

*A Software System
for Computer Aided
Jazz Improvisation*

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



by
Jan Buchholz

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Horst Lichter

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

Over the last two decades, music has witnessed two major revolutions brought about by computing: In the eighties, the emerging MIDI (Musical Instruments Digital Interface) standard enabled computers to control synthesizers and other sound-generating equipment. In the nineties, computers became widely available that were fast enough to generate and play digitally sampled audio streams in real time, leading to new applications such as versatile software synthesizers and hard disk recording. But only recently mainstream processing power has reached levels that make it possible to include the complex symbolic computation necessary for computers to begin generating meaningful music in response to real-time musical human input. This has enabled computers to support musicians in their live performance on stage.

Jazz improvisation, in which musicians generate melodies instantaneously while playing with others over a given song structure, is a particularly demanding type of live performance that has always required many years of training and experience. The goal of this thesis is to change this.

coJIVE, the Collaborative Jazz Improvisation Environment presented in this thesis, enables a wide spectrum of people to begin to improvise to jazz tunes; this thesis focuses on the interactive front-end of *coJIVE*, while a companion thesis [Klein, 2005] deals with the the musical theory back-end. The complete system provides the players with the standard material—lead sheet and automatic accompaniment—required for jazz improvisation, but above all, it substitutes for their missing theoretical knowledge and experience, by carefully adjusting and augmenting their musical input to create aesthetically pleasing music. To provide a satisfying experience to novices as well as experts, *coJIVE* adjusts its support to the players' individual levels of expertise. The system also monitors the jazz session, suggesting to the players when to solo and when to accompany, and adjusting their input based on these roles.

A review of related work shows that numerous prior projects have tackled four related areas: generating improvisation using artificial intelligence, generating automated accompaniment, co-ordinating collaborative performances, and assisting human musicians live—but none of the existing systems integrated these fields into a task support as envisioned for *coJIVE*. Two usage scenarios, of a traditional and a *coJIVE*-augmented jazz session, describe this task in more detail.

The remaining work was carried out following the *DIA* (Design-Implement-Analyze) cycle: A first prototype provided basic improvisation support for a single keyboard player, and was implemented using the MAX/MSP development environment; it showed in user tests that the idea of augmented improvisation is feasible. An advanced prototype was then implemented as a Mac OS X Cocoa application with

support for multiple pieces, two players, and batons as additional input device. User testing led to developing this prototype into the final system with improvements in the support for advanced players, soloing control and other areas.

The final system was reviewed favorably by users; nevertheless, there is still much room for improvement, as outlined at the end of this thesis, to create better jazz performances using the *coJIVE* approach.

Zusammenfassung

In den letzten zwei Jahrzehnten hat die Musik durch die Informatik zwei wesentliche Neuerungen erfahren: In den achtziger Jahren ermöglichte es der entstehende MIDI-Standard (Musical Instruments Digital Interface Standard) Computern, Synthesizer und andere tonerzeugende Geräte zu steuern. In den neunziger Jahren wurden Rechner weithin verfügbar, die schnell genug waren, um digital gesampelte Audioströme in Echtzeit zu erzeugen und abzuspielen, was zu neuen Anwendungen wie flexiblen Software-Synthesizern und Harddiskrecording führte. Doch erst vor kurzem hat die übliche Rechenleistung ein Niveau erreicht, das es erlaubt, auch komplexe symbolische Berechnungen durchzuführen, die für die automatische Erzeugung von sinnvoller Musik basierend auf Benutzereingaben in Echtzeit nötig sind. Dies ermöglicht es Computern Musiker auch bei Live-Auftritten auf der Bühne zu unterstützen.

Die Jazz-Improvisation, in der Musiker Melodien aus dem Stehgreif erzeugen, während sie mit anderen Musikern über die Struktur eines Liedes spielen, ist eine besonders anspruchsvolle Form des instrumentalen Spiels. Das Ziel dieser Arbeit ist es, dies zu ändern.

coJIVE (Collaborative Jazz Improvisation Environment), das in dieser Arbeit vorgestellte Softwaresystem, ermöglicht einem breiten Publikum den Einstieg in die Welt der Jazz-Improvisation. Die vorliegende Arbeit konzentriert sich auf den interaktiven Teil des System, während eine begleitende Arbeit [Klein, 2005] den musiktheoretischen Teil beschreibt. Das vollständige System stellt für die Spieler die üblichen Materialien bereit (Lead Sheet und automatische Begleitung), die für die Improvisation nötig sind, aber darüber hinaus gleicht es ihre fehlenden Kenntnisse und Fähigkeiten durch vorsichtiges Anpassen und Anreichern ihrer Eingabe aus, um ansprechende Musik zu erzeugen. Um sowohl für Anfänger als auch Experten eine angenehme Erfahrung gewährleisten zu können, passt *coJIVE* den Grad der Unterstützung an die individuellen Fähigkeiten jedes Spielers an. Das System überwacht überdies die Session, deutet den Spielern an, wann sie improvisieren und wann sie nur begleiten sollten und passt ihre Eingabe diesen Rollen entsprechend an.

Eine Besprechung verwandter Arbeiten zeigt, dass zahlreiche frühere Projekte vier ähnliche Bereiche untersucht haben: die Generierung von Improvisationen mittels künstlicher Intelligenz, die Generierung automatischer Begleitung, die Koordination von gemeinschaftlichem instrumentalem Spiel und die musikalische Unterstützung von Musikern. Keines dieser Systeme hat diese Teilbereiche jedoch zu einer großen Aufgabe zusammengefasst, so wie es für *coJIVE* vorgesehen ist. Zwei Szenarien beschreiben eine traditionelle, sowie eine mit Hilfe von *coJIVE* durchgeführte Jazz-Session um diese Aufgabe zu verdeutlichen.

Der Rest der Arbeit wurde basierend auf *DIA*-Zyklen (Design-Implementierung-Analyse) durchgeführt : Ein erster Prototyp, entwickelt in der Max/MSP-Entwicklungsumgebung, bot grundlegende Unterstützung für die Improvisation eines einzelnen Klavierspielers. Benutzertests bestätigten, dass das Prinzip der computergestützten Improvisation sinnvoll ist. Ein verbesserter Prototyp, der Unterstützung für zwei Spieler, mehrere Lieder für die Improvisation und elektronische Schlegel als zweites Instrument bot, wurde als Cocoa-Anwendung unter Mac OS X entwickelt. Weitere Benutzertests führten zur Weiterentwicklung dieses Prototypen zum finalen System mit Verbesserungen der Unterstützung fortgeschrittener Spieler, Solo-Kontrolle und in anderen Bereichen.

Das finale System wurde von den Benutzern durchweg positiv beurteilt. Allerdings besteht noch viel Raum für Verbesserungen, wie am Ende der Arbeit beschrieben wird, um bessere Jazz Performances mit dem *coJIVE*-Ansatz kreieren zu können.

Contents

Abstract	i
Zusammenfassung	iii
Acknowledgements	v
Contents	vii
1 Introduction	1
1.1 Jazz Improvisation	2
1.2 Overview	3
2 Related Work	7
2.1 Artificial Improvisers	7
2.1.1 JIG - Jazz Improvisation Generator	8
2.1.2 GenJam - Genetic Jammer	8
2.1.3 ImprovisationBuilder - A Computer Participant in Musical Improvisation	9
2.1.4 CHIME - Multi-Phase Learning for Jazz Improvisation and Interaction	10
2.1.5 Continuator	11
2.2 Automatic Accompaniment	11
2.2.1 JOINME - An Open Advise-based Approach	12
2.2.2 An Artificially Intelligent Jazz Performer	12
2.2.3 Band-in-a-Box	13
2.3 Co-ordination of Collaborative Performances	14
2.3.1 VirJa - Virtual Jazz Session System	14
2.3.2 Improvisession - A Global Delayed Session Music	15

2.4	Musical Assistance for Human Players	16
2.4.1	WorldBeat - Musical Design Patterns	17
2.4.2	CiP - Coloring-in Piano	18
2.4.3	RhyMe - A Musical Instrument with Fixed Mapping of Note-Functions	19
2.4.4	<i>ism</i> - Improvisation Supporting System	20
2.4.5	Electronic Keyboards with Musical Assistance	21
	2.4.5.1 One-Key Chord Triggering	21
	2.4.5.2 Melody-Based Key Illumination	22
2.5	Discussion	23
3	Usage Scenarios and Design	25
3.1	A Jazz Session Scenario	25
3.2	The coJIVE Scenario	26
3.3	Design Approach	27
4	A First Prototype	31
4.1	Design	32
4.1.1	Features	33
	4.1.1.1 Confine to Scale	33
	4.1.1.2 Prohibit Avoid Notes	34
	4.1.1.3 Help Triggering Compound Structures	34
	4.1.1.4 Offer Automatic Structuring	35
	4.1.1.5 Restrict Expressive Parameters	36
	4.1.1.6 Discarded Ideas	37
4.1.2	User Levels	38
	4.1.2.1 Professional	38
	4.1.2.2 Expert	38
	4.1.2.3 Advanced	39
	4.1.2.4 Novice	39
4.1.3	Further Design Decisions	39
4.2	Implementation	40
4.2.1	Environment	40
	4.2.1.1 Hardware	40

4.2.1.2	Max/MSP	41
4.2.2	System Clock	41
4.2.3	Accompaniment	43
4.2.3.1	Bass	43
4.2.3.2	Drums	43
4.2.4	MIDI Connections	43
4.2.5	Visualisation	45
4.2.5.1	Chord Symbols	45
4.2.5.2	Feedback	45
4.2.6	User Levels	46
4.2.7	Features	47
4.2.7.1	Confine to Scale	48
4.2.7.2	Prohibit Avoid Notes	48
4.2.7.3	Help Triggering Compound Structures	48
4.3	Analysis	48
4.3.1	User Tests	49
4.3.1.1	Procedure	50
4.3.1.2	Questionnaires	50
4.3.2	Results	52
4.3.3	Discussion	52
5	Final System Development	55
5.1	Design	56
5.1.1	New Requirements	56
5.1.1.1	Collaboration	57
5.1.1.2	New Musical Interface	58
5.1.1.3	Songs	58
5.1.2	Supported User Profiles	58
5.1.3	Graphical User Interface	60
5.1.3.1	The Lead Sheet	60
5.1.3.2	Song Controls	61
5.1.3.3	Player Fields	61
5.1.3.4	Configuration Controls	62

5.1.4	Features	64
5.1.4.1	Processing a Note	65
5.1.4.2	Processing a Gesture	66
5.1.4.3	Processing Velocity	66
5.1.5	Architectural Design	67
5.1.5.1	System Foundation	67
5.1.5.2	Centralised Tasks	68
5.1.5.3	MIDI Encapsulation	70
5.1.5.4	Processing Input Data	71
5.1.5.5	Data Representation	71
5.2	Implementation	72
5.2.1	Environment	73
5.2.1.1	The Back-end	73
5.2.1.2	Hardware	73
5.2.1.3	Cocoa	73
5.2.1.4	CoreMIDI	74
5.2.1.5	SimpleSynth	75
5.2.2	Implementation Principles	75
5.2.3	System Foundation	76
5.2.4	The Lead Sheet and Data Classes	78
5.2.5	Singletons	79
5.2.5.1	PlaybackEngine	79
5.2.5.2	RecordingEngine	80
5.2.5.3	BackendDummy	81
5.2.6	The MIDI Classes	81
5.2.7	Input Processing	83
5.2.7.1	Note Processing	83
5.2.7.2	Gesture Processing	85
5.2.7.3	Velocity Processing	85
5.3	Analysis	86
5.3.1	User Tests	86
5.3.1.1	Procedure	86

5.3.1.2	Questionnaires	87
5.3.2	Results	88
5.3.3	Discussion	90
5.4	Changes in the Third Cycle	90
5.4.1	Design	91
5.4.1.1	Performance Control	91
5.4.1.2	User Support	92
5.4.1.3	User Interface	93
5.4.1.4	Altering the Architecture	95
5.4.2	Implementation	96
5.4.2.1	Environment	96
5.4.2.2	The New and Revised Classes	97
5.4.2.3	Note and Gesture Processing	98
5.4.2.4	System Performance	100
5.4.3	Analysis	101
5.4.3.1	User Tests	101
5.4.3.2	Results	103
6	Conclusion	107
6.1	Adjusted Musical Assistance	107
6.2	Collaboration	108
6.3	Simulating a Jazz Session	108
7	Future Work	109
7.1	Musical Assistance	109
7.1.1	Melody Guidance	109
7.1.2	Rhythmic Guidance	109
7.1.3	New Chord Voicings	110
7.1.4	Responding to Classical Musicians	110
7.2	Collaboration and Session Arrangement	110
7.2.1	Solo Arrangement	110
7.3	Special Assistance for Jazz Musicians	111
7.3.1	The Retry Button	111
7.3.2	Recording Statistics	112

A	Max/MSP Objects	113
A.1	Basic Objects	113
A.2	Math Objects	114
A.3	Control and Selection Objects	114
A.4	MIDI Objects	115
B	MIDI Basics	117
B.1	Basic Principles	117
B.2	Message Types	118
B.2.1	NoteOn	118
B.2.2	NoteOff	118
B.2.3	ProgramChange	118
B.2.4	ControlChange	118
B.3	Note Values	118
C	UML Diagrams	121
C.1	The Second Prototype	122
C.2	The Final System	125
D	<i>coJIVE</i> in Use	133
D.1	Setting Up the System	133
D.2	The Session	135
	References	141
	Glossary	145
	Index	149

1. Introduction

Watching musicians performing on stage can be a very entertaining and fascinating experience; their ability to control one or more instruments, their knowledge of the songs they play and the way they interact attract the audience's admiration. Most people should know the desire to take part in such performances and receive their "15 minutes of fame".

In the world of jazz, improvisation is commonly used to create surprising, yet appealing performances. Jazz musicians create such performances with what seem to be spontaneously developed melodies coming from a seemingly endless supply of new ideas. For untrained people interested in trying out improvisation for themselves, this sort of creativity seems impossible to master, thus keeping them from even trying. This poses a central question that was the origin of this work:

- How can inexperienced people be supported in order for them to create appealing performances thus providing them with a satisfying experience?

The computers seems to be the perfect tool to represent and apply the knowledge and experience jazz musicians use for their improvisations. Thus, it may be able to facilitate creative performances for people with different levels of expertise by partially taking control over the user's input and correcting it as necessary.

This thesis describes the development of a computer system aimed at enabling a wide variety of people to improvise by substituting for the necessary abilities. The system is called *coJIVE* — Collaborative Jazz Improvisation Environment — and is meant to allow several users to engage in a simplified jazz session, providing each of them with an appropriate amount of support to even out the skills they lack, at the same time co-ordinating the course of events in the session.

Initially, the topic of jazz improvisation must be examined to obtain an overview of the procedure itself. What exactly is the "secret" behind this sophisticated form of instantaneous creativity? An answer to that question requires a closer look at what happens in jazz sessions.

1.1 Jazz Improvisation

Unlike usual musical performances, in which performers replay pre-composed melodies, the art of improvisation can be described as the ability to create melodies from the spot. Of course, this comprises more than “just” coming up with an idea that can be turned into a melody.

To know what can be played and how, a jazz musician needs to analyze the structure of a given song; this song is usually described on a lead sheet, a sheet of paper that contains the necessary pieces of information: the name of the song, its tempo, metre (e.g., $\frac{4}{4}$ or $\frac{3}{4}$), a notation of the main theme and its chord structure. This structure is given in the form of a progression of chords, all textually annotated by a chord symbol. Figure 1.1 depicts an exemplary lead sheet published by Sher [1988]. The chord progression enables an analysis of the scales and their modes (dorian, phrygian, lydian, etc.) underlying each chord; a thorough description of this analysis was compiled by Klein [2005], the companion thesis of this work. Jazz musicians usually perform this analysis while playing. With the scale suitable for the current position in the song and the notes contained in the current chord, the harmonic context can be derived. This context determines what notes would be perceived to be melodic (consonant) and which notes would sound disharmonious (dissonant), and acts as a basis for the improvising musician (improviser) to create his solo.

For a fluid performance, the improviser cannot solely depend on spontaneous ideas; she rather keeps in mind a collection of short phrases and melodies from prior training and performances. By adapting a sequence of those phrases to fit the current harmonic context, a short solo can be created instantaneously requiring the generation of a melody “from scratch”. A more thorough description of the concepts and methods used in jazz improvisation was compiled by Sabatella [1992-2000].

Similar to other musical presentations, jazz improvisations often feature a whole band in what is usually called a session. Borchers [2001] describes this set-up as part of his *pattern language for interactive music exhibits*: “Use a combination of one or several rhythm section instruments, such as drums, bass, piano, or guitar, plus one or several solo instruments...” [Borchers, 2001, p.84]. This configuration, of course, requires a lot of co-ordination to assure a balanced performance, since most of the musicians involved in a performance would like to play at least one solo. While one player is improvising, the others need to provide an accompaniment that serves as a rhythmic and harmonic basis for the solo. At the same time, this accompaniment needs to leave enough room for the improvisation to be in the centre of the performance. This concept is described by Borchers [2001] under the term *Solo & Comping*: “Let players take turns in soloing, including melody instruments as well as accompanying instruments. The remaining players step back musically, and some of them accompany the soloist.” [Borchers, 2001, p.86]. The following chapter will more thoroughly describe such collaborative sessions.

The abilities necessary for improvising and accompanying improvisation in jazz sessions obviously take several years (sometimes decades) of rehearsing to develop. Three main skills can be identified:

1. **Precise command of the instrument:** To translate an idea into a melody, a musician needs to know how to create certain notes or chords by heart. An

experienced piano player, for example, can find the right keys for the notes on a score sheet as he reads them. This kind of expertise can, of course, only be achieved through constant practice over several years.

2. **Thorough knowledge in musical theory:** The kind of knowledge necessary for the analysis mentioned in the last section even surpasses what is commonly taught in classic musical education. Beside keys and the most common scales, jazz musicians need to know the various modes of each scale, and while in classical music chords are made up of three notes, the chords in jazz contain at least four notes.
3. **Experience in collaborative improvisations:** Knowledge of the most basic gestures and how to accompany a fellow musician in his improvisation is needed for a performance to sound balanced. This, of course, encompasses suitable behaviour towards the other musicians based on the roles of the different participant

1.2 Overview

As even absolute novices in the field of musical performances are meant to use *coJIVE* without an introduction, the system needs to provide mechanisms to bypass the necessity for the three skills described in the last section. It is developed in two major parts. Each part has distinctive tasks to perform, and interact to meet the overall goals:

1. The Back-end

The part of *coJIVE* representing musical knowledge is concerned with the analysis of chord progressions that is usually conducted by the jazz musicians themselves. Based on this analysis it is supposed to decide how probable the different notes are in the context of each chord. Furthermore, it should know several chord voicings for each chord symbol that should be made available to client applications. The companion work to the thesis at hand, the diploma thesis of Jonathan Klein [2005], describes the development of the back-end.

