") {}

    CJPattern *apply(const CJChordChange *c[],
                     const CJPeriod::Circulator &_first,
                     const CJPeriod::Circulator &_last)
    {
        if (c[0]->isDominant()     // first chord is dominant
           && c[1]->isMajor()      // second chord is major
           && !c[1]->isDominant()  // but not dominant
           && (c[1]->root().isPitchEqual(c[0]->root()
                              + CJInterval::PerfectFourth())))
        {
            // then it's a V7-Imaj
            return new MajorFiveOne(_first, _last);
        }
        return 0;
    }
}
ruleInstance;
```

Figure 5.1: A typical pattern recognition rule in the first version of the analysis module. This rule checks for a $V^7$-$I^{maj7}$ progression in a major key. It checks the individual properties of each of the two chords and the distance between their root notes.

Similarly to the `CJRule`-derived classes, there is an intermediate class that facilitates some of the work for the case of chord progression objects. This class is called `CJChordProgression` and translates the note query into a simpler form: The original queries ask for notes at a particular time (given in song time format), but usually only the index of the chord change within the current pattern is necessary to select a particular scale. E.g. within a $II^{-7}$-$V^7$ progression, there are only two possibilities for a query: the II chord (index 0) or the V chord (index 1) can be asked for notes.`CJChordProgression` calculates the correct index from the incoming time specification and forwards the query to the method `getNotesAtIndex(...)`, which must be overridden by actual pattern implementations. An example of such an implementation is shown in Figure 5.2.

## 5.5 The Second Analysis Engine

In order to account for some issues that were found in the first analysis engine, a second version was created. The rule-matching process, which differs from the one in the first version, as well as the new note classification scheme are described below.

```
class MajorTwoFive : public CJChordProgression
{
public:
  // constructor, has CJChordProgression's constructor do the work
  MajorTwoFive(const Circulator &_first, const Circulator &_last) :
    CJChordProgression(_first, _last) {}

  // overridden member function: return class name
  const char *name() const { return "IIm-V7"; }

  // get notes for the _index'th chord in the progression
  void getNotesAtIndex(unsigned _index, const ConstIter &_it,
                       CJNoteSet &_notes)
  {
    const CJChordChange *chord =
      dynamic_cast<const CJChordChange *>(*_it);
    const CJNote &root = chord->root();

    switch(_index) {
      case 0:
        {
          // use dorian scale without sixth for first chord
          CJScale s(root, CJScaleType::Major, 2);
          s.remove(root + CJInterval::MajorSixth());
          _notes.add(s);
        }
        break;
      case 1:
      {
        // mixolydian scale without fourth for second chord
        CJScale s(root, CJScaleType::Major, 5);
        s.remove(root + CJInterval::PerfectFourth());
        _notes.add(s);
      }
      break;
    }
  }
}
```

Figure 5.2: A typical implementation of a `CJPattern`. The virtual note suggestion method is overridden by a method that suggests the dorian scale for the first chord of the progression and the mixolydian scale for the second chord. Avoid notes are removed.

### 5.5.1 Rule Matching

The basic inheritance structure of rules is maintained in the second version of the analysis module, but the function of rules is different. While rules create pattern objects in the first version, they assign roman numeral labels to the chord changes in the second one. The return type of `apply()` is no longer a `CJPattern *`, but merely a boolean value that indicates whether the rule actually matched and assigned labels. In the main analysis loop in `CJSongAnalysis::extractPatterns()`, which repeatedly applies all available rules to all positions in the song, this boolean value is used to keep track of whether any relabeling has occurred in one iteration of the outer loop. If nothing is re-labeled after the whole set of rules has been applied to the song, the loop terminates. A typical rule is shown in Figure 5.3.

#### 5.5.1.1 Implemented Patterns

The second version of the analysis engine recognizes most of the patterns listed in Table 2.4. The following patterns are defined in the current version:

**V–I** the common resolution of dominants into tonic chords.

**♭VII–I** another common resolution into the tonic.

**II–V** preparation of a dominant with a minor II chord.

**VI–II** preparation of a II through a $VI^{-7}$ or $VI^{7}$. The recognition of the VI depends on prior identification of the II. VI chords are classically $VI^{-7}$, but can have many substitutes such as $VI^{7}$, $\sharp I^{\circ 7}$ or $\flat III^{-7}$. These substitutes are also recognized by the rule.

**III–VI** preparation of a VI through a III chord.

**IV–♭VII** preparation of the ♭VII through a $IV^{-7}$.

**I–IV** accounts for $I^{maj7}$ $IV^{maj7}$ and $I^{maj7}$ $IV^{7}$. The rule depends on prior recognition of the I in order to identify the IV.

**V–V** adds support for dominant chains.

**I–IV–I** included to support blues, which has dominant tonic chords ($I^{7}$). The rule recognizes the characteristic $I^{7}$–$IV^{7}$–$I^{7}$ pattern in the beginning of the classic blues form.

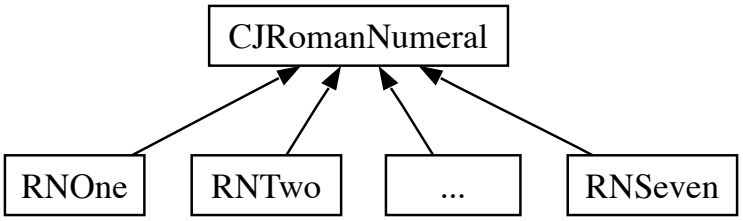
### 5.5.2 Note Classification

The labels assigned in the first pass are actually pointers to static instances of some type derived from `CJRomanNumeral`. These objects are responsible for making the final decision about playable notes in the second pass. They implement `CJRomanNumeral`'s abstract method assignNotes(...)| which is passed a circulator[1] pointing to the chord change in the song. This pointer is used to examine the neighborhood of the chord change.

[1] A circulator is a pointer or iterator that wraps around when incremented at the end or decremented at the beginning of the sequence pointed to.