2. The Front-end

The front-end of the system will be developed to interact with the players, and it applies the knowledge supplied by the back-end to alter the players' input. It should also create audible and visual information necessary for the performance. Small groups of players will be supported to co-operate in collaborative sessions. The work at hand focuses on this part of *coJIVE* and its development.

To attract jazz experts as well as total novices, two different MIDI devices that represent different classes of interfaces were selected: a common MIDI keyboard represents the class of note-based interfaces that require precise control (e.g., knowing what key to press). It resembles the interface of pianos that are often used in jazz session, and thus rather addresses users familiar with playing such an instrument. And yet, one of *coJIVE*'s tasks is to help inexperienced players to create interesting performances using the keyboard, by correcting and possibly enriching their input.

Admittedly, system support to their keyboard input needs to use caution, as a player can otherwise quickly lose the feeling of being in control.

A pair of infrared batons created by Don Buchla [1995] represents the class of *fuzzy* interfaces for less experienced users. They resemble sticks to use on drums or xylophones, and can be used to play notes accordingly: Hitting gestures in mid-air trigger notes, changing the note to be played by moving the baton to the left for lower notes, or right for higher notes. The batons are considered to be a fuzzy interface because experienced players can only estimate which “key” they will hit. Due to this fact, the player’s performance can be more intensely modified by the system, as changes in the relation of gestures to notes are less likely to be noticed by the user.

The main part of this thesis consists of seven different chapters:

Chapter 2, Related Work, describes several systems that pursue similar goals as *coJIVE*, pointing out similarities and major differences in the underlying approaches.

Chapter 3, Usage Scenarios and Design, presents two scenarios describing a traditional jazz session and a session using *coJIVE*; they clarify the goals and tasks of the system and describe how the system should work in general. Additionally, the main approach taken for the design of the system is explained.

Chapters 4 and 5 focus on the prototypes created during development. They each explain the approaches taken by the development, and are structured according to the *DIA (Design — Implementation — Analysis) cycle*: each design phase, in which ideas are collected and processed, is followed by an implementation phase that contains the transition of the design into a working system, and the analysis phase, in which the resulting system is tested and validated.

Chapter 6, Conclusion, is concerned with the final test results, which are summarised and interpreted. The initial goals of this research are compared to the findings in the analysis phases and the revision of expectations throughout the research process is documented.

Chapter 7, Future Work, concludes this work by stating open questions and possible future courses of action. It presents a list of enhancements and additions to the system that should be undertaken to expand the system’s abilities without compromising the basic ideas beneath this work.

Four appendices contain a description of the Max/MSP objects deployed in the first prototype, the MIDI standard, detailed UML diagrams for the second and third prototype, as well as a walkthrough describing the interaction with the final version of the system. Terms from computer science and jazz theory are then explained in a glossary. An index concludes the thesis.

355

There Will Never Be Another You

Med. Swing

Music by Harry Warren
Lyric by Mack Gordon

A $E^b_{MA}7$ $D_{MI}7(b5)$ G^7

There will be man - y oth - er nights like this, _____ And

$C_{MI}7$ (F^7) $B^b_{MI}9$ E^b_{13}

I'll be stand - ing here with some - one new, _____ There

$A^b_{MA}7$ $D^b9(\#11)$ $E^b_{MA}7$ $C_{MI}7$

will be oth - er songs to sing, An - oth - er fall, an - oth - er spring, But

F^9 $F_{MI}7$ B^b7

there will nev - er be an - oth - er you. _____ There

B $E^b_{MA}7$ $D_{MI}7(b5)$ G^7

will be oth - er lips that I may kiss, _____ But

$C_{MI}7$ (F^7) $B^b_{MI}9$ E^b_{13}

they won't thrill me like yours used to do, _____ Yes,

$A^b_{MA}7$ $D^b9(\#11)$ $E^b_{MA}7$ $(F^{13} A_{MI}7(b5) D^7)$

I may dream a mil - lion dreams but how can they come true if

E^b6 $A^b9(\#11)$ $G_{MI}7$ C^7 $F_{MI}7$ B^b_{13} E^b6 (B^b7)

there will nev - er, ev - er, be an - oth - er you?

©1942, 1987 Twentieth Century Music Corp. © Renewed 1970 Twentieth Century Music Corp. All Rights Throughout The World Controlled by Morley Music Co. International Copyright Secured All Rights Reserved Used By Permission

Figure 1.1: A lead sheet describing the song “There will never be another you” [Sher, 1988].

2. Related Work

Although using computers in public performances seems quite common today, their creative influence inside these performances is negligibly small and often still subject to research. Usually the machines play back pre-recorded parts to accompany human players or provide capabilities for sound generation to be controlled by human musicians. The lack of creative control over the performance is mostly due to the fact that “computer-generated music” has a bad reputation even today. It is commonly believed that computers allow the creation of music at the push of a button, and that this music is of poor quality creative-wise. Accordingly, musicians fear to lose their credibility if computers take over some of the creative parts in their performances.

Still the possibilities computers entail have already attracted several groups of researchers to develop computer-based systems that facilitate, support, or create performances. This chapter will present a selection of systems devoted to provide help for one or more human players to create and co-ordinate performances, similar to *coJIVE*. A special emphasis is put on those aspects that *coJIVE* approaches differently. But the first section of this chapter takes a look at systems dedicated to create improvisations on their own using knowledge from the field of artificial intelligence, as these systems have to follow similar paradigms as *coJIVE*. The chapter is concluded by a discussion of the systems’ most important innovations and their shortcomings in the context of jazz improvisation.

2.1 Artificial Improvisers

The thought of creating a computer program that is able to create improvised performances, although frightening for musicians who fear to get “replaced”, is a most interesting prospect for researchers in the field of artificial intelligence; yet, representing the knowledge necessary within such a program and having a computer apply this knowledge while playing on its own is a task that has only been completed partially so far. Indeed, various efforts in this field of research have produced considerable results, some of which are presented in this section.

Since these systems are concerned with creating performances and not with the support of a human performer, stating the differences between them and *coJIVE* would

not be beneficial to the work at hand. Instead, the aspects of jazz performances these systems go into that are also interesting for this work are highlighted.

2.1.1 JIG - Jazz Improvisation Generator

Grachten [2001] presents JIG, a system generating “formulaic” improvisations based on two models for jazz improvisation. The first model describes such performances to be composed of so-called *motifs* (or *formulae*), short melodic patterns drawn from the personal repertoire of the improvising musician. The second model focuses on the spontaneous creation of note sequences that some jazz improvisers use to connect motifs. This simplification of the process of improvisation is very interesting as it is easy to utilise while no vital aspect is neglected.

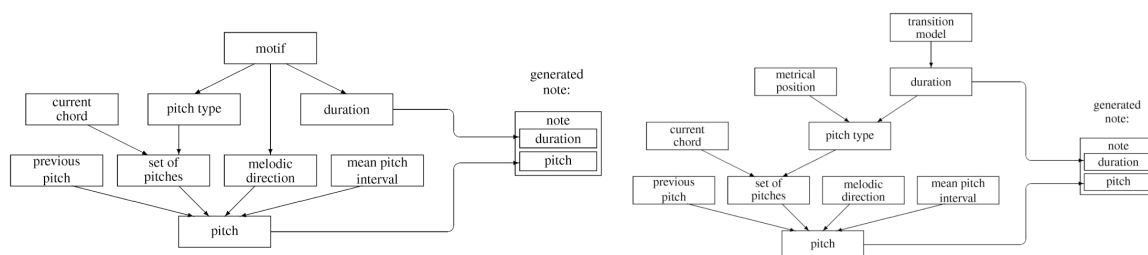


Figure 2.1: JIG’s note generation in motif(left) and *melodying* phase(right) [Grachten, 2001].

Accordingly, JIG works with two alternating note generation phases, the *motif* phase and the *melodying* phase [*sic*]. The motif phase extracts short melodic patterns from the current song’s main theme, adjusts the pitches to the scale and transforms the motif by reversion, contraction/expansion of notes, etc.; that way new melodies are created that somewhat relate to the original melody of the song. The melodying phase deploys probabilistic models to generate note duration and pitch as well as the duration of the pause to the next note. The performance is preceded by an analysis of the chord sequence that selects the scales for the chords. These are chosen by a simple algorithm that calculates the scale matching most of the chords in the sequence; when a new scale must be calculated, the one with the least differences to the previous scale is chosen. With the calculated scale a probabilistic selection based on random values can be made by the active generative phase.

2.1.2 GenJam - Genetic Jammer

Genetic algorithms are heuristics used to solve optimisation problems by applying techniques derived from natural evolution. Starting with a population consisting of so-called chromosomes (representations of solutions for the problem), such algorithms use mutation, inheritance, selection and recombination to generate new generation derived from this population; a fitness function is used to evaluate and select some of the chromosomes, from which the next generation is then created.

With GenJam, Biles [1994] presented a system that uses such a genetic algorithm to generate jazz performances. The system uses chromosomes containing a sequence of pairs of note value and duration as representation of a potential performance.

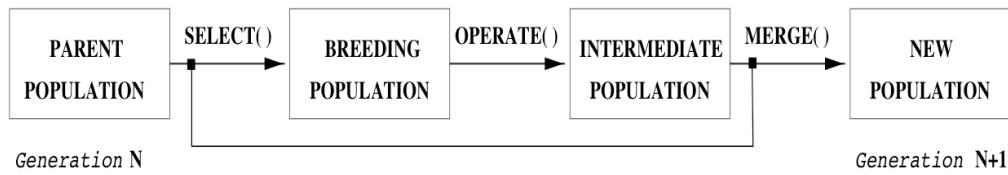


Figure 2.2: The genetic algorithm deployed in GenJam [Biles, 1994].

The note value is not stored as the absolute pitch (as in MIDI) but rather as the note's degree in the current scale. The initial population is generated; the durations of each note in a chromosome add up to the duration of the whole song, and the note values are determined at random. Once this population is created, the genetic process is started using a fitness function that repeatedly evaluates the newly created chromosomes by a set of characteristics: intervals between notes (preventing too large intervals from occurring), long notes (only fit if the note really is harmonic, for example, a *chord note*), rhythmic position of the note (at downbeat or half-bar), pattern matching (the system looks for similar melodic patterns in a chromosome), etc.

2.1.3 ImprovisationBuilder - A Computer Participant in Musical Improvisation

Focusing on the structure of performances by small groups of jazz musicians, Walker [1997] developed ImprovisationBuilder, an artificial improviser that tries to fit in with this kind of groups. This ImprovisationBuilder draws on the conversational character of jazz improvisation; it recognizes the roles of the various participants and reacts accordingly by accompanying or soloing.

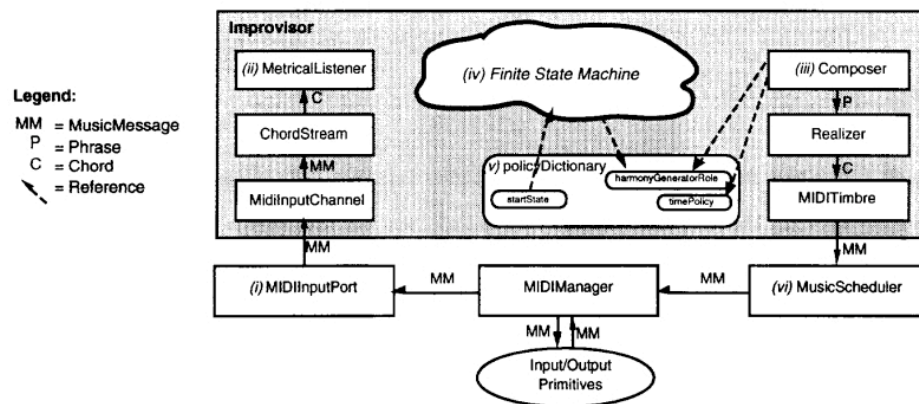


Figure 2.3: Structure of an Improviser component in ImprovisationBuilder [Walker, 1997].

An object-oriented approach was used to model the different sub-tasks performed by jazz musicians as classes that are aligned in a linear chain to create a performance. This chain is always headed by a *listener* analyzing the performances of the other

participants, and recognizing if they are accompanying or improvising, based on the range of pitches used and the amount of notes played; depending on the result, the state of the system is altered in a finite state machine. A *composer* is used to create performances deploying methods similar to JIG: short melodic patterns are adapted to the current scale (previously determined by the system) and connected by randomly generated sequences of notes. The creation chain in *ImprovisationBuilder* is concluded by an object of the so-called *timbre* class that acts a sound generator.

The research on *ImprovisationBuilder* was supported by deploying prototypes: two pianist composers, who also helped out in the development of the system, constantly used the prototypes in their home studios. That way, a permanent stream of feedback was created, which was clearly beneficial for the system, not only in terms of usability.

2.1.4 CHIME - Multi-Phase Learning for Jazz Improvisation and Interaction

The Computer Human Interacting Musical Entity (CHIME), developed by Franklin [2001], deploys a *recurrent neural network* to learn how to improvise and interact with human players. This kind of neural network uses feedback loops to learn from its own results. The system was trained to play three different melodies and trade fours with human players. The process of learning was split into two phases: in the

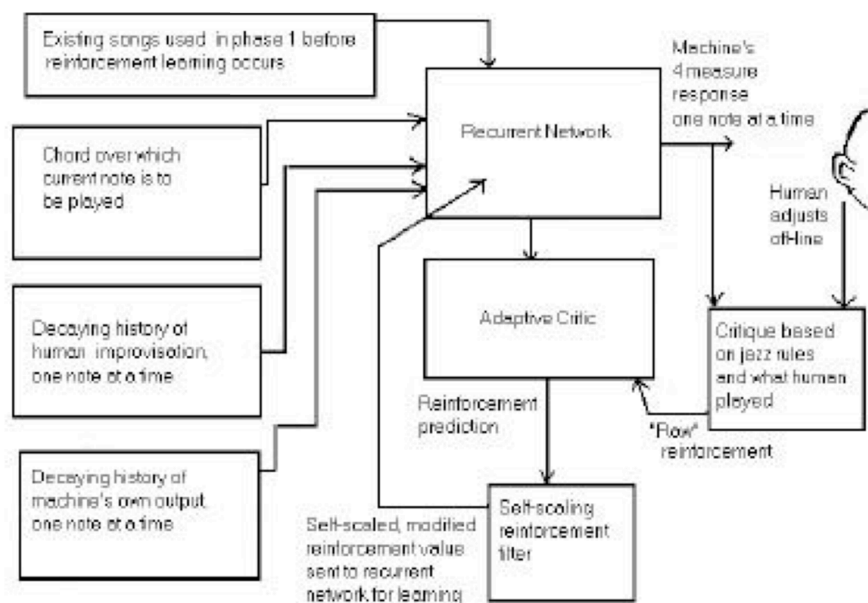


Figure 2.4: Structure of CHIME [Franklin, 2001].

first phase, the neural network was trained with the three melodies using supervised learning. This learning method presumes that the correct output results (i.e., the melody) are known, to feed the errors made by the network back into it. The second phase was conducted using reinforcement learning, in which the network's output is evaluated by a set of rules that rate the suitability of single notes and the results handed back to the network. Additionally, the system was fed with the performance

of a human player in that phase to derive ideas for its own melodies. The resulting system is able to interactively play with other musicians, using their input to create new melodies.

2.1.5 Continuator

The Continuator [Pachet, 2002] is a system built to create and play “continuations” of performances by human players, i.e., it analyses the performance, and tries to find an appropriate performance as an answer similar to jazz musicians. To achieve this, it looks for recurring patterns in the improvisation of the preceding soloist.

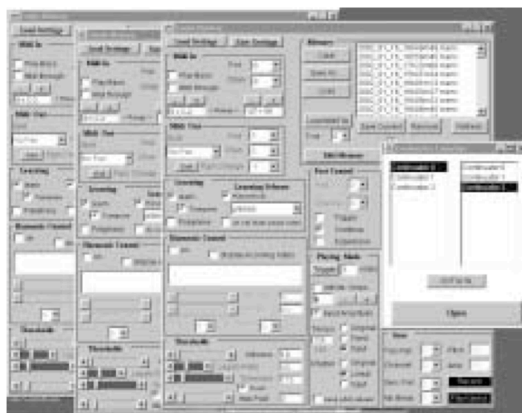


Figure 2.5: Several instances of the Continuator running on one computer [Pachet, 2002].

While listening to a human player performing, the system builds a prefix tree with the incoming data, weighting subsequences of notes by the number of their occurrence. For this purpose, not the pure data but simplified information obtained from reduction functions (e.g., using pitch regions instead of absolute pitches) is used. The generation takes place by determining a note item, then using the previously created prefix tree to find an appropriate continuation (e.g., a pitch region); this is repeated for a note item taken from that continuation, and so on. This calculation is executed by a hierarchical Markov model that is further augmented by using external information: data like the last eight notes played by the human player (used for harmonising the performance instead of relying on information on the chord structure) or the overall velocity of the rest of the band (to adjust the Continuator’s velocity to) is used to alter the system’s behaviour. This adjustment is made using a fitness function that compares the computed continuation to the external information. This “attachment” to the external input can be adjusted by the user.

2.2 Automatic Accompaniment

Accompaniment, as mentioned above, is necessary for an improvised solo to come across to the audience as it was intended: it provides a rhythmic and a harmonic foundation that a melody can relate to. This kind of support seems easy to be provided by computers, as recreating pre-recorded parts is a standard task for current machines. But there are also other approaches to automatically creating accompaniment taken by current research.

2.2.1 JOINME - An Open Advise-based Approach

Similar to improvising, playing an accompaniment demands knowledge and experience from musicians that is collected over years. In order for a system to use this kind of knowledge, the information must be represented inside the system, while at the same time being open to extensions (just like a musician keeps on learning). Therefore, the knowledge needs to be separated from the actual calculation. JOINME, developed by Balaban and Irish [1996], is based on this idea of using not algorithms to compute an accompaniment but so-called *advices* [sic].

JOINME uses a music knowledge base to analyse chord sequences and producing the accompaniment. This knowledge base contains object-oriented representations of notes, intervals, chords, and scales, as well as rhythmic points and rhythmic structures; rhythmic structures are patterns of rhythmic points, which in turn are points in time and represent notes. There are two kinds of advice that can be used to determine the value of a rhythmic point: a Domain Advice determine the notes possible at a rhythmic point based on the harmonic context at that time; a Composition Advice restricts the possible notes, for example, based on the calculated value of the previous rhythmic point and decide on note to play. The set of advices in JOINME can easily be extended by subclassing the classes representing the desired kind of advice.