```
static class TwoFiveRule : public CJChordProgressionRule<2>
{
public:
  TwoFiveRule() : CJChordProgressionRule<2>("II-V") {}

  bool apply(CJChordChange *c[])
  {
    // if second numeral is already labeled, it has to be a V
    if ((c[1]->romanNumeral()
      && c[1]->romanNumeral() != CJRomanNumeral::Five))
      return false;

    // if first numeral is already identified, it must not be a II
    if (c[0]->romanNumeral() == CJRomanNumeral::Two)
      return false;

    // recognize a II-V
    if (c[1]->root().isPitchEqual(c[0]->root()
                                  + CJInterval::PerfectFourth())
      && c[0]->isMinor()
      && c[1]->isDominant())
    {
      // label chords with II and V
      c[0]->setRomanNumeral(CJRomanNumeral::Two);
      c[1]->setRomanNumeral(CJRomanNumeral::Five);

      return true;
    }

    return false;
  }
}
ruleInstance;
```

Figure 5.3: A typical rule in the second version of the framework. The rule recognizes a II–V progression in minor or major tonality and labels the chords accordingly. The rule must also take care not to assign the same labels twice, because otherwise the analysis would not terminate.




In the current implementation, several roman numeral classes are defined and statically instantiated. These classes are named `RNOne`, `RNTwo`, and so on, all of which implement the abstract `CJRomanNumeral` interface. Their instances are pointed to by static members of the `CJRomanNumeral` class called `CJRomanNumeral::One`, `CJRomanNumeral::Two`, and so on. It is generally possible to implement completely different objects to allow a different kind of analysis. Of course, the first analysis pass would also have to be modified in order to assign and use such new labels. Figure 5.4 shows how a Roman numeral object makes the decision on playable notes in code.

```
static class RNFour : public CJRomanNumeral
{
public:
  const char *name() const { return "IV"; }

  void assignNotes(const CJPeriod::Circulator &_position)
  {
    CJChordChange &c = *dynamic_cast<CJChordChange *>(*_position);

    // decide on scale: mixo#11, lydian or dorian
    if (c.isDominant())
      c.notes().add(CJScale(c.root(), CJScaleType::MelodicMinor, 4));
    else if (c.isMajor())
      c.notes().add(CJScale(c.root(), CJScaleType::Major, 4));
    else
      c.notes().add(CJScale(c.root(), CJScaleType::Major, 2));
  }
}
rnFour;
```

Figure 5.4: An example implementation of a Roman numeral. Here the playable notes for a IV chord are determined. In this case, there are only three possibilities: a dominant IV usually uses the mixo♯11 scale, a major IV uses the lydian scale, and a minor IV uses the dorian scale.

## 5.6 Input-Dependent Probability Modulation

The probabilities calculated by the theoretical analysis module are modulated depending on the players input. For that purpose, the `CJFramework` object maintains a state for each player, keeping a history of the last few notes played. Currently only the last note is used to influence the theoretical distribution to allow for the playing

of chromatic (semitone-wise) lines. In order to make this work, a distribution over 12 pitch classes is not enough any more, since the octave in which a note was played also counts. Hence, the theoretical distribution is replicated over the 128-note MIDI spectrum and re-normalized. If the last note played was a probable note, i.e., one deemed playable by the analysis routine, neighboring improbable notes get a certain non-zero probability assigned. Notes are considered improbable if their probability is below 1/128 (the probability all notes would have in a uniform distribution). If an improbable note is played, the probability remains unchanged, since other notes have a higher probability anyway.

# 6. Evaluation

The most important task of the back-end is performing the harmony analysis and determining note probabilities for improvisation. In the following sections, the output of the analysis routine will be discussed, generated from several tunes that have been encoded in coJIVE's XML format.

## 6.1 The Data Set

Seven songs were transcribed into the coJIVE XML format. These tunes were also used in two of the user studies described in [Buchholz, 2005] and are well-known jazz standards:

1. Fly Me To The Moon (Bart Howard)
2. Solar (Miles Davis)
3. All The Things You Are (Jerome Kern & Oscar Hammerstein II)
4. There Will Never Be Another You (Harry Warren & Mack Gordon)
5. Straight, No Chaser (Thelonious Monk)
6. Giant Steps (John Coltrane)
7. Stella By Starlight (Victor Young)

## 6.2 The First Analysis Engine

In the following section, the performance of the first version of the analysis engine is discussed. Results from the harmony analysis (pattern-matching) process will be presented in the first part, followed by an evaluation of the note classification process. In the last subsection, problems that occurred in the first analysis engine will be explained.

### 6.2.1 Harmony Analysis

The first analysis engine is able to find different II–V and V–I patterns (minor and major). Other patterns were defined, but not effectively used. An example analysis is shown in Figure 6.1.

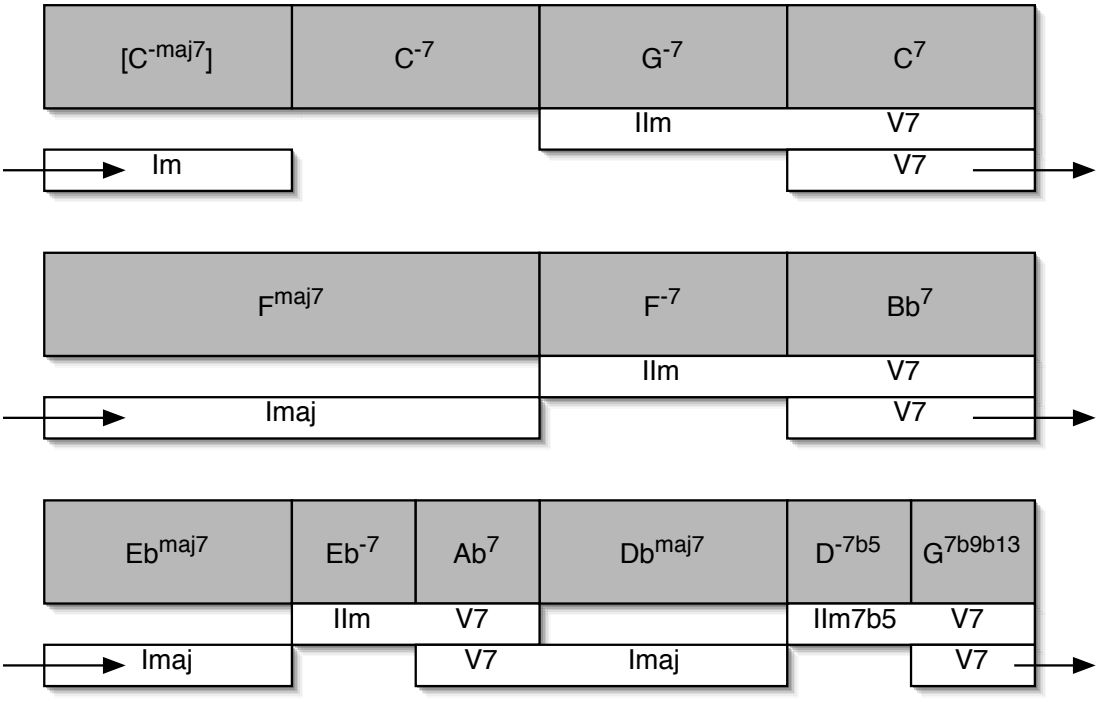



Figure 6.1: Visualized output of the first analysis engine for the tune "Solar" by Miles Davis. The grey blocks are the given chord changes, the white boxes represent the recognized patterns. The width of the boxes is proportional to their duration. Arrows indicate that the pattern wraps into the next line. Note that the last pattern wraps around to the very first chord, since the song is analyzed as a circular sequence.