JOINME offers a flexible approach to creating accompaniments for performances of human players. For the goals of the work at hand, this flexibility is not necessary as the focus lies on the human performance rather than on the provided accompaniment; yet, this way of artificially creating musical structures could be interesting to be incorporated into future versions of *coJIVE* to make the system output more varied.

2.2.2 An Artificially Intelligent Jazz Performer

An automaton-based approach to build a system that tries to mimic the behaviour of a jazz bass player is presented by Ganascia et al. [1999]. The automaton used to create the performance models a mood, short term memory, and a musical memory inside its state, which all affect the performance. To facilitate interaction with its environment, a simple interaction model is used: the audience as well as a drummer and a pianist are also understood as automata, whose output serves as input for the bass automaton. A similar approach has already been presented by Dannenberg and Bloch [1985], focusing on the input provided by a keyboard. Roger B. Dannenberg and Bloch [1985] has been involved in numerous projects concerned with interactive system over the last two decades; in recent years he turned to topics like query by humming and sound synthesis.

The artificial bass player is divided into two automata: the *Executor* actually triggers notes through a MIDI connection based on their start time and duration. This data, along with the note values, is calculated by the *Improviser* that uses the input from the other automata — including a grid automaton transmitting information on the chord structure of the current song — and fragments taken from real jazz bass performances to decide on the notes to play. These fragments are stored in the musical memory as so-called PACTs (potential actions) and can be combined to create new PACTs and thus, new performances.

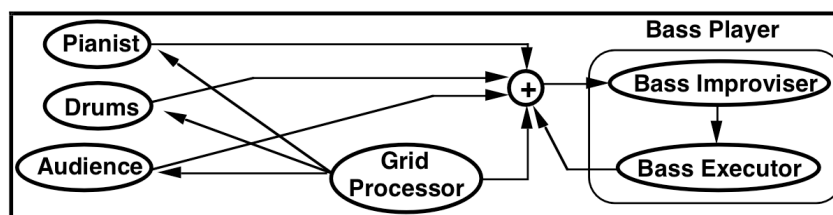


Figure 2.6: The Simplified Interaction Model used for the Jazz Bass Automaton [Ganascia et al., 1999].

The most interesting aspect of this system is the modelling of the external aspects of a sessions, especially the audience, as automata, which is used to influence the bass player’s performance. Yet, the approach of the system would not necessarily be suitable for *coJIVE* since the variable output of the bass could distract inexperienced users if the performance becomes too unusual.

2.2.3 Band-in-a-Box

The only commercially available system in this category is Band-in-a-Box, designed and developed by Dr. Peter Gannon [1989], a program designer and musician for PG Music, Inc. This program offers various features for musicians to obtain a suitable accompaniment for their rehearsal, and can be used to record, compose, and depict new songs and performances in scores.

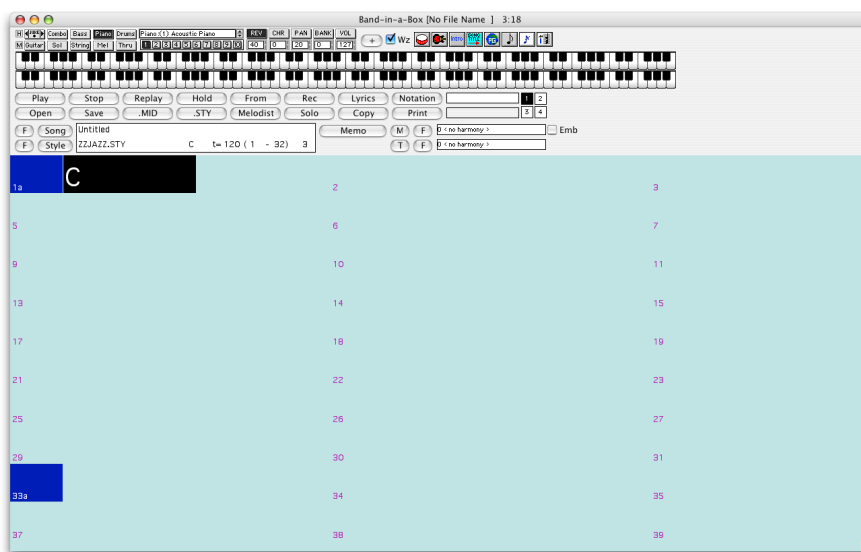


Figure 2.7: The main screen of Band-in-a-Box [Gannon, 1989].

Band-in-a-Box presents the user with a grid resembling a lead sheet, in which the chord structure can easily be altered using a *Chord Builder* tool. Furthermore, several interfaces using different input metaphors are supplied: a screen depicting all elements in a MIDI drum set (kick drum, snare drum, etc.) can be used to input and

record drum performances. To offer a suitable display for guitar-like input, another screen depicting the neck of a guitar can be used. If a faster approach towards an appropriate accompaniment is necessary, a great amount of files containing data on styles (such as swing or blues) or complete songs can be chosen from. Tools using techniques from the field of artificial intelligence can be triggered to create professional solos or whole songs.

Band-in-a-Box has several big advantages: it offers a wide variety of different features, and is thus a very powerful tool when it comes to creating and supporting performances. Because of this, the program has been able to spread among musicians during recent years, especially in the jazz sector. This fact may, of course, be due to the lead-sheet-based structuring of the program — which is very interesting in the context of this work —, and its accounting for various jazz styles and elements unique to jazz (e.g., chord symbols). Yet, the system has one major disadvantage: its usability. Since the lead sheet takes up a lot of the main screen, the other functions are accessible only through elements clustered in a very small area. Additionally, usage is often inconsistent: while the drum- and guitar-screen use appropriate metaphors for depiction, the *Chord Builder* looks like a complex configuration dialogue. In the context of *coJIVE*, this complex control would not be suitable, since users should only need to focus on their performance when using the system.

2.3 Co-ordination of Collaborative Performances

Jazz sessions with several musicians require some co-ordination. The sequence of soloists and the length of the solos can be determined before the performance, but it is also possible (and customary) for the jazz musicians to manage these aspects during the performance. This procedure, of course, demands some knowledge on how to communicate these changes while playing.

As most computers today are connected to a network, systems aimed at the co-ordination of musical performances often use this connectivity to co-ordinate remote musicians, each in front of his own computer. *coJIVE*, on the other, hand focuses on the support and co-ordination of collocated multiple players based on their levels of skill, and is intended to be a stand-alone system.

2.3.1 VirJa - Virtual Jazz Session System

Most computer programs dealing with performing with a human player need a leading participant to follow (or they even need to be the leader themselves). In jazz performance, musicians often co-ordinate their collaboration by establishing eye contact and using gestures while playing, without having a defined leader. With this technique, the structure of a performance can be changed, shortened or extended spontaneously by all of the participating musicians. To enable players to use gestures in a remote session connected by a computer network, Goto et al. [1996] presented VirJa, the virtual jazz session system.

VirJa provides all participants in a session with a video image of every other participant, similar to video conferencing systems. Two basic gestures are defined to structure a performance while playing: if the current soloist leans to one side, she

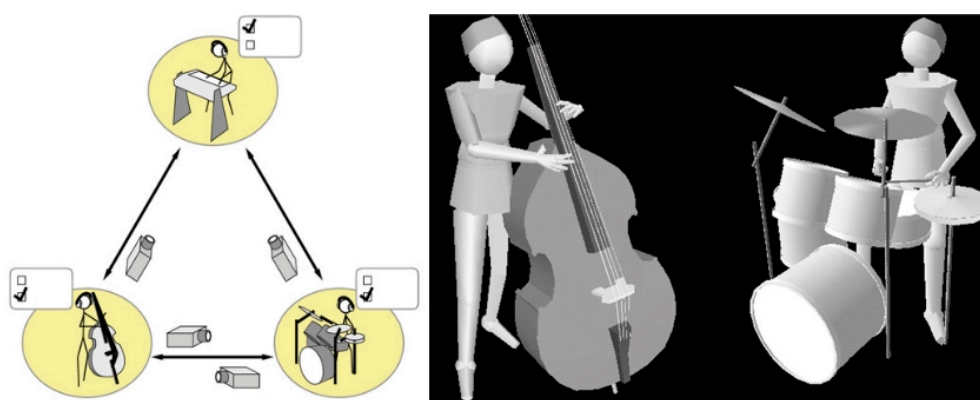


Figure 2.8: The structure of a trio in VirJa(left) and virtual players(right) [Goto et al., 1996].

wants the player to her left or right to take over the soloing role; with pointing at his head, the soloist orders the rest of the group to return to the *head* of the song and replay its main theme. The system tries to recognize the gestures, transmitting a signal to the other participants on recognition. This mechanism is, of course, necessary to integrate artificial players controlled by a computer, which was done rather successfully in VirJa. For human players to relate and react to those artificial participants, those systems are represented by characters created with computer graphics. A special communication protocol called RMCP is used to transmit MIDI data, gesture information, beat information, and animation information. An experiment, in which a jazz pianist and computer-controlled bass and drum players performed a jazz standard, showed that the representation of the computer players and the lack of a need for a leader led to an interesting expressive performance due to the interaction of the participants.

Clearly, VirJa breaks with one paradigm most computer music systems have in common: well-defined leadership. Yet, the coordination of the performance relies entirely on the participants' use of the predefined gestures; although the set of expected gestures is fairly limited compared to live jazz sessions, the users need to know them and how a jazz session is structured in general. For inexperienced players it could be difficult to meet that task as they are usually occupied with playing on their own. Even classical musicians capable of controlling the piano, for example, still have to be briefed on the topic of jazz performances.

2.3.2 Improvisation - A Global Delayed Session Music

In a group of musicians, the performance of each participant needs to be synchronized with the others; usually the players stick to the rhythm provided by the drums. In remote sessions over computer networks this poses a problem: even as network connections are becoming faster and more stable, transmissions sent through a network still need some time to travel, experienced as latency at the receiving end. So a general synchronisation of performances on different computers is almost impossible. Nagashima et al. [1992] developed a concept called *global delayed session*

music and created Improvisession, a system using the concept to allow remote improvisational music sessions over networks. A similar principle was described by Welzl [1992].

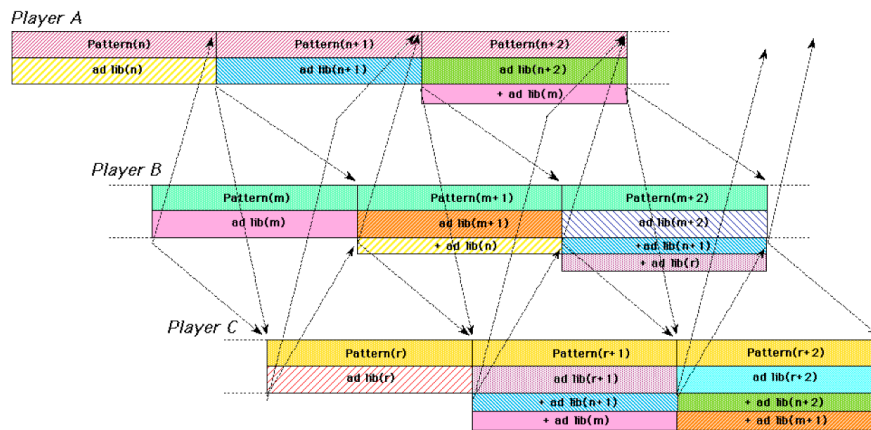


Figure 2.9: The concept of global delayed session music in Improvisession [Nagashima et al., 1992].

Global delayed session music (GDSM) is based on looping over short song structures. Every participant in the session is expected to be at a different position in this loop at a specific moment in time. The performance of a participant is recorded over the length of a loop and is then sent to all other participants. On receiving such a performance, the system delays the playback of that particular performance until the end of the loop is reached and another pass starts. The concept was developed during the development of a series of Improvisession prototypes: Improvisession-I focused on the broadcasting of performances; Improvisession-II integrated a first implementation of GDSM using the Max/MSP environment, also providing accompaniment; New interfaces like touch sensors were integrated in the third prototype to cover new ways to improvise apart from playing traditional interfaces like keyboards. A final application prototype was built in co-operation with Yamaha; this prototype added a visualisation of pending performances that have just been received. With this depiction, players can see and interpret what will be played in the next loop and may act accordingly (e.g., stopping their performance with a note close to the first note in the upcoming part to establish an audible connection).

Although providing mechanisms to create improvisational performances over networks without the need for actual time synchronisation, GDSM and the various Improvisession systems neglect some aspects important to the traditional approach of such sessions: In jazz improvisation, the musicians communicate by soloing; a soloist often uses phrases the last soloist has played and alters them to form an “answer”. As GDSM delays solos, an answer to them is also delayed and is therefore taken out of its temporal context.

2.4 Musical Assistance for Human Players

The abilities most people do not exhibit that are however necessary for improvisation are fully based on experience and training: controlling an instrument and the

knowledge of musical theory are skills that cannot be learned overnight. Systems that deal with human players providing musical input need to handle this problem by carefully revoking the users' control over the performance without depriving them of the feeling to be in control.

2.4.1 WorldBeat - Musical Design Patterns

The *Ars Electronica Center* (AEC) in Linz, Austria is a modern museum demonstrating on five floors how information technology will change various aspects of everyday life in the years to come. The KnowledgeNet floor, the second of these five floors, is dedicated to learning and working with computers. One of the exhibits created for this floor is called WorldBeat developed by Borchers [1997]. It was on permanent public display in the AEC from 1996 to 2000. The exhibit demonstrates how computers can be used to facilitate learning in the field of music, and is therefore divided into different modules with certain features: directly playing different instruments (new or classical), conducting a virtual orchestra, finding a song's name by humming its melody, or training one's ability to recognize instruments from their sounds.



Figure 2.10: A Visitor of the AEC in Linz using WorldBeat [Borchers, 1997].

The most interesting WorldBeat module for this thesis, however, is called Musical Design Patterns. It allows users to improvise to generic blues songs without playing “wrong” notes. The system provides a basic accompaniment (drums and bass) that the player can adjust to his liking in tempo, groove, and the overall blues scheme used. The user can choose among various traditional instruments, which are then controlled like a xylophone with infrared batons that are used throughout the system for operating and navigating the exhibit. The computer recognises hitting gestures, and maps the horizontal position to a note within the current blues scale, thus ensuring that the result will be in harmony with the accompaniment.

The feedback from visitors of the AEC showed that Musical Design Patterns was the most popular module in WorldBeat; creating music without the need for prior education or a long introduction seems to be a desirable experience for most people. But this orientation towards the museum's visitors resulted in restraints that are less appropriate in the context of this work: the restriction to a generic blues song based on one of three blues schemes (simple, enhanced, and complex) allows a pre-defined mapping of notes to the position of the batons, which is too inflexible in case more

specific songs should be included into the system in the future. The system also only aims at users with no experience and knowledge in the field of music; the batons present an interface feasible for everyone as they only provide rough control over the notes. For experienced musicians, the input mode for the batons could be set to include all notes. A keyboard could also be connected to the exhibit, an option that was rarely used.

2.4.2 CiP - Coloring-in Piano

A major aspect in all kinds of music is expression; besides playing the correct sequence of notes, it is of grave importance to a performance to find the right dynamic, volume, rhythm, etc. to reflect the mood of a piece. CiP, the Coloring-in Piano [Oshima et al., 2002a], was developed to offer pupils (and teachers unfamiliar with a given piece) of classical music the possibility to concentrate on those aspects of their performance. Its name seems to refer to the concept of “coloring-in” a picture (i.e. filling out pre-defined areas with colour), since it also simplifies the task of performing a given melody by pre-defining the sequence of notes.

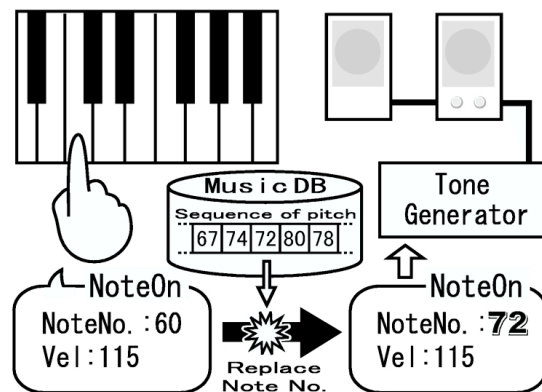


Figure 2.11: The structure of Coloring-in Piano [Oshima et al., 2002a].

CiP consists of a YAMAHA Grand Piano with a pedal controller, an SGI workstation for processing the input, and a tone generator. The piano delivers the input to the workstation via MIDI, which itself is connected to the tone generator, again via MIDI. Before playing a certain piece, its sequence of notes is fed into a database connected to the CiP system. During the performance, each note the player plays is replaced by the next note in this sequence, while the original velocities and pedal messages are preserved; thus the player’s expressive intention remains unchanged. The evaluations following the development of CiP compared the velocities and gaps between notes used in a conventional performance to those in the conducted trial performances; the test subjects performed a piece with only one finger, then again with two fingers, and finally with all five fingers of a hand. The results showed that the observed parameters in the trial performances drew closer to their conventional equivalents as more fingers were used.

Although CiP frees the performer from finding the right notes and thus facilitating expressive intentions, the system is limited to the note sequence of the given piece;

therefore, any creative input by the user concerning the melody is lost. Also, the rigid note sequence means that the player actually needs to know the exact melody to pace it appropriately to match the harmony. This task is still very hard to meet for an inexperienced musician. If a user wants to add her own ideas to the performance, a more flexible system is needed.

2.4.3 RhyMe - A Musical Instrument with Fixed Mapping of Note-Functions

In a scale, the notes contained have a *harmonic function*, named I to VII, I being the scale's root note as mentioned in its name (e.g., C in C dorian, F in F mixolydian). Jazz musicians often remember phrases as sequences of these functions, so a phrase can be easily fit to a scale. Aimed at Be-Bop style jazz improvisation, RhyMe focuses on harmonic functions and changes the interface (a keyboard) according to each new scale. The system was developed by Nishimoto et al. [1998] as part of MusiKalScope, a system which tries to facilitate and visualise jazz performances.

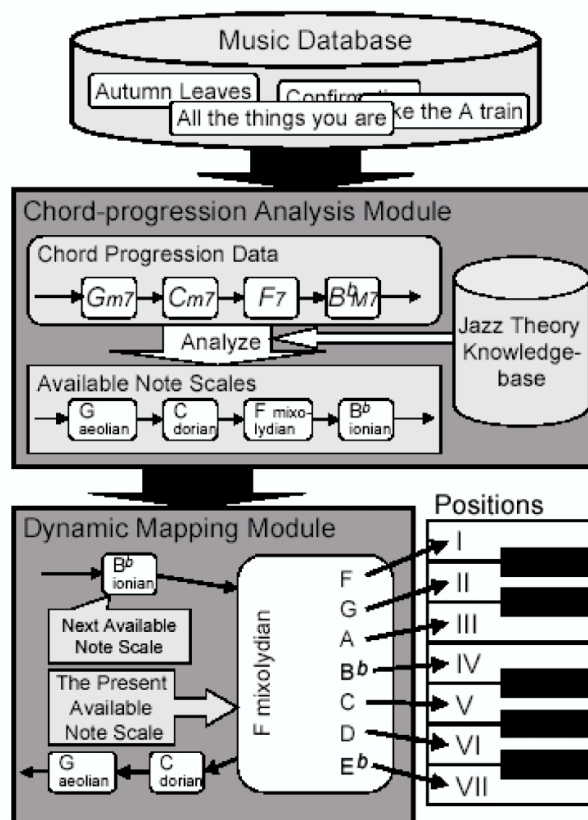


Figure 2.12: The structure of RhyMe [Nishimoto et al., 1998].

A database belonging to RhyMe contains the chord sequences for several well-known jazz pieces; if one is selected, an offline analysis calculates the scale for each chord in the progression. This analysis is conducted with the help of a knowledge-base developed according to the Berklee Theory (developed in the College for Music by the same name [Jungbluth, 1981]). In a performance, the scale belonging to the current chord is selected and the notes are mapped to the white keys of an octave based on their functions: the root note (I) is mapped to the C-key, the II is mapped

to the *D*-key, and so on; so a specific function can always be found on the same key. To research the potentials and benefits of RhyMe, a series of experiments was performed, which had the test subject play over the same piece both with and without the support of RhyMe followed by an inquiry. The results clearly stated that using RhyMe made the subjects feel that their performance was more appropriate considering the style of music and its performance.

With the remapping of notes to the keys on the keyboard, RhyMe leaves the choice of notes to the player, thus being open to creative input. Yet, this mapping not only has benefits: on the one hand, all notes in a scale are handled equally, although they do not necessarily have the same importance in a performance; some notes in the scale can sound dissonant (so-called *avoid notes*), while using the current *chord notes* guarantees a harmonic performance. On the other hand, the constantly reoccurring remapping of the keys can cause confusion for experienced users as well as novices to the instrument; pressing the same key at two different points in time eventually leads to the sounding of two very different notes, while experienced users rely on their knowledge of where to find a specific note if needed. Thus, a precise control over what is played can only be achieved after training a certain song using the system for some time.

2.4.4 *ism* - Improvisation Supporting System

The systems presented thus far in this section all focus on the pieces played, calculating the result of mappings in advance. As a system intended for non-improvising players who can play an instrument, *ism* [Ishida et al., 2004] processes the input provided by the user instead of the structure of the piece to find the most appropriate output.

As part of *ism*, a large database containing 208 melodies from standard jazz pieces is used to evaluate the player's choice of notes. Therefore, the probability of the sequence of notes played before plus the current note is calculated by an N-gram model; beside the note number, this model also takes into account the kind of note (chord tone, etc.), the interval to the last note, and its position on a beat. If the probability surpasses a certain threshold, which can be used as a parameter to adjust the level of support provided by the system, the last note does not need to be corrected. Otherwise, *ism* calculates the note maximising the probability when added to the sequence of previous notes. To validate this system, a group of non-improvising musicians was asked to perform an improvisation. This improvisation was recorded and corrected by hand and the system. The two resulting melodies were compared and the appropriateness of the system's correction was calculated using the number of corrected notes (by the system) and the number of all correction-requiring notes (determined by hand). *ism* turned out to be very precise in correcting notes; it left notes outside the current scale unchanged if those were appropriately deployed.

Using the threshold, *ism* can be easily adjusted for a wide variety of people. Yet, the correction is only aimed at people who can play an instrument already, while novice users are not supported appropriately. Mistakes like hitting two neighbouring keys with one finger are not corrected by the system. There is also no function to enrich the input by a novice user, for example, by mapping chords to single keys.

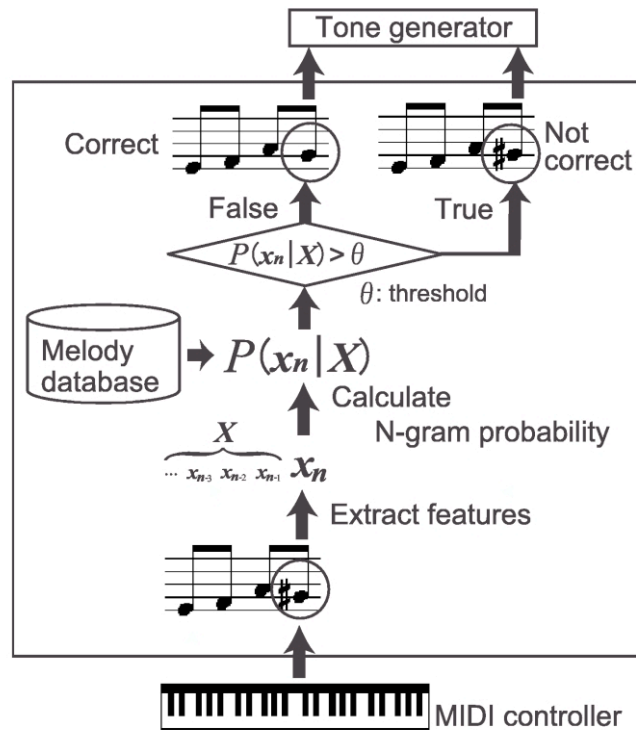


Figure 2.13: The structure of *ism* [Ishida et al., 2004].

2.4.5 Electronic Keyboards with Musical Assistance

Beside being subject to research projects like the ones mentioned above, musical assistance is a topic that has found its way into commercial products. Especially electronic instrument can offer various features due to the rapid evolution of small electronic circuits in the last few decades; most electronic keyboards today include tiny computer systems that can be programmed to offer accompaniment, record performances, etc. This part of the section examines two features aimed at training and supporting inexperienced players: offering the possibility to trigger chords with one finger, and the illumination of keys to visualise the course of a predetermined melody. Both will be accompanied by a contemporary product providing the feature.

2.4.5.1 One-Key Chord Triggering

If pianists, keyboard players and organists would only play melodies with one hand, their performance would sound thin after a while. To support such melodies and to broaden their performance, musicians often accompany these melodies themselves by playing suitable chord voicings with the other hand. In classical music these voicings are usually annotated in the sheet music; in jazz improvisation, of course, the player needs to decide himself on which voicing to use. Furthermore, he has to know how to play the voicing, i.e., what keys to press. To abolish these needs, some electronic keyboards, such as the Roland [2004] Juno D, offer the possibility to map certain voicings to certain keys.

Using these voicings, even untrained players can generate broad performances without knowing the underlying chords. Which chords are used depends on the context



Figure 2.14: The Roland Juno D synthesizer [Roland, 2004].

to ensure that what is played fits the current harmonic context; the Juno D synthesizer, for example, offers different chord maps that map different chords — rather than different versions of one chord — to the keys. Naturally, these mappings limit the choice of voicings suitable at any point of time in a performance, and they require the user to learn which chord is mapped to which key. For the work at hand, this behaviour is rather inappropriate as it forces the user to adopt his playing to the new functions of the keys. Still, a more context-bound version of this technique could enable particularly novices to play more complex performances.

2.4.5.2 Melody-Based Key Illumination

Learning to play a specific song usually involves repeatedly playing the melody of the song to memorise the sequence of notes and an appropriate expression for each one (e.g., velocity, duration). To support this process, some electronic keyboards supply a visual display of the notes: they illuminate the keys corresponding to the notes in the correct sequence to inform the user what key to press next. One such is the LK-90TV from Casio, Inc. [2004]: it offers a database with 100 songs to rehearse, and can also use MIDI files to illuminate keys.



Figure 2.15: The Casio LK-90TV with illuminated keys.

On the one hand, this function facilitates the learning process: songs are mostly recited by reading the notation, which itself is an ability that needs to be learned and trained for quite some time. Accordingly, the illumination of the appropriate keys shifts the focus from the score — an abstract representation of the song — to the used interface, the keyboard, and the actual activities of the player, pressing keys. On the other hand, enforcing a certain sequence of notes is not suitable in terms of jazz improvisation: improvising rather involves developing one's own melodic ideas

than a precise re-enactment of pre-composed songs. While this technique still leaves an extensive degree of freedom — the player can choose to use other keys than those illuminated, in contrast to the proceeding of CiP — it offers no real aid in terms of spontaneously creating melodies.

2.5 Discussion

The research and development of the systems presented in this chapter includes some remarkable efforts made to create or support performances with the help of computers; most of the systems were able to produce interesting results in the later stages of their emergence, proving those assumptions correct that have initially led to their development. Yet, they all specialised on certain aspects, each of them only focusing on a few of the sub-tasks that make up a performance. For the work on *coJIVE*, some of the approaches taken by these systems can be adopted to fit the task of supporting users in jazz improvisation; but only an appropriate combination of several features can bring the system closer to allowing satisfying results for preferably all potential users.

At first glance, the artificial improvisers seem to have a totally different perspective and approach on the topic of jazz improvisation: They try to mimic the behaviour of jazz musicians, while *coJIVE* focuses on assisting human players. Yet, the rules followed by both approaches are quite similar as both try to implement skills necessary for improvising. The guidelines compiled in musical theory must be followed when creating or correcting a performance. Collaborating with other players implies an additional set of directives to allow fluent and balanced performance. Accordingly, the behaviour of those systems can act as examples for the mechanisms that need to be developed to meet the goals of this work.

The systems generating an accompaniment mainly focus on the recreation of the behavioural patterns of a backing band; this accompaniment is a necessary element in a jazz session. As the work at hand is more concerned with its users' performances, creating the accompaniment is secondary. However, the accompaniment systems could be interesting for a future combination with a working version of *coJIVE* to offer a more varied backing by the system. Therefore, an open and extendable approach as offered by JOINME could be of interest because it also aims at a complete representation of musical knowledge.

While *coJIVE* focuses on allowing collocated collaborative sessions on one computer — offering conditions similar to real jazz sessions —, the systems mainly concerned with this sort of performance connect distributed players through computer networks. Clearly, this distribution of users would also be an obvious step to take for the work at hand; VirJa already offers mechanisms to bring remote players closer by a kind of video conferencing system, although Improvisession pointed out the delay of a network to be a fatal problem. It handled this problem by introducing a new paradigm for interaction in music, the global delayed session music, which however diverges heavily from the common way of interaction in jazz session. So to stick to this original style of interaction, *coJIVE* keeps with the concept of one location. Still, similar rules of behaviour are the basis for all systems concerned with co-operation in music. So a look at the implementation of these rules in the other systems can be beneficial to this work.

The systems presented in this chapter focusing on musical assistance offer the most interesting solutions in the context of this work; they also focus on substituting for a user's missing knowledge and experience. Yet, not all of the solutions can be adopted for the cause of improvising in jazz: CiP, for example, enforces the strict re-enactment of a particular pre-recorded melody, which represents a contrast to the activity of improvisation. RhyMe and *ism* have taken more appropriate approaches by analysing the underlying song and redirecting or carefully correcting the user's input based on their implementation of musical knowledge. With these mechanisms, the user's play is steered towards more suitable choices without dictating him what to choose. Additionally, WorldBeat has shown how this can be successfully deployed using batons, offering a wide degree of freedom to all users. It could be adopted to fit the users' needs, and could also be used by two users (some users just used the batons together, each one playing with one of them). Commercially available electronic keyboards - like the Roland Juno D - have already pointed out ways to break the traditional relation of one action resulting in one sound, which may be pursued further to offer broader approaches to performing, especially for inexperienced users.

3. Usage Scenarios and Design

Initially, the scope of application for the system needed to be defined: to determine what mechanisms and device the system would have to deploy, its tasks needed to be clearly established first. Therefore, a set of goals that the system would have to meet were formulated:

- Allow several users to perform in a simplified jazz session over jazz tunes, and support them by providing computer-generated accompaniment.
- Provide each player with a suitable amount of support adapted to her skills and needs.
- Provide a structure for the session, and ensure that this structure is preserved.

Since the course of events in a normal jazz session as well as its particularities in contrast to other music performances yet need to be defined, this chapter will present two scenarios. The first of these scenarios describes the activities in a real jazz session, focusing on the aspects that *coJIVE* should account for. The second scenario then describes how the use of the final system will reflect the same (or similar) aspects.

Finally, a section describes the overall design approach taken for developing the system based on the aspects observed in the scenarios. For that purpose, a design principle for new musical instruments that was adopted in the design process of this work is presented. Although *coJIVE* was planned to be created as a whole system rather than a single instrument, the ideas presented in this principle can also be applied to it, as the basic assumptions are very similar.

3.1 A Jazz Session Scenario

Jonathan, Thorsten, Marius and Philipp meet in the Jakobshof, a local pub in Aachen, to take part in a jazz jam session. Marius has played drums for several years, Thorsten is a long-time bass player, and both have been active in playing jazz for many years. Jonathan has just finished his studies in jazz piano at a well known

jazz school in Essen, while Philipp is a self-educated trumpet player with more than five years of practice.

As soon as it is their turn to perform, the four musicians agree upon a song to play. They also decide who has the first solo, and that solos should not be longer than two times the length of the song structure (the so-called *chorus*). Everyone takes out the lead sheet of the song and places it clearly visible on a stand in front of him. Marius counts eight beats aloud to mediate the rhythm to the rest of the group and then starts the performance; he plays a simple swing rhythm as the chosen song is a swing, while Thorsten contributes a walking bass playing one note on each beat. Jonathan plays a rhythmic pattern using different voicings of the current chord noted in the song structure on the lead sheet, and Philipp recites the song's main melody with his trumpet.

Once the end of the chorus is reached, Thorsten and Marius start over with their performance, while Jonathan and Philipp commence to improvise. The first solo belongs to Jonathan as decided in advance; he starts by playing a slight variation of the first part of the main melody that Philipp had just played. He continues by using different short melodic patterns that he had learnt in earlier sessions. In this solo, Philipp only plays few notes to support Jonathan's performance without disturbing the soloist. After improvising for two choruses, Jonathan switches his role in the session with Philipp. The trumpet player now improvises, picking up and slightly changing some of the melodic patterns he heard in Jonathan's solo. Shortly before the end of the chorus, Philipp seems satisfied with his performance and gestures at the others that he is finished.

Marius uses this opportunity to hold up four fingers thereby triggering a playing mode called *trading fours*. Jonathan plays another solo that this time only lasts for four bars, and then Marius performs a drum solo for the subsequent four bars. The group repeats this alternation (with Jonathan and Philipp taking turns in improvising before Marius' drum solos) of melodic improvisation and drum solo for the next two choruses. During these shorter turns the players more frequently mimic or reuse melodic material from their respective predecessor.

After that, the musicians repeat their initial behaviour; Marius and Thorsten go back to playing a swing drum pattern or a walking bass respectively, while Philipp again recites the song's main melody on his trumpet. Again, Jonathan uses chord voicings to create a harmonic base and support the melody. At the end of the chorus the group brings the session to a termination and leaves the stage for someone else to take over.

3.2 The coJIVE Scenario

Eric and Eugen happened to be in the Jakobshof when Marius, Thorsten, Jonathan and Philipp performed together, which made them eager to try out jazz improvisation themselves. However, they both do not exhibit the necessary skills since they have only little experience in performing music. So they enlist the help of *coJIVE*.

Before performing with the system, they have to agree upon who uses what instrument. As Eric has some experience in playing the piano, he will play the keyboard while Eugen obtains a set of infrared batons that are played in a xylophone-like

manner. They furthermore decide on a song to play, and select the appropriate file to open from a file dialogue, which is then displayed in form of a chord progression on the screen using a cursor to mark the current position in the song. One after the other, they adjust settings corresponding to their skills to enable an estimation of their abilities in the system.

Eric starts the session by pressing a start button. The system plays a bass-line, a drum track, and the melody of the song. Eric and Eugen both carefully test their instruments in accompanying the melody. As the end of the chord progression approaches, the system notifies Eric that his first solo will start soon; it displays a countdown of the beats remaining until his solo starts, and illuminates a spotlight pointing at Eric to further indicate his new role as soloist. In his solo, Eric plays carefully to create an initial melody. The system corrects some of the notes he presses that it found to be inappropriate by triggering one of its neighbouring notes. Also, it occasionally fills in additional notes with some of the notes played by Eric to form a certain chord to be played. Both aspects are also displayed on the screen on a virtual keyboard, so Eric can see his actions and the reaction of the system.

At the end of the next chorus, Eugen is notified about the fact that his solo will begin soon (again with a countdown of the remaining beats and the illumination of another spotlight). While Eric was improvising, Eugen's performance occasionally grew more muted when Eugen trigger many loud notes. With this mechanism, the system tries to prevent Eugen from unintentionally disturbing Eric's solo in accordance to the behaviour an accompanying musician would exhibit in a real session to abstain from drowning the improvisation with her own performance. With the roles now changing, Eugen starts to play louder, and the system makes it harder for Eric to disturb Eugen's solo. Eugen's performance is also displayed by the system. The system only offers suitable notes to be triggered by the batons, rather than correcting his input. Although he cannot specifically play certain notes, he still can determine the general shape of the melody by moving the batons to the left or right.

In the sessions, the system switches the roles of Eric and Eugen yet a few times each time leaving between one and three choruses for the soloist to create a melody in. After each having played a few solos, they both decide to end the session by pressing the stop button of the system.

3.3 Design Approach

These scenarios give an outline of what the final version of *coJIVE* should be able to perform was given without limiting the choice of solutions for the different tasks. A rough outline of the interaction with the system and its output is shown in figure 3.1. This depiction is used throughout this work in different versions to display the stages of the development. To find a suitable approach for the development of the system, the research conducted for similar systems was considered in more detail.

The previous chapter has shown that by using computers, new instruments can be created that entail new capabilities: with traditional instruments a musician has to adapt to the instrument by rehearsing, which takes a lot of time. Computer-based instruments, on the other hand, can be adjusted to the needs of the users. Therefore, such systems focus on how their usage can be facilitated, rather than requiring thorough experience for interesting sounding results.