### 6.2.2 Note Classification

When patterns overlap, the note suggestions of all the overlapping patterns are collected in a histogram and normalized to yield a probability distribution. Since the chord changes are patterns themselves (suggesting only chord notes), some playable notes can always be determined. Chord changes, being patterns themselves, always contribute to the final distribution. Consequently, the probability of a chord note is slightly higher than the one of a mere scale note, provided that suggestions from other patterns always include the chord notes. This condition is fulfilled in the current implementation.

### 6.2.3 Issues with the First Analysis Engine

Unfortunately, the blending strategy sometimes yields undesired results, especially over dominant chords. In these cases, almost all pitch classes are classified as equally probable. The example in Figure 6.1 shows a completely unproblematic case. All the II–V–I progressions are "regular", i.e., either complete major II–V–Is ($II^{-7}$ $V^{7}$ $I^{maj7}$) or complete minor II–V–Is ($II^{-7\flat5}V^{7(\flat9\flat13)}I^{-6/-maj7}$). The V parts in the II–V and V–I patterns suggest basically the same notes, since they both assume the same gender.

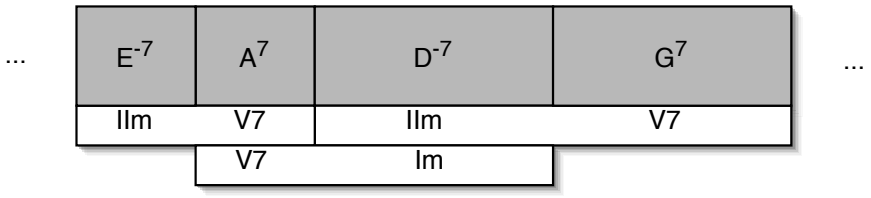



Figure 6.2: Bars 12–14 of the tune "Fly Me To The Moon", analyzed using the first analysis engine. A severe conflict can be observed at the $A^7$, created by the first IIm–V7 and the V7–Im patterns. Another, less significant conflict exists at the $D^{-7}$.

Problems arise when dealing with II–V–I progressions that switch between major and minor tonality, as is the case for $II^{-7\flat5}V^7\ I^{maj7}$ or $II^{-7}\ V^7\ I^{maj7}$. These progressions start with a II chord from the major or natural minor scale and end with a I chord from the melodic minor or major scale. For a smooth transition from one gender to the other, an altered dominant chord is preferable. However, in each of the two cases above, only one half would be recognized as minor and thus have the altered scale assigned to the dominant chord. The other half would be seen as major. Now a V chord in major context usually implies a regular 9 and 13, while in minor context the ♭9, ♯9 and ♭13 are more common. Consequently, the V parts of the recognized II–V and V–I patterns suggest completely different options in such a case, which leads to an almost uniform probability distribution allowing anything to be played.

An example of such a conflict is shown in Figure 6.2. The IIm–V7 assumes the dominant to resolve into a major I chord, but in fact the $V^7$ leads into minor. The first pattern suggests the A mixolydian scale with the 11th (D) omitted, implying a regular 9 and 13 (B and F♯), whereas the second pattern suggests the A altered scale, containing the options ♭9, ♯9, ♯11 and ♭13 (B♭, B♯, D♯, F) . The perfect first, major third and minor seventh (A, C♯ and G) are contained in both scales, as is the perfect fifth (E), which however is not directly implied by the chord itself. This constellation leads to a distribution with 10 non-zero entries out of 12 total. The chord notes have a reasonably high probability, while all the other non-zero entries have equal, low probabilities, which is undesirable. The distribution for this particular chord change is shown in Figure 6.3.

Generally, dominants are the most flexible chord types — they allow the highest number of altered options. All a dominant needs to function is the presence of the major third and the minor seventh. Virtually all other intervals (except for the perfect first, which defines the root note) can be altered. Hence, especially for dominants it is necessary to analyze a larger context, possibly even the theme melody or the user's input, in order to determine the most appropriate alterations. Just blending votes of several small-context analyses can lead to a highly ambiguous result.

## 6.3 The Optimization Approach

After the first analysis approach had shown serious deficiencies, the optimization approach from [Choi, 2004] was implemented for evaluation. The algorithm is lean and elegant, requiring no analytic rules at all. But unfortunately the choices it makes are not particularly correct or common among musicians. The following table shows the algorithm's result and the desired solution for a minor II–V–I progression:

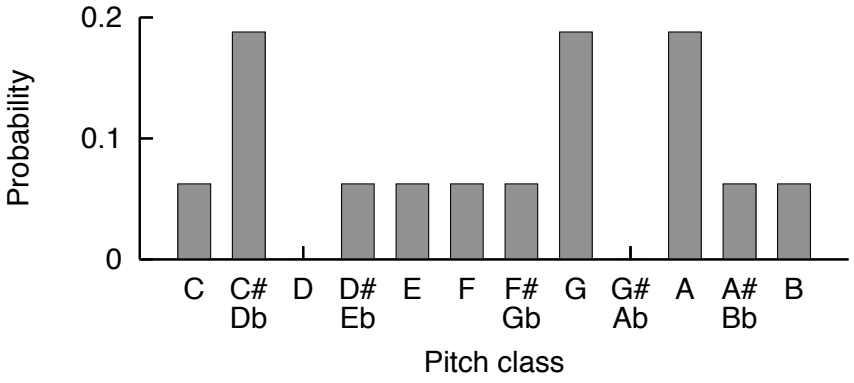



Figure 6.3: Probability distribution over the 12 pitch classes in bar 12 of "Fly Me To The Moon". The essential chord notes have very high probability, and almost all of the remaining notes have an equally low probability. This distribution results from the analysis shown in Figure 6.2.