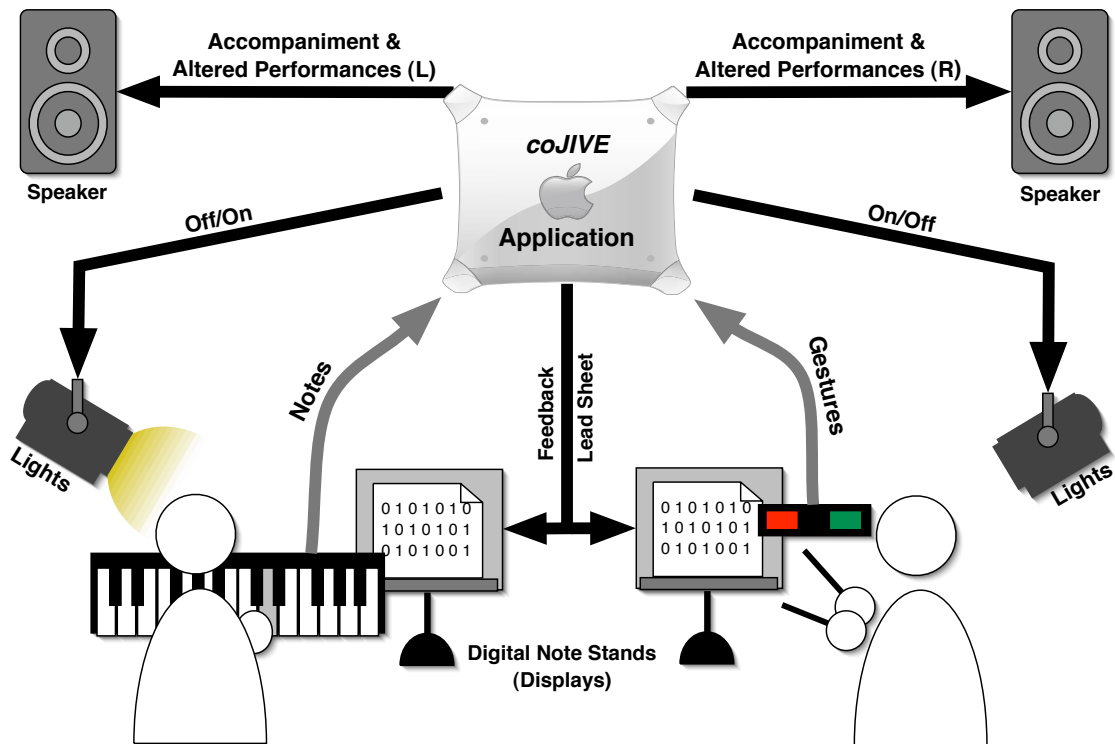


Figure 3.1: *coJIVE*'s main interaction concept.

Nishimoto et al. [2003] describe a set of requirements that should be satisfied when designing such new instruments:

- Low initial barrier. Easy entrance to performing with the instrument must be assured. Novices should almost immediately be able to create satisfying results to keep them interested in using the instrument further. In the case of musical performances, such results would entail harmonic sounding melodies superior to what the user could create with traditional instruments.
- Room for improvement. The instrument should hold the possibility for the user to extend his skills in using it by rehearsing. This improvement should, of course, be comprehensible for the user so that the prospect of continuing training will seem beneficial.
- No limit inferior to traditional instruments. Finally, the limitations to performances using the instrument should not be more restrictive than those for conventional instruments. This can be achieved by falling back to the control principles of those musical instruments.

They claim that the key to satisfy these requirements is customisability of the instrument. Therefore, a design principle for this kind of new instrument aimed at facilitating musical expression was developed. This principle is based on two aspects that should be taken into account when creating such instruments:

1. Reduction of unnecessary degrees of freedom. Limiting expressive parameters (e.g., pitch, velocity) to those values the system computed as appropriate.

2. Specialising the interface for songs to be played. Customising the instrument based on the song that is currently played.

The systems presented in the related works chapter concerned with substituting for the user's lack of abilities all took approaches that were based on these aspects: they either limit the number of available choices (i.e., notes), or take over control to a certain degree by redirecting the input.

In addition to song-based customisation, users can clearly benefit from a system's ability to adjust to their specific level of expertise: systems focusing on novices may be too limiting for musicians with a classical education, while the versatility of traditional instruments is too much for novices to cope with. So a main criterion for evaluating the quality of a new instrument is its capability to adapt to the user's skills and satisfy her needs in terms of musical performance. This should, of course, not only lead to a restriction of the player's scope, as suggested by Nishimoto's design principle, but also enable a player to perform in more advanced ways beyond his usual way of playing. One way to achieve this is to diverge from the traditional one-to-one relation in playing instruments: triggering a note on traditional instruments (by pressing a key, plucking a string, etc.) will always lead to the sounding of that particular note alone. Using one-to-many relations, an instrument can offer an easy approach to complex sound structures (e.g., chords, short melodies) resulting in more complex performances even for novice players.

As a starting point for the development of *coJIVE*, the systems presented in the chapter on related work and the research conducted during their development were examined. The research revealed several directions to take when trying to facilitate musical performances, presenting solutions to problems connected to these directions. For the work at hand, two steps had to be taken:

1. Finding solutions for the directions taking into account and benefiting from the capabilities of the back-end.
2. Modifying and extending given solutions for their application in jazz improvisation and inter-working among each other.

Beyond that, new directions not covered by the related systems due to different perceptions on the topic of jazz improvisation or diverging goals had to be identified and accounted for with special solutions.

4. A First Prototype

Based on the design principle described in the last chapter, another close look at the related systems was taken to derive a set of mechanisms; *coJIVE* would need several features to meet its task of facilitating jazz improvisation by substituting for a player's missing abilities and experience. Therefore, the system also needed to be adjustable to the users' levels of expertise.

To enable users to perform in a jazz session, the system also had to provide an accompaniment: for the players to choose the right timing for the notes and verify their consonance in the context of the song, generated drum- and bass-parts should offer a rhythmic and harmonic foundation. Furthermore, a clear structure for the session needed to be defined and assured. In normal jazz sessions, the musicians have several modes in which a performance can be structured; the most basic of these modes is the alternation of solos, in which they take turns in soloing. As novices usually do not know about such structures in performances, using this mode was the obvious choice because the emerging structures can be easily comprehended.

To attract a wide spectrum of users, two different interfaces were chosen: the MIDI keyboard uses the input method of traditional pianos (keys) and thus offers precise control over what is played; MIDI-wise it sends *NoteOn* and *NoteOff* messages including a value representing the note to be played and the velocity, which indicates how hard the key was pressed and how loud the note needs to be played. The system's task concerning this interface was to offer the possibility to bypass the need for precise control (and thus training and experience). The second interface, a set of infrared batons that recognise gestures, provides more fuzzy control; it is deployed using a xylophone-based metaphor by recognising hitting-gestures in mid-air and triggering corresponding notes. This procedure prevents precise control as no targets to hit are present. Thus, a player has no real orientation for placing his gestures besides moving the gestures left or right to lower or raise the pitch.

To allow evaluation and further enhancements of these solutions, *coJIVE* was created using a development process based on exploratory prototyping: several prototypes were built to evaluate ideas and enhanced iteratively in *DIA cycles*. These development cycles are conducted repeatedly and consist of three phases (each contributing its first letter to the name *DIA*): the design phase, in which the functions of a system

Chord symbol	Scale
$B\flat^{maj7}$	$B\flat^{maj}$
G^{alt}	$A\flat^{mel.min}$
C^{min7}	$B\flat^{maj}$
F^{alt}	$G\flat^{mel.min}$

Table 4.1: Scales for the chord progression. The analysis of the sequence of chords delivers one scale per chord.

are specified, the implementation phase in which these functions are implemented, and finally the analysis phase that is used to evaluate the functions and their implementation. For the work on *coJIVE*, the analysis was conducted using user studies to assure a user-centred design of the system.

The first DIA cycle was merely concerned with musical assistance; the system support in this area needed to be implemented before confronting the users with the concept of jazz sessions. Accordingly, the collaborative aspect was neglected by the first prototype and only the keyboard was used: a second instrument was not necessary at the time, and the keyboard, being the more precise of the two interfaces, implied a more thorough support by the system. To obtain results fast, it was decided to deploy rapid prototyping by using Max/MSP [Cycling '74, 2004], an environment for graphical composition of applications.

4.1 Design

To allow a fast implementation, the decision was made to use static methods and mechanisms based on sets of predetermined data. For this purpose, a generic song with a short sequence of chords ($B\flat^{maj7}-G^{alt}-C^{min7}-F^{alt}$), a metre of $\frac{4}{4}$ (meaning that each bar in the song would consist of four quarter notes), a fixed bass-line and a simple drum pattern were created. The scales and voicings belonging to the chords in the sequence were calculated by hand; the resulting scales are shown in table 4.1. The method used to derive the scales is also described by Klein [2005].

With the related systems in mind, a first set of features for musical assistance was developed. These features were designed to be adjustable to the level of the user playing with the system; some of them are more appropriate to people with a low level of expertise than to those with a high level or vice versa. Consequently, a set of user levels was defined that was meant to incorporate a wide spectrum of people. Finally, these levels were each assigned a particular subset of the features.

Furthermore, some decisions were reached concerning the visualisation of information for the user; the system needed to display information on the chords assuming that experienced musicians would be able to interpret them and act accordingly. The user should also get some feedback on his performance; a clear distinction between the notes the user played and the notes resulting from the system's processing of the input should be possible, allowing the player to improve while using the system. Figure 4.1 depicts, which parts of the initial concept were implemented in the first prototype.

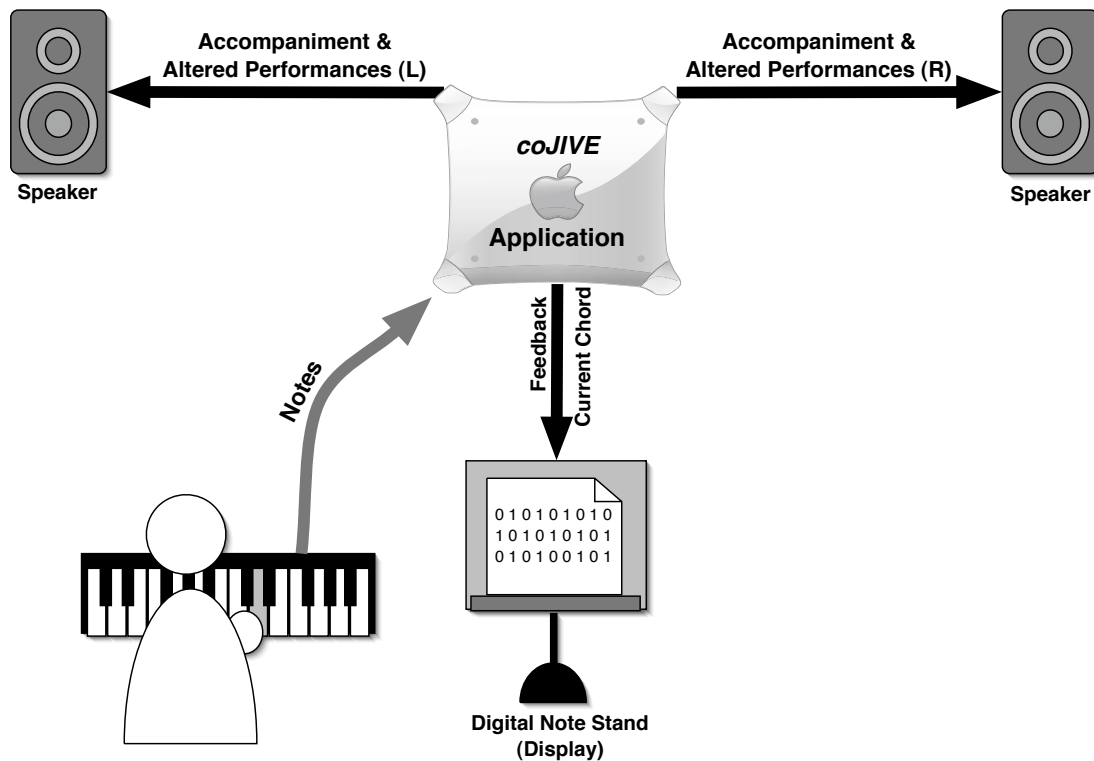


Figure 4.1: The interaction concept of the first prototype.

4.1.1 Features

To create effective ways to even out missing musical education and allow complex performances in groups, the skills necessary for performing in a jazz session need to be identified. In the introductory chapter the three main skills were already named:

1. Precise command of the instrument
2. Thorough knowledge in musical theory
3. Experience in collaborative improvisations

To account for the lack of these skills, some of the directions pointed out by the research on related systems were followed. The given solutions in these systems were evaluated for their use in *coJIVE* and adopted or replaced by more appropriate solutions in the context of jazz improvisation in the attempt to meet Nishimoto's first requirement (*low initial barrier*) for new musical instruments. This section will present the solutions and their emergence.

4.1.1.1 Confine to Scale

An aspect keeping most novices from playing freely is harmony. Without thorough training it is almost impossible to know what notes fit the harmonic context at the current position in the song. By reducing the set of all possible notes (and thus the freedom of the player) to the subset containing the notes that are most likely to sound harmonic, the probability for gross errors is narrowed down accordingly. Notes outside the current scale, although not always dissonant to what the accompaniment

is playing, are more likely to be perceived as being unfit. Nishimoto et al. [1998] have already proven this to be an appropriate starting point for song-based reduction of freedom: their RhyME system calculated the current scale, and mapped the scale notes to the white keys of an octave on the keyboard, starting with the root note on the C key.

Oct.	C	C \sharp (D \flat)	D	D \sharp (E \flat)	E	F	F \sharp (G \flat)	G	G \sharp (A \flat)	A	A \sharp (B \flat)	B
B \flat^{maj}	C	C	D	E \flat	E \flat	F	F	G	G	A	A	B \flat
A $\flat^{mel.min}$	B	D \flat	D \flat	E \flat	E \flat	F	G	G	A \flat	A \flat	B \flat	B
G $\flat^{mel.min}$	B	D \flat	D \flat	E \flat	E \flat	F	G \flat	G \flat	A \flat	A	A	B

Table 4.2: Mapping of the notes in each of the scales depicted in table 4.1. Notes outside of the scale are mapped to appropriate scale notes to prevent dissonant performances.

For *coJIVE*, a similar method was used, while keeping notes at their original place on the claviature where possible. A key was only re-mapped when its original note was outside the current scale, and in that case the key was mapped to a direct neighbour of its original note. Furthermore, the musical context of the notes was taken into account for the re-mapping (i.e., if a scale contains C \sharp but no C, the C key was mapped to C \sharp). So the user was *confined to the current scale*. Table 4.2 shows the mappings used in the different scales.

4.1.1.2 Prohibit Avoid Notes

Grachten [2001] describes how the artificial improviser JIG decides on what notes will fit at a certain moment in time. Beside the outside notes, it takes a closer look at the notes inside a scale: just as notes outside a scale do not always sound disharmonious in the context of that scale, not all the notes in the scale necessarily sound good. This is why these notes are usually avoided by jazz musicians; accordingly they are called *avoid notes*.

Novices, of course, do not know which notes to avoid and should thus be supported correspondingly by the system. So this feature was designed to re-map the keys originally pointing towards avoid notes, just as the previously described feature re-maps outside notes. In the scale progression derived from the chords used in the prototype, only the scale B \flat^{maj} did exhibit avoid notes; as the phenomenon of avoid notes is additionally connected to the chord, that scale, which was used twice in the chord progression (for the chords B \flat^{maj} and C \flat^{min7}), had different avoid notes in both cases. The melodic minor scales used for the other two chords did not contain any avoid notes. Table 4.3 shows an overview of the avoid notes and the re-mappings.

4.1.1.3 Help Triggering Compound Structures

Previous systems (such as CiP [Oshima et al., 2002a] or *ism* [Ishida et al., 2004]) tried to correct or divert the users' input towards a more appropriate result with traditional one-to-one relations. The number of notes coming from the system usually could not exceed the number played by the user, leaving the performance at the

Chord	Scale	Avoid Note	Mapped to
$B\flat^{maj7}$	$B\flat^{maj}$	$D\sharp^{(E\flat)}$	D
		E	F
C^{min7}	$B\flat^{maj}$	A	$A\sharp^{(B\flat)}$

Table 4.3: Re-mapping of avoid notes. The avoid notes are identified based on the underlying chord and scale and are mapped to other scale notes.

abundance the user can provide. Accordingly, the result could not remotely be compared to what experienced musicians are able to play, as they in general do not only play melodies made up of single notes but use chords to provide their performances with a certain richness.

An inexperienced performer, of course, cannot be expected to know the chords in a song, and it is even less likely for her to play them. To enable this kind of user to achieve a certain complexity within a performance, a mechanism for easy access to compound sound structures such as chords needed to be provided. For that purpose the common one-to-one relation needed to be replaced by a one-to-many relation. Again the decision was made to re-map certain keys; every key below $C5$ (middle C) was appointed a specific voicing of the current chord, which is build on the note assigned to the key by the previous two features (with that note being the lowest note in the voicing). That way, the re-mapping followed the order of the keys with all notes above the key pressed, while a mapping to a voicing with the note assigned to that key in the middle would not necessarily reflect the movement on the claviature (e.g., the voicing for the C key sounds higher than the one mapped on the D key because of the notes' arrangement within the voicing). With a clear distinction as to where this feature applied (all keys up to $C5$), the user was able to make a distinct choice where and when to use it. Table 4.4 depicts the mapping of voicings of the chords, each made up of four notes, to each scale note.

4.1.1.4 Offer Automatic Structuring

The structure of a collaborative jazz performance is usually determined before the session, but this structure is sometimes changed spontaneously during the performance. This is based on a thorough comprehension of rules that are used to define this structure and how to communicate changes (e.g., with certain gestures). Without knowledge of these rules users are very unlikely to be responsive to each other, especially with a system like *coJIVE* that focuses on providing them with an entertaining and interesting experience. Walker [1997] states that “*Jazz performances are governed by a well-known structure. The computer should model this structure and track progress through it during the performance*” as a rule for his artificial improviser *ImprovisationBuilder*. Accordingly, the feature of automatic structuring was designed to enable the system to define and control the structure of the performance. As the system generated an accompanying performance for the users, such a mechanism was needed all the more.

As the Max/MSP prototype was not concerned with collaboration, the deployment of this feature was postponed to the next prototype.