| chord | $D^{-7\flat5}$ | $G^{7}$ | $C^{-6}$ |
|---|---|---|---|
| result | C harmonic minor | C melodic minor | C melodic minor |
| correct | E♭ major | C harmonic minor | C melodic minor |

The reason for this behavior seems to be the definition of the cost function. If cost is defined as simply the set difference of neighboring note sets, its minimization does not lead to the desired results. Harmonic functions prescribe resolution of notes into certain directions, i.e., they suggest a movement. In a dominant chord, for example, the seventh and third have a leading function: The seventh typically moves down by one semitone, becoming the third of a following major chord. The dominant's third moves up by a semitone, becoming the root note of a following tonic chord. The pure minimization approach always favors notes that avoid movement. This works perfectly for a II–V–I in major key, where the major scale of the I chord can be used for all three chords. However, it does not work in other situations, e.g., in a minor II–V–I. In such a progression, three different major scales, one for each of the chords, are the typical solution (II of natural minor, V of harmonic minor, I of harmonic minor). The pure minimization algorithm tries to avoid this movement of notes and chooses inappropriate scales. Choi states in his paper that he defined an additional rule for II–V progressions, assigning bonus scores or penalties to certain scales. However, it seems difficult to manually tweak the cost function until it yields the desired results. More musical knowledge in the form of bonuses and penalties would have to be built into the algorithm. The consequence would be a rule-based algorithm that takes the detour of a cost minimization.

## 6.4 The Second Analysis Engine

Issues with the first analysis engine lead to the development of a second version that tries to solve the analysis task more reliable and less ambiguously. The following two subsections discuss results of the harmony analysis and note classification modules in this new version of the framework.

### 6.4.1 Harmony Analysis

In our second approach, the probability-blending scheme was replaced by a non-ambiguous analysis strategy. Using rules that can utilize or overwrite previously assigned labels, each chord symbol will have no more than one Roman numeral assigned after the analysis has converged. An example analysis by this new engine is shown in Figure 6.4.

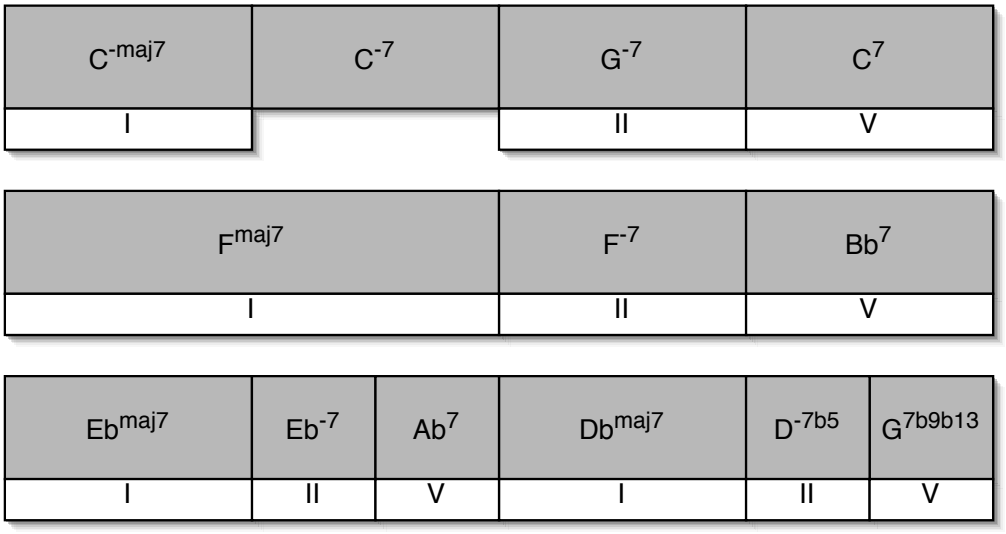



Figure 6.4: Analysis of "Solar" generated by the second analysis engine. The results are basically the same as in the first version, since this piece consists only of complete major or minor II–V–I progressions.

The problematic example from the previous section is correctly solved by the new approach. Conflicts are automatically resolved through the prioritization and interdependencies of the patterns. The new analysis of the example is shown in Figure 6.5. Due to interdependency of small patterns, a larger context is recognized: The sequence $E^{-7}$ $A^{7}$ $D^{-7}$ $G^{7}$ is recognized as the progression III–VI–II–V, which allows a more appropriate and precise note selection than the previous analysis method.

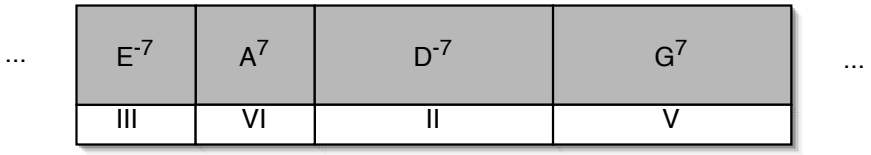



Figure 6.5: Analysis of the same passage of "Fly Me To The Moon" as in Figure 6.2. By design, there are no more ambiguities, and the analysis is more precise: Recognizing degrees such as III and VI requires a larger context to be taken into account, which is achieved by the use of interdependent patterns.

### 6.4.2 Note Classification

After the first pass of Roman numeral assignment, the roman numeral objects are used to determine actual note suggestions, potentially by looking at their neighbors.

Where no Roman numerals could be assigned during the pattern matching phase, only the chord notes receive non-zero probability. Due to the recognition of larger contexts, notes can be classified more precisely. For the V in mixed-gender II–V–I progressions, the algorithm correctly chooses the altered scale. In the case of the examples in figures 6.2 and 6.5, the altered scale is chosen because it turns out that the dominant does not actually have V function, but is rather a VI. Dominants on degree VI usually imply the altered scale. The new probability distribution for the $A^7$ chord in the example is shown in Figure 6.6. It is favorable over the distribution shown in Figure 6.3, because the number of notes it selects is exactly seven (instead of 10). Seven is also the size of a typical scale. The new distributions do not yet raise the probabilities of chord notes over the ones of other scale notes. However, this could easily be added to the implementation.

Since the first pass of the new scheme performs some sort of partitioning on the search set of possible interpretations of chords (by determining the Roman numeral and thus a part of its harmonic function), it is easier to determine a suitable scale for a chord in the second step. Due to the aforementioned flexibility of dominant chords, the decision on scales over V chords still involves the largest amount of case analysis. However, the search set for V chords is already easier to handle, since all secondary dominants (those with functions other than V) are separated from primary dominants in the first pass.

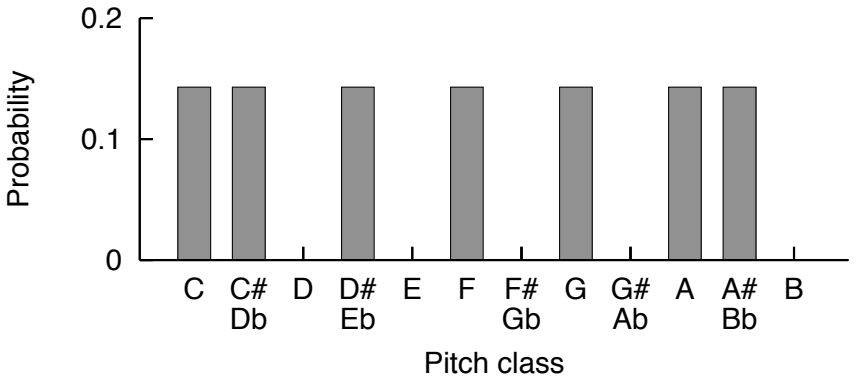



Figure 6.6: Probability distribution in bar 12 of "Fly Me To The Moon" calculated by the second analysis engine. Only 7 instead of 10 notes (as in Figure 6.3) are playable.

## 6.5 Input-Dependent Probabilities

The calculation of probabilities that depend on previously played notes is currently limited to chromatic scales. However, this simple change allows for a technique often used by pianists: two scale notes that are a whole tone apart can be connected by a half-tone step, thus including a note that does not belong to the theoretically appropriate scale. Furthermore, the chromatic scale is often played where true "inside" playing is hard, e.g., due to very fast chord changes or the need to play very fast notes. The chromatic scale can always be played, as long as it ends on some "correct" scale note.

## 6.6 Limitations

### 6.6.1 Analysis

So far, the analysis algorithm can only deal with sequences of chords that have functional interrelationships. This is a given in most older jazz standards. Modern jazz, heavily reharmonized tunes, in which harmonic functions are not obvious any more, and modal jazz tunes where chords have no function in the classical sense, are not covered by this approach. However, integrating default Roman numeral assignments for unlabeled or isolated chords (for example a default of II for every unlabeled minor chords) might help produce acceptable results in such cases.

### 6.6.2 Song Format

The song format does not yet support the encoding of tunes with explicit repetition signs, jump labels or alternative endings. However, thanks to the extensible XML file format, it should be possible to extend the system by these facilities relatively easy.

# 7. Conclusions and Future Work

## 7.1 Conclusions

With the coJIVE back-end, a system was designed and implemented that applies musical theory knowledge to chord sequences in order to suggest playable notes for improvisation. It was successfully integrated with the front-end of the coJIVE application and used in two out of three user studies. The setup and results of these studies are discussed in detail in [Buchholz, 2005].

As a byproduct, the back-end provides a set of useful data structures that can be used for analytical computations on tonal music. These portable C++ classes can serve a larger audience than, e.g., Pachet's MusES system, which is implemented in Smalltalk. Furthermore, an XML song file format was defined that can be used to describe lead sheet information in textual form. The framework routines can be used to load the lead sheet files into analyzable data structures and automatically gather melody notes from MIDI theme files referenced by the XML file.

With coJIVE, we have developed a system that improves and supports musical improvisation by players who are untrained in jazz or music in general. In our user studies, especially untrained users confirmed that what they played sounded significantly better when the support was turned on. Still, it was difficult to motivate people to overcome their inhibitions and explore. Furthermore, rhythmic ability, which is in large part a matter of training, turned out to be a prerequisite for interesting and groovy solos, since the system deliberately leaves the timing of notes unchanged. Unfortunately, some classically trained musicians and especially jazz musicians who knew which notes they wanted to hit often felt too restricted by the adjustments the system made to their playing.

One possible application for coJIVE is an interactive exhibit, where groups of people can just grab the infrared batons and the keyboard and have a little jam session. During a leisure activity like a visit to an exhibition of interactive systems, people might be more open to experimenting than in a planned and scheduled user test.

## 7.2 Future Work

This section gives some ideas on how the back-end part of coJIVE could be improved in future versions, regarding the quality of the analysis and its consequences, and regarding the ease of use for pattern implementers.

### 7.2.1 Note Duration and Velocity

The analysis could potentially gain accuracy by taking into account the duration and possibly the velocity of melody notes. These features could be used to separate important notes from mere passing tones, the latter of which do not indicate particular scales, but are just used as connections between scale notes.

### 7.2.2 Data-Driven Analysis

The theoretical information is currently hard-coded into the framework in the form of static class members or methods. The databases of useable scales, chords and most importantly the pattern descriptions could be moved to external data files, making changes to the analysis logic easier and reducing the need to recompile the framework. For the pattern descriptions, a scripting language would have to be devised that can describe the properties of patterns to be recognized, as well as the scales or notes resulting from the recognition of patterns.