Chord	Scale	Note	Mapped voicing
$B\flat^{maj7}$	$B\flat^{maj}$	C	C-D-F-A
		D	D-F-A-C'
		F	F-A-C'-D'
		G	G-C'-D'-F'
		A	A-C'-D'-F'
		$A\sharp^{(B\flat)}$	$A\sharp^{(B\flat)}$ -D'-F'-A'
G ^{alt}	$A\flat^{mel.min}$	$C\sharp^{(D\flat)}$	$C\sharp^{(D\flat)}$ -F-B- $D\sharp^{(E\flat)}$,
		$D\sharp^{(E\flat)}$	$D\sharp^{(E\flat)}$ -F- $A\sharp^{(B\flat)}$ -B
		F	F- $G\sharp^{(A\flat)}$ -B- $D\sharp^{(E\flat)}$,
		G	G-B- $D\sharp^{(E\flat)}$ '-F'
		$G\sharp^{(A\flat)}$	$G\sharp^{(A\flat)}$ -B- $D\sharp^{(E\flat)}$ '-F'
		$A\sharp^{(B\flat)}$	$A\sharp^{(B\flat)}$ -B- $D\sharp^{(E\flat)}$ '-F'
		B	B- $D\sharp^{(E\flat)}$ '-F'- $G\sharp^{(A\flat)}$,
C ^{min7}	$B\flat^{maj}$	C	C- $D\sharp^{(E\flat)}$ -G- $A\sharp^{(B\flat)}$
		D	D- $D\sharp^{(E\flat)}$ -G- $A\sharp^{(B\flat)}$
		$D\sharp^{(E\flat)}$	$D\sharp^{(E\flat)}$ -G- $A\sharp^{(B\flat)}$ -D'
		F	F-G- $A\sharp^{(B\flat)}$ - $D\sharp^{(E\flat)}$,
		G	G- $A\sharp^{(B\flat)}$ -D'- $D\sharp^{(E\flat)}$,
		$A\sharp^{(B\flat)}$	$A\sharp^{(B\flat)}$ -D'- $D\sharp^{(E\flat)}$ '-G'
F ^{alt}	$G\flat^{mel.min}$	$C\sharp^{(D\flat)}$	$C\sharp^{(D\flat)}$ - $D\sharp^{(E\flat)}$ -F $\sharp^{(G\flat)}$ -G
		$D\sharp^{(E\flat)}$	$D\sharp^{(E\flat)}$ -F $\sharp^{(G\flat)}$ -A- $C\sharp^{(D\flat)}$,
		F	F-A- $C\sharp^{(D\flat)}$ '- $D\sharp^{(E\flat)}$,
		F $\sharp^{(G\flat)}$	F $\sharp^{(G\flat)}$ -A- $C\sharp^{(D\flat)}$ '- $D\sharp^{(E\flat)}$,
		$G\sharp^{(A\flat)}$	$G\sharp^{(A\flat)}$ -A- $C\sharp^{(D\flat)}$ '- $D\sharp^{(E\flat)}$,
		A	A- $C\sharp^{(D\flat)}$ '- $D\sharp^{(E\flat)}$ '-F'
		B	B- $D\sharp^{(E\flat)}$ '-G $\sharp^{(A\flat)}$ '-A'

Table 4.4: Mapping of chord voicings. The one-to-one relation of traditional instruments is breached by triggering four notes that make up an appropriate chord voicing for one note triggered by the user.

4.1.1.5 Restrict Expressive Parameters

Expression in musical performances has already been the subject to some research. Oshima et al. [2002a] shifted the focus of the user from what is played to how it is played. A study on expressiveness in piano improvisations presented by Baraldi and Roda [2001], in which several subjects were asked to express musical intentions (e.g., resolute, obscure, sweet, or simple) with one note and varying freedom of expression (choice of the note, velocity and duration were ruled out one after another), showed relations between the expressive intentions and expressive parameters.

In the context of collaboration this can be used to support the improviser's intention by carefully diminishing the other players' input. An appropriate parameter for this purpose is velocity, which affects the volume of the notes played. When accompanying a soloist, a player should make sensitive use of this parameter, as loud notes at the wrong time can disturb the solo and interfere with the soloist's expressive intention (e.g., if the improviser tries to create a soft melody).

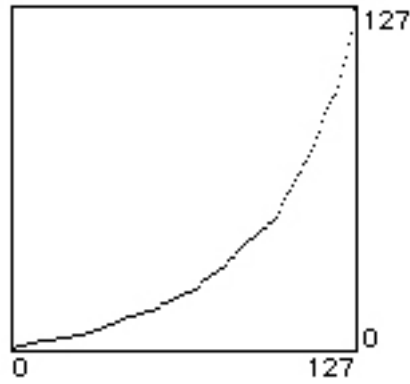


Figure 4.2: Bent velocity curve for accompanying players. This new mapping of input velocity to actual velocity keeps the accompanying player from “drowning” the solo with loud notes.

The mechanism to ensure this behaviour by systematically intervening with the accompanying users’ playing was designed to *bend* the velocity curve of their instrument (Figure 4.2); no velocity value was to be prohibited from being used, but the higher values should become less likely to be reached. Although this feature was implemented, it was not used at that stage, similarly to the automatic structuring, because of the focus on musical support.

4.1.1.6 Discarded Ideas

Besides the features mentioned above, some other ideas came up during the development process. As they were not applicable in the field of jazz improvisation, they were not used for the system; the reasons for their dismissal will be discussed in this section.

The approach to facilitate access to compound structures is not only applicable to the harmonic dimension (multiple notes played at the same point in time, i.e., chords), but also to the dimension of time: in accordance with the mental model of motif-playing presented by Grachten [2001], *coJIVE* could offer playing short melodies at the push of a key to equip players with a set of standard motifs. This idea however was neglected as it contradicted with the concept of improvisation the system follows: as the motifs used by jazz musicians are taken from their personal repertoire developed by listening to, analysing, and playing pre-composed melodies, using them can clearly be perceived to be a creative contribution; a predefined set of such motifs, however, would reduce the creative influence a player has when triggering one of these motifs.

CIP [Oshima et al., 2002a] restricts a player to play the sequence of notes stored in a database; in improvisation, of course, this would be far too limiting. A variation in which the player is directed towards the previous solo by making the next note from the previous solo more probable could overcome this limitation. This would, of course, imply dynamic, probability based re-mapping of notes, while in the first prototype static re-mappings were used. Furthermore, the calculation of note probabilities is a task performed by the back-end framework [Klein, 2005].

Another interesting feature supplied by current composition tools is the *quantisation* of recorded notes: if triggered, it moves the notes to more appropriate temporal positions (e.g., on a bar or a beat) to straighten out inaccuracies generated by the player while recording. A quantisation of the performance while a user plays could furnish his performance with more accuracy. However, this feature would entail some problems: correcting a note's temporal position can, of course, only happen by delaying it since otherwise the systems would need to foresee the occurrence of the note, which is impossible. But delaying a note would resemble latency that most media-based systems try to avoid it often irritates the user; with recognisable latency, she cannot connect her actions to the system's output. Accordingly, the idea of using quantisation on notes was deemed inappropriate.

4.1.2 User Levels

With the features at hand, one question needed to be answered: which of the features does one type of user need? The user type, of course, refers to the musical abilities of the person using the system. For the assignment of the features, a system of four skill levels was created roughly categorising the group of future users, thereby meeting the second of Nishimoto's requirements: going from the lower levels to the upper ones would offer enough room for the users to improve. This section will explain and discuss the system of user levels.

4.1.2.1 Professional

Musicians used to performing in jazz sessions are summarised in this category. Professionals usually have several years of experience performing and improvising in groups; they are able to interpret chord symbols, calculate scales and communicate with other musicians during sessions. No real support is needed for these musicians; any interference by the system would only limit and disturb them.

So, none of the features were applied on the professional level to enable this type of user to play just as she is used to; this way the last of Nishimoto's [Nishimoto et al., 2003] requirements was satisfied: the system would have no limit beyond that of traditional instruments.

4.1.2.2 Expert

The group of experts comprises musicians with a classic musical education who may have had some brief contact with jazz. These musicians have a good command of their instrument and have years of experience in performing classical pieces. Yet, they are not really used to playing freely without a score to base the performance on.

As classical pieces often stay in one key and potential key changes are noted on the score sheet, experts are not accustomed to determining the scales to use themselves; actually, scales are rarely used in these pieces. So for experts assistance in calculating scales for the given chord structure is necessary for them to diverge from their usual style of playing (by reading notes from the score).

Accordingly, the only feature applied on this user level is *confine to scale* so experts would not have to think about the key in which to play and could focus more on the spontaneous creation of melodies.

4.1.2.3 Advanced

Musicians in the middle of a classical education are called advanced in the context of this categorisation. They are still in the process of developing precise command of their instrument and are still lacking knowledge in terms of musical theory. As they are still rehearsing playing sheet music, improvisation is a topic they have not yet dealt with.

Besides the need for a determination of what scales to use, this group of people should also be assisted in terms of controlling the instrument. Their not yet accurate way of handling the instrument should be accounted for, as well as their lack of experience in handling complex structures. Only if these handicaps can be overcome they can start to improvise.

Confine to scale was applied here to relieve an advanced player of the decision what scale to use, which would be complicated by his missing knowledge in music and jazz theory. This train of thought also led to the application of *prohibit avoid notes*. To provide these players with the means to create more complex performances, which for them is usually limited by their lack of experience, *help triggering compound structures* was applied also.

4.1.2.4 Novice

Novices are people with hardly any or no experience in music besides listening to it. They have never really played an instrument, no knowledge in the field of musical theory and thus cannot read notes or interpret chord symbols. To free them from the fear of failure, they have to be provided with a lot of support. And yet, this group of players is the most likely to start playing freely, if their missing skills can somehow be compensated, as they have no acquired way of performing music to fall back on.

As the features at that point did not allow further differentiation (besides applying and not applying them), this user level exhibited the same premises as the advanced level. Therefore, it was also provided with the features *confine to scale*, *prohibit avoid notes* and *help triggering compound structures*. The decision was made to keep both user levels despite their similarity to keep a consistent classification for the upcoming prototypes.

4.1.3 Further Design Decisions

To offer an overview to the user containing all relevant pieces of information and conveying the principles that the system is built upon, some more decisions were made.

The currently valid chord symbol should always be displayed: experienced musicians can therefore decide on a scale by themselves and perform like being in a common jazz session. Furthermore, the user's input and the corresponding output by the system should be visually comprehensible for the user to understand the system's corrections and enhancements to her performance; to achieve this comprehensibility and to stick to the keyboard as the primary interface, both input and output were to be displayed on virtual keyboards. In future version of the system, this could be also implemented on the real keyboard.

To clarify the differences between the four user levels and the corresponding amounts of support, the levels should be visually separated; each level needed to be appointed its own space on the screen containing representations of the features applied in that level. By this visualisation, the user was supposed to get an idea of the degree of support the system was providing and its differences in comparison to the neighbouring levels.

Finally, the accompaniment the system was supposed to provide should be a simple but steady foundation for the session. Therefore, it was decided to use a swing pattern for the drum track, as swing is the style most people can directly relate to jazz. No variations should be made during playback to not disturb or irritate inexperienced users. The bass should also use a simple *walking bass* pattern. This pattern plays a note on each beat “walking” up and down the pitches. That way, the bass would stress the beat without offering a rhythmic pattern for the players to follow.

4.2 Implementation

With Max/MSP, rapid prototyping was enabled by avoiding the need for developing graphical user interfaces and MIDI connections to access the devices; Max/MSP offers a variety of objects and mechanisms for these purposes. That way, the analysis could be conducted much earlier leaving more time for the ongoing work, which then could be conducted based on the results.

Furthermore, some hardware was necessary for providing all the aspired features. This section will first introduce all the hard- and software used for the realisation of the design to point out the difficulties and possibilities implied by these aspects. After that, the implementation of the different system elements is described to mediate the inner workings of the prototype, which is depicted in Figure 4.3.

4.2.1 Environment

This section presents a brief overview of the environment, in which the first *co-JIVE* prototype was developed and deployed. Beside the hardware used for running and controlling the prototype, a short description of the Max/MSP environment is provided.

4.2.1.1 Hardware

PowerMac Workstations

The Apple workstations on which the system was developed were each equipped with two G5 PowerPC processors with 2 Ghz and 512 MB of RAM. For displaying, 23” Apple Cinema Displays with a resolution of 1900*1200 pixels were used. The workstations furthermore offered several USB and FireWire ports, Bluetooth and 802.11g connectivity.

M-Audio Radium 49 MIDI Keyboard

This keyboard comes with 49 non-weighted keys with velocity, a pitch bend wheel, eight rotary faders and eight sliders, the last two programmable to modulate arbitrary MIDI parameters. The device can be connected to a computer via a USB port or integrated into a MIDI network via a MIDI port. Furthermore, the keyboard is equipped with a port for a power supply unit and a compartment for batteries.

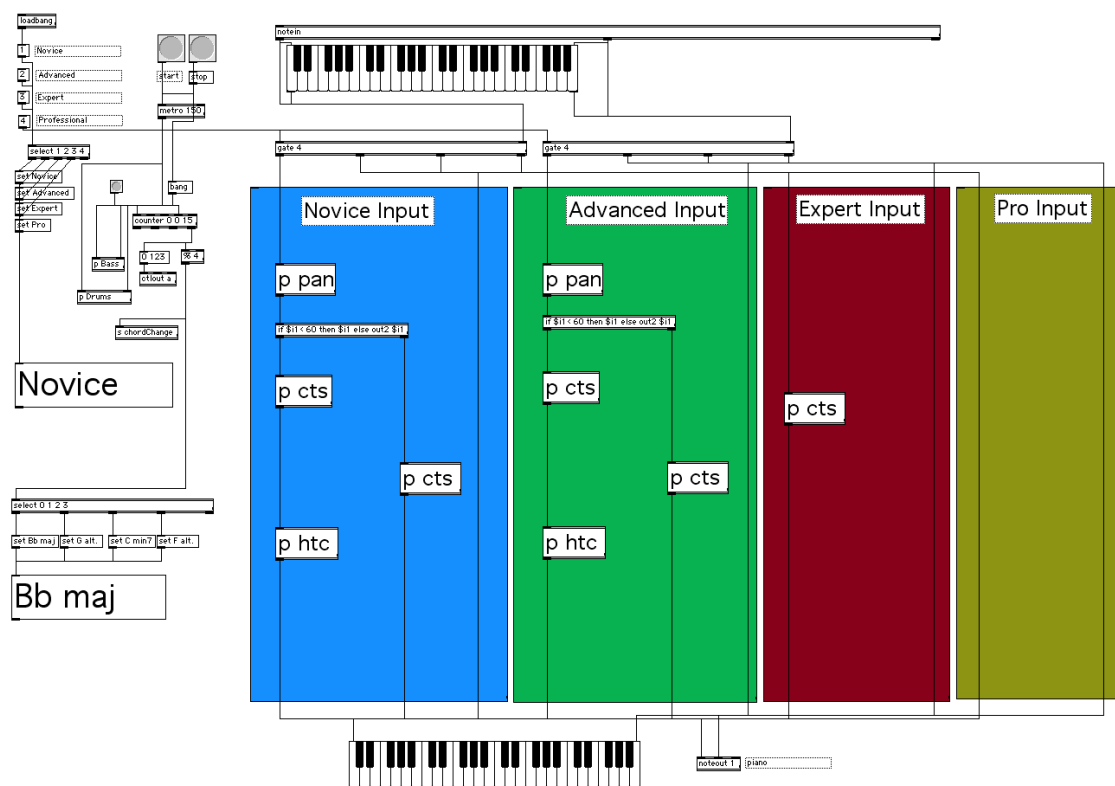


Figure 4.3: The Max/MSP prototype.

4.2.1.2 Max/MSP

Max/MSP is a graphical environment especially aimed at multimedia projects; initially, it was developed for MIDI-based applications, but as computers became faster and more reliable, it was extended to incorporate features for processing digital audio stream and images. The environment offers fast access to MIDI-devices and graphical user interfaces without the need to actually write your own code, thereby supporting creative layouts.

A patch field metaphor based on connectable objects is used to create applications: one can choose from a large set of such objects, each created for a specific purpose; these objects have input and output ports (called **inlets** and **outlets**) which can be connected with each other via patch chords. That way, a signal (e.g., a MIDI message) can be routed and processed in numerous ways. Besides these functional objects there are **message** objects, which can contain several sorts of messages (text, numbers, etc.) to be triggered, for example, on specific events or conditions.

Groups of connected objects are assembled in patches, which can be further structured by using sub-patches with individual input and output ports; the sub-patches themselves can contain further sub-patches, allowing the creation of complex hierarchies. The implementation section in this chapter will offer more insights into this method of building applications.

4.2.2 System Clock

For a common timeline to base playback and performance on, a system clock generating time signals was needed. This was implemented using a **metro** object provided

by Max/MSP; a metronome generates clock ticks in a specific frequency (defined by the time of delay between two ticks in milliseconds), which can then be counted and thereby interpreted to be specific points in time.

The resolution of time was set to a 16th note, meaning the metronome would generate 16 ticks per bar. With a tempo of 100 beats per minute (bpm), the corresponding delay time to be used as the parameter for the metronome object is 150 milliseconds, calculated as follows:

$$100 = \frac{60000 \text{ msec per minute}}{4 * x \text{ msec per beat}} \Rightarrow x = 150$$

As a beat occurs on each quarter note, 60,000 milliseconds (one minute) is divided by the delay time of a quarter note (four times the delay time of a 16th note) to receive the beats per minute. With a fixed value for the bpm, the equation can be transformed to display the result for the delay time of a 16th note (x).

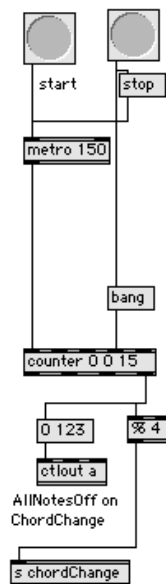


Figure 4.4: Metronome object acting as System Clock.

As the metronome ticks 16 times per bar, a **counter** object is used to indicate the beginning of a new bar. It counts every 16th note, which represents the start of a new bar, each time triggering the sending of a message propagating a chord change; this message contains the number of the new chord (which is kept between 0 and 3 with the help of a **modulo** object to assure a loop over the four predefined chords, each lasting a bar) and is sent using a **send** object (abbreviation: “s”) and can then be received throughout the system by using **receive** objects (abbreviation: “r”).

The metronome (and thus the clock) is started and stopped by **button** objects; if clicked, these objects send out a *bang*, a generic signal that can be used to trigger events (e.g., start a **metronome** object) or the sending of a message (like the message ‘stop’ used to stop the ticking of the metronome). The stop button is also connected to a certain input of the **counter**, which if banged resets its count to 0.

Furthermore, on each chord change an *AllNotesOff* message is sent to avoid dissonance in the performance: as some of the features use re-mapping of keys, a key that is held down during a chord change may be re-mapped to another note. If this key is released, a *NoteOff* message for the new note mapped to that key is sent. The old note that had initially been triggered by that key however still sounds on, even if it does not belong to the new scale. The *AllNotesOff* message on changing a chord bypasses this effect.