### 7.2.3 Song Structure

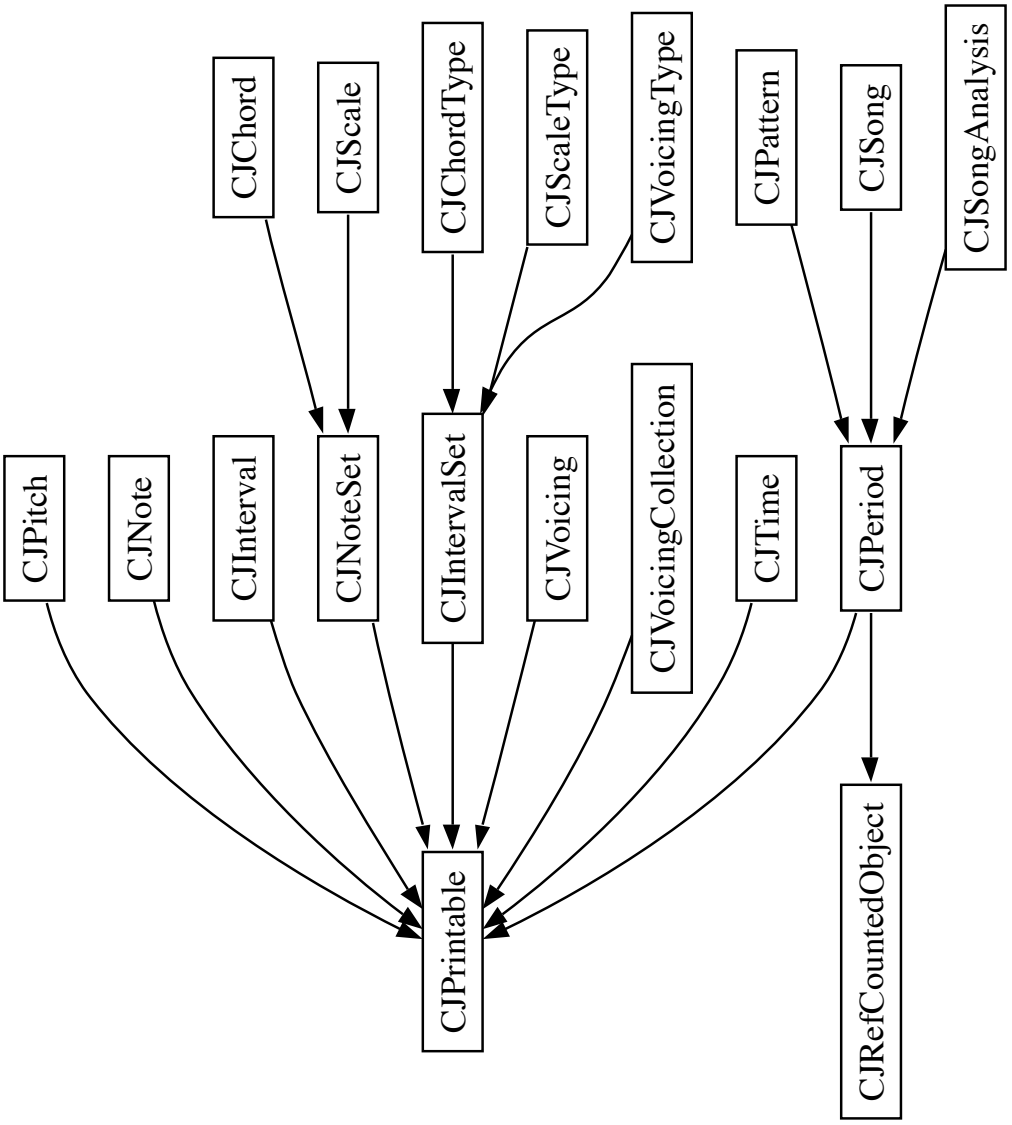
So far, the song file format supports only songs that loop indefinitely, due to missing repetition facilities. However, due to the nature of XML, the format can easily be extended by repetition and jump marks, in order to support more complex song structures, special endings, intros, outros, etc. Of course, the inner data structures and the main analysis loop would also have to be adjusted in order to accommodate such features. So far, coJIVE's front-end does not support these features either, so there a modification would also be necessary.

### 7.2.4 Improved Melody Guidance

Currently, the input-dependent modulation of probabilities is limited to the possibility of playing chromatic notes. By changing probabilities depending on previous input, the melody is guided into a certain direction. Using the history of previously played notes, which is already implemented, a database of melodic patterns could be used to offer several predefined directions for the melody at each point. Statistical models of real-world data gathered from professional musicians could allow for more sophisticated schemes that encourage (but not enforce) better melodies. Furthermore, such models can capture the different playing styles of different musicians. Blending this sort of "practical probability" with the theoretical one determined by the analysis could help in finding the right melodic patterns over chord changes. Several methods of building models were originally developed with autonomous solo generation in mind (see section 3.3), but could possibly be adapted to help support human improvisation.

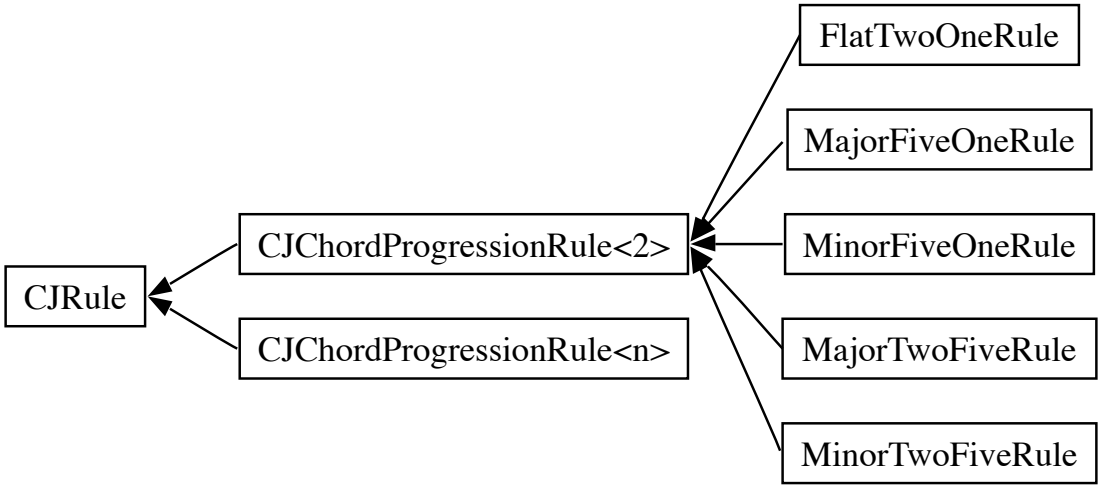
# A. Class Inheritance Diagrams

## A.1 Class Inheritance for Both Versions

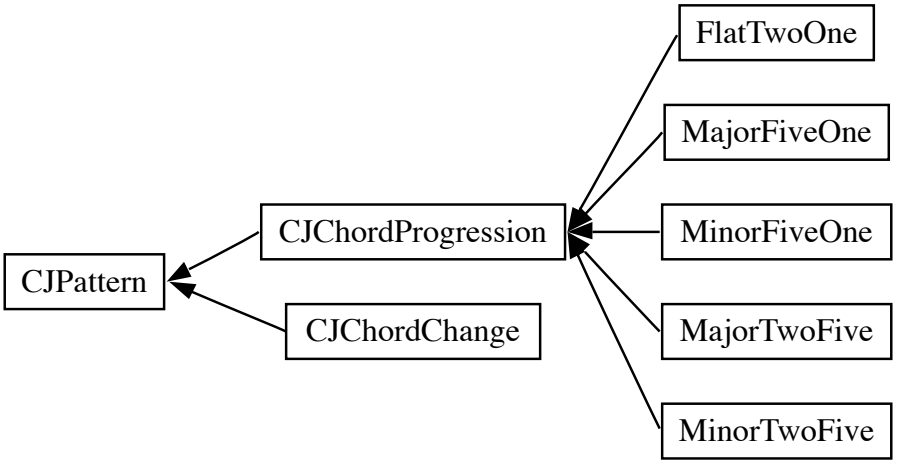


## A.2 Class Inheritance in Version 1

### A.2.1 Rules

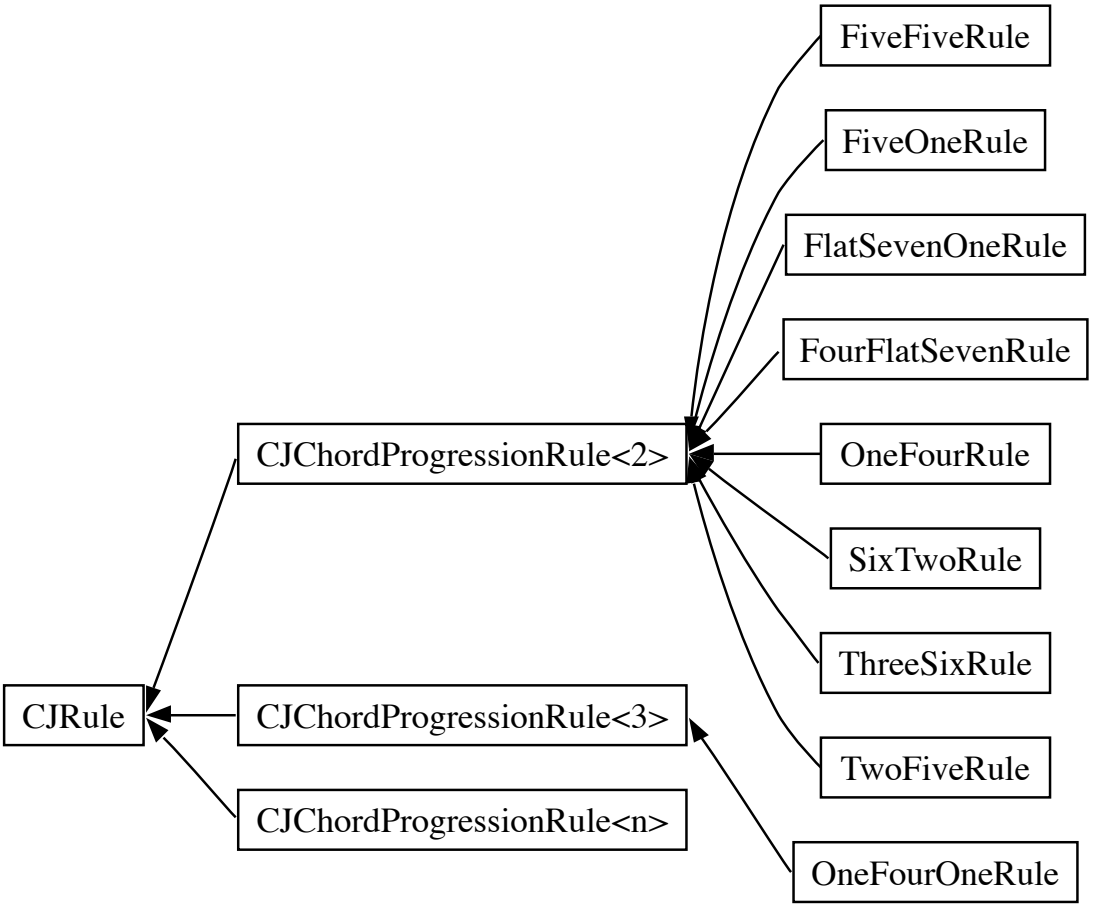



### A.2.2 Patterns

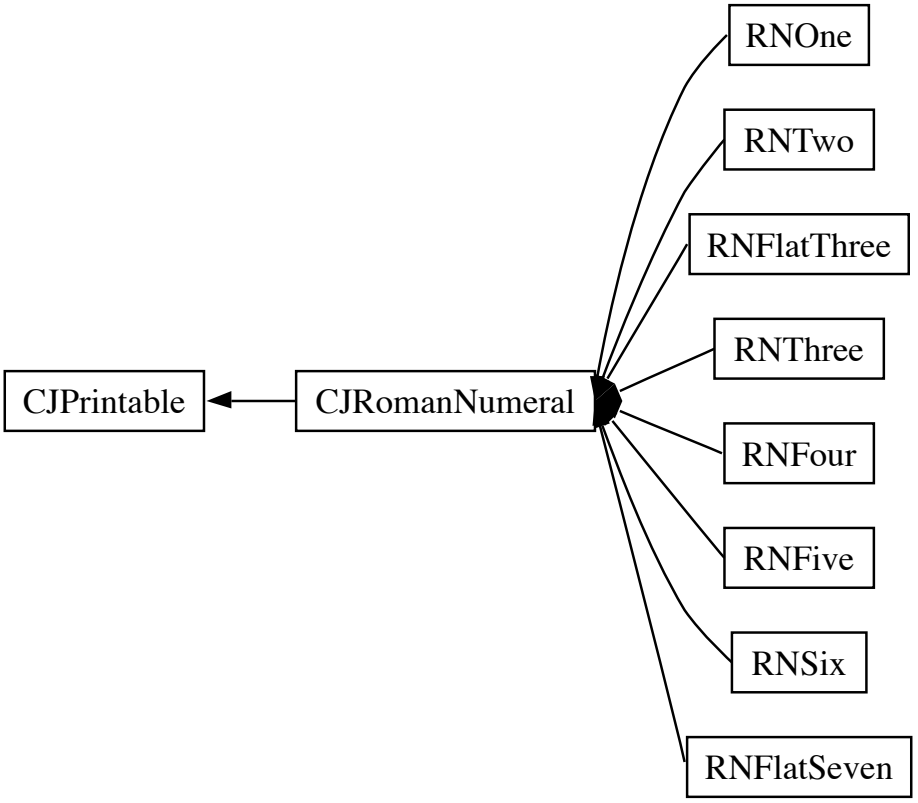


## A.3 Class Inheritance in Version 2

### A.3.1 Rules



### A.3.2 Roman Numerals



# References


Jan Buchholz. A Software System for Computer Aided Jazz Improvisation. Diploma thesis, RWTH Aachen University, Aachen, Germany, May 2005.