4.2.3 Accompaniment

The accompaniment was implemented in sub-patches: both bass and drum patterns were encapsulated in their own patches, which are fed with the clock ticks and the stop signal through their input ports, to separate the accompaniment from the rest of the system's functions. The creation and triggering of MIDI notes is described in the next section.

4.2.3.1 Bass

The pattern used to create the walking bass was derived from the previously determined scales to emulate the behaviour of a jazz bass player: each chord change is accompanied by playing the root note of the new chord; from there the bass walks up or down the range of pitches, eventually changing its direction to draw near the next chord's root note before the next chord change occurs.

As the walking bass only plays on quarter notes, a **counter** object is used to count every 4th tick. The derived count of quarter notes, representing the position in the four bars, is kept between 0 and 15 by a **modulo** object (as four bars are made up of 16 quarter notes in the $\frac{4}{4}$ metre). A **select** object then triggers a message containing the note value according to the quarter note count. If a stop signal is received, the counter is set back to 0.

4.2.3.2 Drums

The drum patch was created to emulate a drummer playing a swing pattern: only open and closed hi-hat, as well as a rim shot sound were used avoiding the dominant sounds of a snare or kick drum.

Again, a **counter** object was applied to count the quarter notes, whose count was held between 0 and 3 by a **modulo** object to achieve a short loop. A **select** object interpreted the quarter notes count and fired messages accordingly: on the first quarter note (0), an open hi-hat (note value 46) is triggered; on the third quarter note, a closed hi-hat (note value 44) and a rim shot (31) are triggered. To achieve a swing effect, 75 milliseconds after the fourth quarter notes (which equals the time of an eighth note at 100 beats per minute) another closed hi-hat is played; a **delay** object is used to achieve this. Similar to the bass patch, the stop signal is used to reset the counter

4.2.4 MIDI Connections

As already mentioned above, Max/MSP was created for the use with MIDI-based interfaces and devices; it offers several objects to receive, create and send MIDI

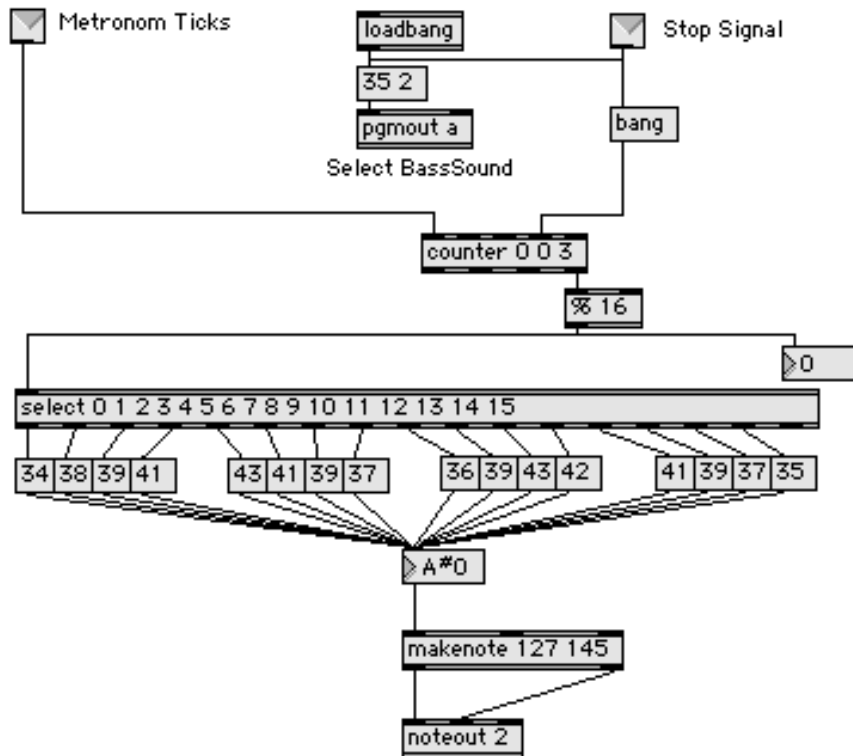


Figure 4.5: The bass patch.

messages. These objects usually need parameters to specify a port (given by a character) and channel (a number between 1 and 16) to send to or receive from.

`notein` objects receive *NoteOn* and *NoteOff* messages and then send the note value, velocity and channel on which the signal was received through their three output ports. The prototype deploys a `notein` object to receive the user's input through a specified port. Its counterpart in Max/MSP is accordingly called `noteout`: these objects send *NoteOn* and *NoteOff* messages on a specified port and channel. They have three input ports for note value, velocity and channel. The first *coJIVE* prototype deploys several of these objects: the notes produced based on the player's input, as well as the bass and drum notes are all sent through separate `noteouts`.

If the duration of a note needs to be set to a fixed value or no *NoteOff* message can be expected, a `makenote` object can be used: it has input ports for note value, velocity and duration; the last two values can also be set as static parameters in the object. If a `makenote` object receives a note value, it will send note value and velocity through its output ports followed by the note value and a velocity value of 0 after the given duration to assure the previously triggered note is turned off again. A `makenote` object is used in the drum patch to delimit the duration of the closed hi-hat.

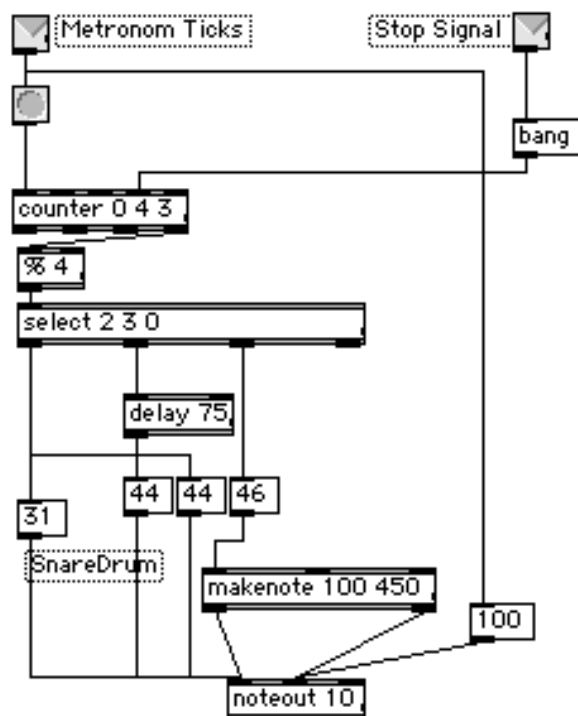


Figure 4.6: The drum patch.

4.2.5 Visualisation

To visualise information on song and performance as well as segmentation of functionality in the system mostly simple means were used: special objects could be used, as well as the deployment of sub-patches (for bass and drum) and purpose-based arrangement of elements in the main patch. The user levels, for example, were represented by differently coloured fields to achieve an obvious separation; their implementation is discussed further in the next section.

4.2.5.1 Chord Symbols

To offer the user insights into the song's structure, the current chord needed to be displayed. For this purpose, simple textual representations of the chord symbols were deployed, which could easily be changed when a chord change occurred.

A `select` object was fed with the chord change value mentioned in section 4.2.2. In accordance to the new chord's number, a `message` object containing 'set' and the new chord symbol was triggered; with this message, the contents of a further `message` object can be set to the new chord symbol. This last `message` object was enlarged so it could be used as a display for chords. Figure 4.7 shows this arrangement.

4.2.5.2 Feedback

The feedback visualisation was chosen to offer consistent look and feel along with the instrument: the player's input as well as the output derived from the input by revision through the system, were displayed using `keyboard` objects; these objects resemble keyboards and can highlight the keys. Output and input ports for note

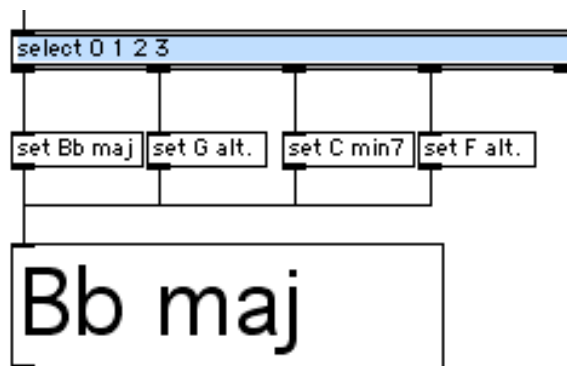


Figure 4.7: Visualising the chord sequence.

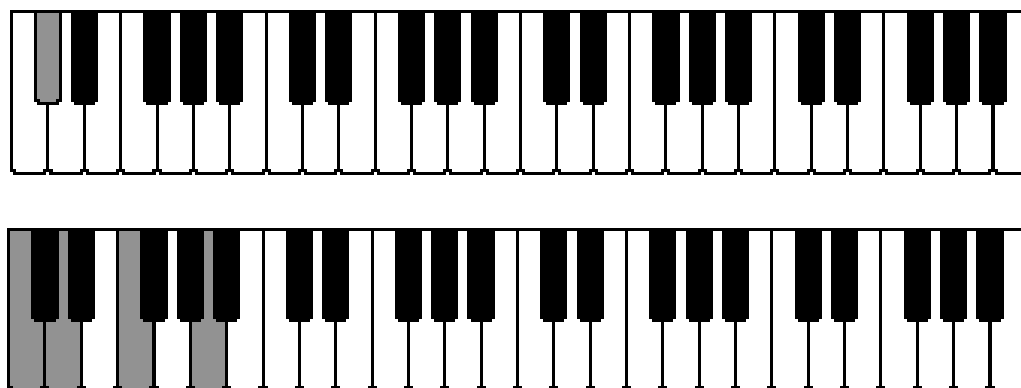


Figure 4.8: Visualising player input(upper) and the system's response(lower).

value and velocity allow displaying of notes as well as triggering notes by clicking keys on the object.

Figure 4.8 shows feedback for both input and output of a $C\sharp(D\flat)$ note: the note is processed by the *confine to scale* (re-mapping to C) and *help triggering compound structures* features (extension of a single note to a chord voicing). With this kind of feedback, the user was supposed to comprehend the system's mode of operation and reconstruct possible corrections.

4.2.6 User Levels

Selection of user levels was enabled by messages containing the numbers of the levels. These numbers were then fed to two **gate** objects, which would open one of four output ports for the messages fed to another input on the gate; one of the gates received the note value, the other one received the velocity. With these gates, a selective routing of the input data to the selected user level.

As the coloured fields only were used to visually separate the user levels; the function of the user levels in the prototype were to assemble and connect the sub-patches containing the corresponding features. The professional level was not equipped with

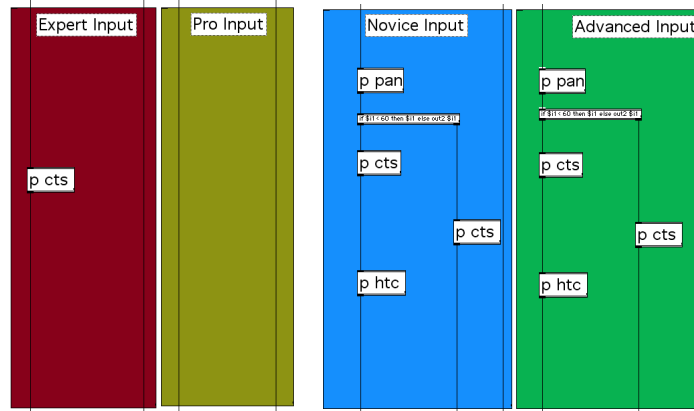


Figure 4.9: Assembly of features for the different user levels.

any of the features, so here the input data was only handed over to the keyboard visualising the output and the `noteout` object triggering the MIDI notes, without doing any processing. Expert users should be provided with the *confine to scale* feature; this feature only processes the note value, so this value had to be routed through the sub-patch of *confine to scale* while the velocity could be directly handed to feedback and output. Figure 4.9 depicts these two user levels in the prototype.

?? shows the implementation for the advanced and novice levels; as mentioned earlier, the combination of features is the same in both levels due to the features' static nature. Here too, the velocity was never processed. The note value however was first routed through the sub-patch containing the functionality of the *prohibit avoid note* feature, followed by a routing through an if-then-else object, which decided whether the note was lower than C5. This distinction was primarily necessary for the *help triggering compound structures* feature but was done even before the *confine to scale* feature to make sure every note below C5 (which could in some cases be changed to C5 by *confine to scale*) was routed through *help triggering compound structures*. So for these notes, *confine to scale* and *help triggering compound structures* were connected in series, while all other notes were only routed through *confine to scale*.

4.2.7 Features

To enable reuse of the implementation of each feature in several user levels, their functionality was encapsulated in sub-patches. These sub-patches were equipped with an input and output port to allow flexible combination of the features. As the features all worked depending on the current chord, they all had a receiver object to receive the chord change signal. The chord change number (increased by 1) was used to open `gate` objects that routed the note value to the area in the sub-patch representing the context of the current chord. Beforehand, this note value was separated from its octave (resulting in a value between 0 (C) and 11 (B)) to achieve a consistent behaviour in all octaves, using a `modulo` object; the resulting notes were of course brought to the correct octave before leaving the sub-patch (with a `division` and `multiplication` object to calculate the octave and an `addition` object to reconnect note and octave).

4.2.7.1 Confine to Scale

For every scale, a `select` object was used to interpret the incoming note value. Which of these `select` objects obtained the note value for further processing was decided by the `gate` mentioned above. For every possible note value the corresponding `select` object triggered a message containing the note value derived from the mappings depicted in table 4.2.

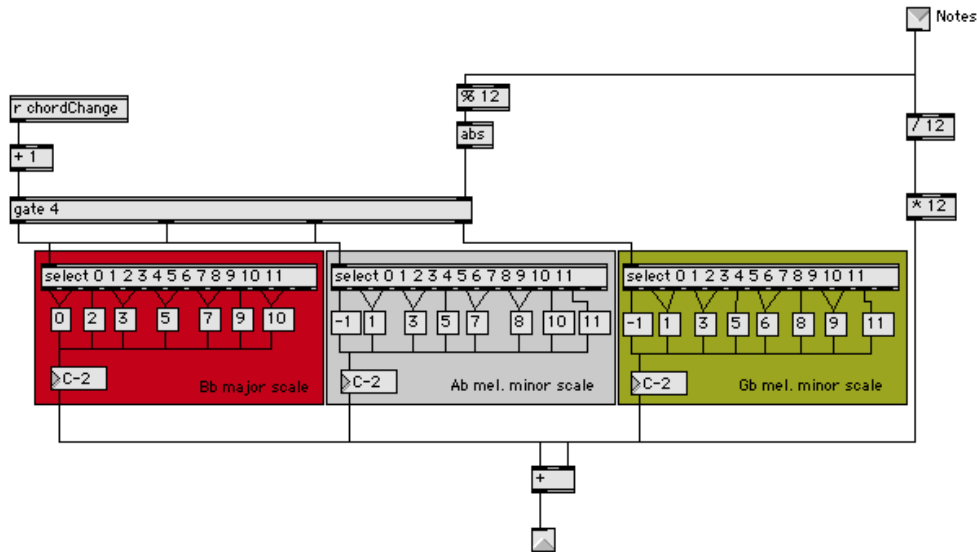


Figure 4.10: Implementation of Confine to scale

4.2.7.2 Prohibit Avoid Notes

Similar to *confine to scale*, this feature used `select` objects to trigger re-mappings of note value if necessary. As all other notes were of no concern to this feature, they were just passed on making use of the `select` objects' last output port, which would hand out the received signal if none of the given values matched.

4.2.7.3 Help Triggering Compound Structures

The `select` objects in this feature were not used to trigger only single note values but messages with lists of note values, each list representing a chord voicing. These lists of values then had to be turned into single note values that could be triggered one after another; to achieve this, `unpack` objects were deployed. So, the voicings could then be treated like a series of single notes.

4.3 Analysis

With this working prototype at hand, an analysis of how effective the features and their combinations were in terms of assisting human players in improvising over jazz tunes could be performed. Furthermore, the allocation of features to the user levels as well as the 4 level classification of users in the system needed to be checked for appropriateness. So, a user study including a series of user tests was conducted. The results to be received from these tests were meant to lead the further development of the system.

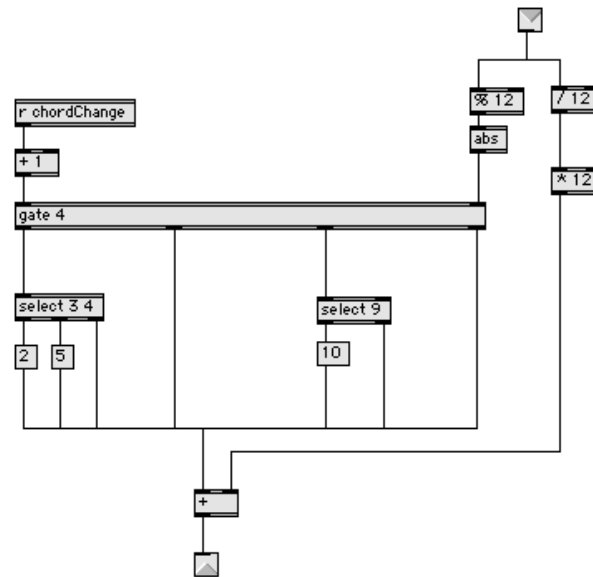


Figure 4.11: Implementation of Prohibit avoid notes

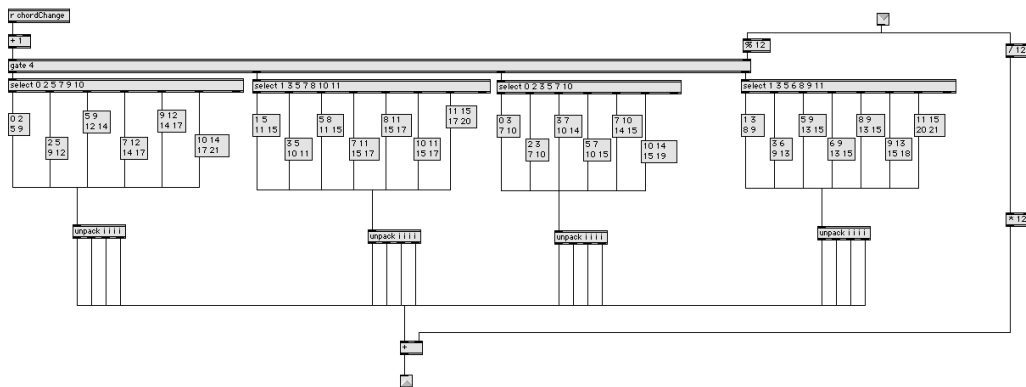


Figure 4.12: Implementation of Help triggering compound structures

4.3.1 User Tests

The user study was conducted with eight test subjects. This rather small number of users was chosen to obtain a rough overview of the system's effectiveness; as the prototype itself only made a rough estimation of the users' skills and the provided support, a more thorough investigation was postponed until the system would reach a more complex state.