Don Buchla. Buchla Lightning System II. http://www.buchla.com/lightning/index.html, 1995.

Andrew Choi. Analysis of Jazz Chords as Optimization. www.sixthhappiness.ca/blog, February 2004.

John Coltrane. Giant Steps. Atlantic Records, 1960.

Judy Franklin. Multi-Phase Learning for Jazz Improvisation and Interaction. In *Proceedings of the Eighth Biennial Symposium on Arts and Technology (Perception and Interaction in the Electronic Arts)*, 2001.

Axel Jungbluth. *Jazz-Harmonielehre, Funktionsharmonik und Modalität*. B. Schott's Söhne, Mainz, Germany, 1981.

J.D. Koftinoff. libjdkmidi. A C++ MIDI Library. www.jdkoftinoff.com/main/Free_Projects/C++_MIDI_Library/, May 2004.

M-Audio. MIDIsport 2x2 USB MIDI interface. http://www.m-audio.de/mspor22.htm, 1999.

K. Nishimoto, H. Watanabe, I. Umata, K. Mase, and R. Nakatsu. A Supporting Method for Creative Music Performance - Proposal of Musical Instrument with Fixed Mapping of Note-functions. May 1998.

François Pachet. An object-oriented representation of pitch-classes, intervals, scales and chords: The basic muses. Technical report, LAFORIA-IBP-CNRS, Universite Paris VI, 1993.

François Pachet. Surprising Harmonies. *International Journal on Computing Anticipatory Systems*, 1999.

François Pachet. (1997) Computer Analysis of Jazz Chord Sequences: Is Solar a Blues? *Readings in Music and Artificial Intelligence*, February 2000.

François Pachet. Interacting with a Musical Learning System: The Continuator. In A. Smaill C. Anagnostopoulou, M. Ferrand, editor, *Music and Artificial Intelligence*, volume 2445 of *Lecture Notes in Artificial Intelligence*, pages 119–132. Springer Verlag, September 2002.

Chuck Sher, editor. *The New Real Book*. Sher Music Co., Petaluma, CA, 1988.

Belinda Thom. Bob: An interactive improvisational music companion. In *AGENTS '00: Proceedings of the Fourth International Conference on Autonomous Agents*, pages 309–316. ACM Press, 2000.

Lee Thomason. TinyXML. A simple, small, minimal C++ XML parser. www.grinninglizard.com/tinyxml, 2000–2005.

Dimitri van Heesch. doxygen. A documentation system for C++, C, Java, Objective-C, IDL (Corba and Microsoft flavors) and to some extent PHP, C#, and D. www.doxygen.org, 1997–2005.

Various Authors. KDevelop - KDE Development Environment. www.kdevelop.org, 1999–2005.

Viscount. Viscount Viva and Viva X Portable Pianos. http://www.viscountus.com/pianos/Viva.htm, 2003.

William F. Walker. A computer participant in musical improvisation. In *CHI '97: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 123–130. ACM Press, 1997.

# Glossary

**chord**

A set of simultaneously triggered notes. 10

**circle of fifths**

When starting from C and moving in steps of perfect fifths, all pitch classes are traversed. It is often used to count the number of accidentals in the signature of keys: Transposing by one fifth upwards adds a sharp sign to (or removes a flat sign from) the key signature. C major has no sharps or flats, G major has one sharp (F♯), D has two (F♯ and C♯) etc.. Moving downwards by fifths has the opposite effect: One fifth down from C is F, which has one flat sign. The full circle of fifth reads: C, G, D, A, E, B, F♯/G♭, D♭, A♭, E♭, B♭, C. 8

**dominant**

A chord that contains the major third and minor seventh intervals with respect to its root note. Dominants have the strongest leading effect of all chord types and are thus of great importance for creating local key centers. Dominants are very flexible in terms of possible alterations. 13

**enharmonic spelling**

the assignment of one of several possible note names to a pitch class. E.g. C♯ belongs to the same pitch class as D♭ - the two notes sound the same, their enharmonic spelling is different. In a similar way, this translates to intervals: Intervals that describe the same semitone distance can have different names. 5

**key**

see tonality 8

**mode**

A mode is a rotation of a scale. E.g. when the C major scale is viewed with respect to its second note D, it is called the *second mode of C major*. The greek term for this mode is *D dorian*. 8

**option**

A note that is not a chord note. When viewing chords as stacks of thirds upon a root note, the first four notes (forming the intervals 1, 3, 5 and 7 w.r.t. the root) are considered chord notes. All notes on top of that (usually 9, 11 and 13) are options which can be altered up or down. 11

**scale**

a set of notes written in ascending or descending order. Scales provide closed systems over large passages of musical pieces and define tonality. 7

**tonality**

Generally a closed harmonic system consisting of a set of notes. The usual tonalities are the major and natural minor scales. 8

**tritone**

the distance of three whole-note steps, usually in the form of an augmented fourth or a diminished fifth. 16

**voicing**

A particular arrangement of notes of a chord. 14

# Index