Four of those subjects were novices with hardly any or no experience in music theory and performing; the other four subjects were musicians of different levels, one having experience in improvising on the piano in jazz sessions. All of the subjects were familiar with using computers, which did not effect the results as the Max/MSP-based prototype offered no standard user interface and the subjects did not have to handle the configuration on their own.

The evaluation of the tests was performed with the help of questionnaires that were handed to the subjects after the tests. They were deployed to collect the opinions and experiences of the users concerning the prototype as well as their personal experiences in music to estimate their level of expertise.

4.3.1.1 Procedure

Every test subject was asked to perform with the system in all user levels; only the advanced level was omitted due to its similarity to the novice level. They started with the professional level, followed by the expert level and finally performed in the novice level. This succession was chosen to let them experience the increasing of support, starting with no support resembling the handling of a traditional instrument. Before playing in a level, the users were informed about how the system would behave using the different features; that way they knew what to expect and how to use some of the features (e.g., the chord triggering of *help triggering compound structures*).

The duration of their performance in each level could be determined by the subjects themselves; this was based on the assumption that the duration of a performance would relate to the amount of pleasure a subject experienced while performing. It was expected for each subject to enjoy the level that best approximates his personal level of expertise the most; this level was supposed to offer the most appropriate combination of support and necessary freedom for the subject. To allow a comparison of the values, the duration was noted for each level.

The subjects were not instructed or limited in terms of what to play on the keyboard: they received no score or any other form of notation of melodies, chords, etc. That way, the subjects were supposed to play freely and thus improvise without any disturbances.

4.3.1.2 Questionnaires

The questionnaires were handed to the subjects right after their performances not to lose any of the subjects' opinions and experiences. They contained questions separated in three sections, questions on the musical knowledge and experience of the subject, questions on the course of the test and finally questions on the overall impression.

The first section was included to estimate the subjects' musical skills; this was necessary to allow a thorough interpretation of her statements on the prototype: a novice was expected to have different opinions on the systems in comparison to an experienced musician. The following questions were given in this section:

1. **Do you play the piano/keyboard/organ?**

This was supposed to measure the subjects' familiarity with the interface; the duration of how long one of these has been played by the subject and of potential breaks in playing the instruments were also asked for to narrow down the estimation.

2. **Can you read scores?**

This was one aspect used to approximate the subjects' knowledge in musical theory. It was supported by the question if a score cannot only be read but also played at the same time by the subject.

3. Can you read chord symbols?

Two chord symbols were given and the subjects' knowledge of those was questioned for further insight into this aspect. Their knowledge would, of course, indicate a more thorough knowledge as they are mostly used in jazz while they are evaded in classical music.

4. Do you know what scales are?

For a further investigation of this part of musical knowledge, the subjects were asked if they know the set-up of minor and major scales.

The task of the second section was to acquire an overview of the subjects' reaction to the course of the test. For each user level, they were asked to estimate the duration of their performance and to rate their enjoyment. This was supposed to show if a connection between the experienced pleasure and the perceived duration of the performance existed; the duration were later on compared to the durations noted in the test. It was expected that the difference between perceived and real duration would grow with increasing enjoyment. Further questions relating to the user level the subject enjoyed most were posed:

1. Did the system support you sufficiently?

It was expected mainly for novices to answer this question negatively as experienced musicians would only perceive more support to be limiting in terms of creative freedom. In general, this question was meant to show if the allocation of features to the user levels was suitable.

2. Do you think your performance was harmonic and suitable?

If the system's goal to allow interesting harmonic performances for a wide spectrum of people was already achieved in the first prototype, this question was supposed to be answered positively; a negative answer would indicate the need for more thorough support.

3. Do you think you had control over your performance?

The system would only ensure a satisfying experience if the user could take over a certain degree of creative control; this question was deployed to show if the system was too limiting in that context.

The final section of the questionnaire was concerned with the subjects' overall impression of the system and contained the following questions:

1. Would you like to further on use the system?

As the system was supposed to offer satisfying experiences, this question was used as indicator of how good the system could meet the task.

2. Would you use the system together with others?

To allow an assessment of whether the features in the system (and thus its overall behaviour) were suitable for deployment in collaborative sessions, this question was included in the questionnaire.

3. Personal remarks on the test or the system

This point was added (although not being a question) to acquire anything not covered by the other question and any further thoughts or ideas the subjects would have while using the system

There was no time-limit imposed on completing this questionnaire to not put pressure on the subjects. Otherwise, some aspects may have been forgotten or intentionally left out.

4.3.2 Results

In the test most subjects behaved as expected: the rather inexperienced subjects played carefully to see how the system would react, while the musicians exhibited a more selective behaviour in their performances. Additionally, some aspects so far not considered during research were discovered in the performance of subjects not familiar with the keyboard as a musical interface: they not only were imprecise in their selection of what keys to play but also in playing the; sometimes, they would press two neighbouring keys with one finger or slip from one key to another.

Beyond these insights, the observation of the subjects in the tests already indicated that they were enjoying themselves with the system: especially the novices seemed to act very careful at first but apparently grew more confident with increasing level of support. The results taken from the questionnaires backed this impression: all but two subjects rated the enjoyment they experienced with the system to be increasing along with the level of support by the system. None of the subjects had the impression to be insufficiently supported.

In terms of evaluating their own performance, the subjects made more diverse statements: half of the subjects had the feeling to have control over what they had performed, the other half did not; interestingly, this other half did not entirely consist of experienced musicians in contrast as to what was expected. Three subjects thought that their performance was harmonic, three subjects did not and another two subjects noted to only sometimes have performed something suitable.

Furthermore, a set of interesting remarks and statements were given on the questionnaires. One subject stated that for a personal analysis of the chord progression, an overview showing most or all of the chord symbols would be necessary. The display of feedback on the performance (subsubsection 4.2.5.2) was mentioned by several subjects to be very helpful; one of these subjects additionally stated to have been able to correct some mistake with the help of the depicted correction. Some of the experienced subjects indicated their discontent with the allocation of aiding features in the different user level: the request for a level including a chord triggering feature (*help triggering compound structures*) but without the correction of single notes (*confine to scale* and *prohibit avoid notes*) was made. Finally, the one subject with experience in jazz performances thought of the single note correction to be too restricting.

4.3.3 Discussion

A lot of the aspects of this first prototype did prove to be quite beneficial in terms of the goals of the system: the supporting features seemed to augment the users' personal satisfaction over their performances; visualising input and output provided information for the users that some of them could use to alter their play.

Despite the successful application of these aspects, the tests discovered several points that needed further development:

-
- The information about the song structure (i.e., chord symbol) displayed by the prototype was not sufficient. To enable experienced musicians to perform their own analysis of the song (thereby enabling jazz musicians to follow their usual procedure), several or all of the chord symbols making up a song structure need to be displayed.
 - The prototype only accounted to some extent for the user not knowing *what* to play but not *how*. It should be taken into consideration that a large group of people is not familiar with using a keyboard to play sounds (i.e., if they know how to play no or other instruments) and thus do not exhibit the correct behaviour for this interface.
 - In general, the system should be more flexible in terms of being adjusted to the users' skills and needs. On the hand a categorisation with four user levels seems to be unfit to thoroughly cover the sighted spectrum of users. The aiding features, on the other hand, were themselves too inflexible: *confine to scale*, for example, only made a distinction between scale and outside notes, leaving all scale notes to be equally important. A more detailed distinction could help to achieve a more flexible version of this feature.

5. Final System Development

The findings of the analysis phase after implementing the first prototype have shown that some of the presumptions applied in the design were correct. Yet, they also uncovered an inappropriate behaviour of the system, which was based on two aspects: on the one hand, the system of user levels used was far too rough to accommodate for all important differences between the user groups. On the other hand, the features used so far were far too inflexible to offer the desired adjustability. In the upcoming *DIA cycle*, this behaviour needed to be corrected.

Therefore, the decision was made to develop a new user level scheme, which was supposed to be multidimensional in contrast to the last classification: the user should rather be able to set several parameters describing his level of expertise than just choosing from one of four predefined levels. That way, a more detailed, and thus hopefully a more precise estimation of the users' skills should be achievable. To identify proper candidates for these parameters, a look back on the skills to be substituted for by the system was taken. They seemed to be the right choice to derive the level of support from since the system was meant to account for their lack.

As the features had proven to not being adjustable enough, the static mappings implemented in the Max/MSP-based prototype now seemed inappropriate for further deployment. Furthermore, the equal handling of notes to be valid or invalid in the context of a chord was too inflexible itself, and did not reflect their harmonic ranking (as not all scale note sound equally well, and all outside notes not all sound bad). As the back-end framework [Klein, 2005] was implemented to analyse a chord progression and thereby determine probabilities for all notes, the supporting features needed to be altered to benefit from that flexible evaluation of notes.

Further need for enhancements was identified in terms of displaying the song structure: showing only the current chord had proven to be insufficient for musicians with the ability to conduct their own analysis of the structure. So the lead sheet as a whole should be displayed by the upcoming prototype to offer sufficient information. With this depiction, the system would also adopt the common display format for song structures, and thus take on a shape more recognisable to experienced musicians. In addition, this would allow the integration of a time scale in accordance

with the chord progression. Of course, this would lead to a redesign of the user interface to include time scale and lead sheet.

Besides these upcoming advancements, the new prototype would face a couple of new requirements: while the aspect of collaboration was neglected in the last *DIA cycle* to pay attention to the aspect of musical assistance, this cycle was meant to include concepts to allow and facilitate collaborative sessions: a clear structure needed to be defined and enforced by the system without restricting users too much and too obviously. With the prospect of two or more players using the system at the same time, the deployment of the second interface was necessary: the Buchla Lightning II System was introduced into the new prototype. This interface, of course, brought its own requirements and possibilities to be incorporated and taken care of. Finally, the decision was made to enable the system to perform multiple songs taken from the repertoire of jazz standards commonly used by jazz musicians. With this supply of songs, the system was supposed to allow a more varied number of performances, and thus be more attractive to the users.

5.1 Design

Looking at a more flexible evaluation of notes, and a necessary integration of the back-end framework, which itself was written in C++, Max/MSP seemed inappropriate for the further development of the system. Instead, Cocoa, Apple's environment for application development under Mac OS X, was chosen for the new prototype to be build in. This change, of course, involved a shift in the development paradigm: the signal flow metaphor used in Max/MSP was now replaced by object-orientation used in Cocoa, resulting in an architectural rather than patch-based construction of the system.

Additionally, the user interface needed to be redesigned to fit a window-based application, which offered some new possibilities like hiding rarely used functions, a dynamically resizable user interface, etc. Yet, some of the ideas used to create the appearance of the first prototype in Max/MSP needed to be transferred to the Cocoa interface. More relevance was put on simple and consistent design of the user interface.

Furthermore, the new requirements had to be integrated in the user interface and the system's architecture. Since they are partially vital for the main goals of the system (i.e., the collaboration), they could not just be added to the feature and functions used so far. The system needed to be fully redesigned to account for their integration. Figure 5.1 shows the interaction and output of the new prototype.

5.1.1 New Requirements

The new requirements that were added in the second *DIA cycle* all came with new problems but also new possibilities: to incorporate solutions satisfying these requirements, some of the concepts used in the last prototype needed to be adjusted or extended. Yet, new ways on how to meet the initial goals of the system could be discovered, and turned into suitable solutions.

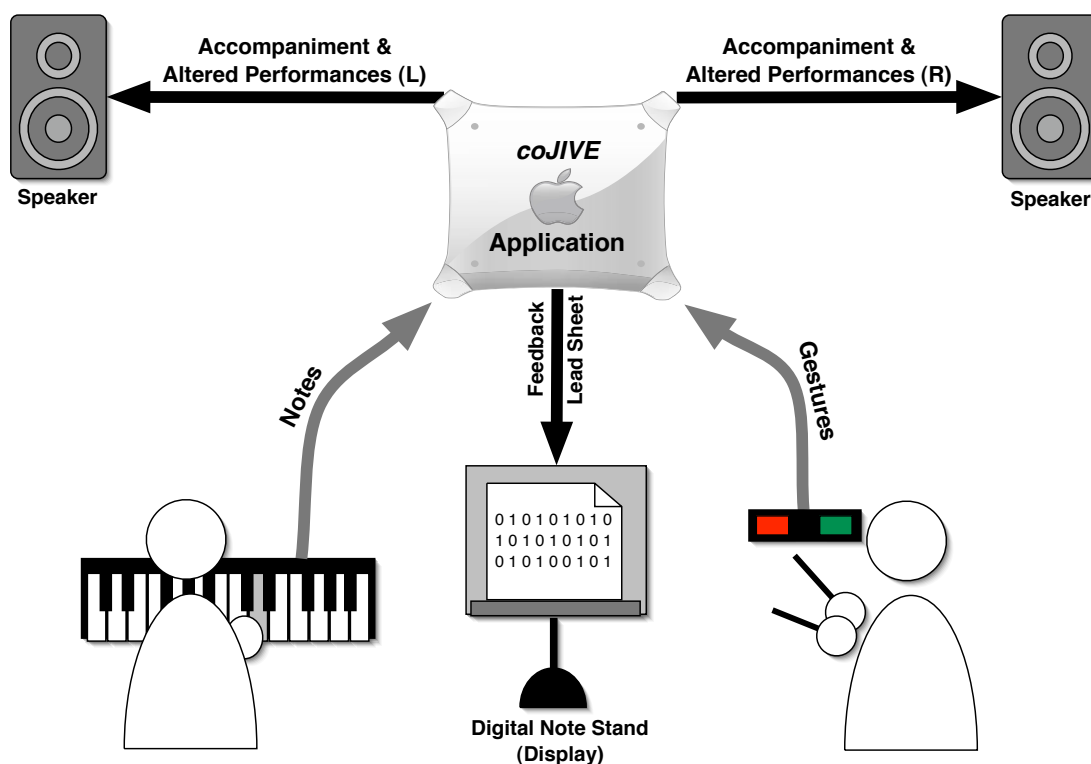


Figure 5.1: The interaction concept of the second prototype.

5.1.1.1 Collaboration

The system's goal to allow collaborative performances was neglected so far in favour of a more thorough treatment of the topic of musical assistance. In the new prototype however, collaboration of several users should be made possible by a fairly simple mode of interaction. Otherwise, novice users needed to be thoroughly instructed, which would contradict the goals of the systems. Therefore, the system needed to communicate the mode to the users.

This mode was found in one of the most common ways of collaboration in jazz sessions: alternating solos. A group of jazz musicians normally starts a song by playing its main melody then taking turns playing solos. In these solos, the soloing musician improvises for a predetermined amount of time (in bars) or until she signals to be finished. In the system, this performance structure needed to be enforced as most of the users could not be trusted to know this concept of collaboration. So, the common duration of a solo was specified to be eight bars ensuring that each player would be able to play several solos, and to check if a solo change could be correctly executed by all potential users.

In terms of assuring the performance structure implied by these solos, two mechanisms were deployed: on the one hand, the graphical user interface was designed to constantly display information on which user was currently soloing. On the other hand, the *restrict expressive parameter* feature was re-implemented with a more flexible behaviour. With this feature, the users accompanying the soloist were detained from playing too loud (and probably disturb the solo). A more thorough discussion of this feature will be presented in subsection 5.1.4.

5.1.1.2 New Musical Interface

The Batons added a new controlling paradigm to the system, that allowed the users to control an instrument with xylophone-like hitting gestures. These infrared sticks do not need any targets to hit, so the system calculated “virtual” targets that were not displayed. So any estimation a user could make as to what notes he played could only be rough. The virtual targets’ positions were calculated to be arranged along the x axis of the batons’ input area to represent the playable notes. Their width was calculated by using the probabilities of the notes they represented to reflect the importance of each note in the context of a chord. For these different widths to take effect, the range of notes accessible, and thus the number of possible targets had to be limited, which would result in a trade-off: if too many targets needed to be fit on the x axis, the targets’ widths would be too small; yet, a very small range would be too limiting in terms of expressivity. A range of two octaves (C4 to B5) was chosen to account for both these aspects. Subsection 5.1.4 describes the calculation process in more detail.

5.1.1.3 Songs

The generic song in the first prototype was sufficient for a test of the deployed features. Yet, the simple song structure did not offer much diversity. For a more varied experience, the new prototype was enabled to load songs instead of having one standard song. That way the repertoire of songs the system could use would be easily extendable. A set of songs taken from common jazz standards was included to test the system’s handling of different songs:

- All the things you are (by Jerome Kern & Oscar Hammerstein II)
- Fly me to the moon (by Bart Howard)
- Giant steps (by John Coltrain)
- Solar (by Miles Davis)
- Stella by starlight (by Victor Young)
- Straight no chaser (by Thelonious Monk)
- There will never be another you (by Warren & Gordon)

Theses songs were noted in the song format described in subsection 5.2.1.1. When a user chooses one of the files, the system hands over its name and path to the back-end for an analysis, and to retrieve information on the song.

5.1.2 Supported User Profiles

As the classification of users used in the last prototype had proven to be too narrow, the question arose how to change it to account for the most important differences between different user groups. Extending the classification to consist of more groups, thereby narrowing down the description a user has to fit if she belongs to one of these groups, would surely account for some of these differences. Yet, this one dimensional evaluation of the users’ skills would be very unlikely to cover all the

aspects that interact to form a person's musical abilities. Accordingly, a multi-dimensional classification needed to be developed and deployed to achieve a more detailed description of the users' abilities, and thus achieve a more appropriate adjustment of the systems' support.

First of all, the dimensions for this new classification had to be defined. These could then be turned into parameters for the new prototype. As the abilities to be assisted or substituted by the system had already been defined, and the features to be adjusted were orientated towards those aspects, they were the logic choice to use for the new dimensions. The next step to take was to define a range of values for each aspect under consideration of two factors: on the one hand, the range should cover the whole scale of values an aspect had to cover. On the other hand, the range should consist of an appropriate number of discrete points so that a user could conduct a fast of the system. The following system was derived:

- **Command of the instrument (instrument skill)**

Important for performing with an instrument is the ability to handle it properly: how to invoke a certain note or how to supply the note with a specific expression (e.g., with a certain velocity or duration) is not only a matter of knowledge but also of thorough training. To estimate this aspect, three different levels were given in this dimension of the system:

1. No experience with playing the instrument
2. Occasional use of the instrument
3. Fluent playing of the instrument

- **Knowledge of music theory (theory skill)**

The knowledge in musical theory also has a great impact on the ability of a person to perform musically. In the context of this work, it was necessary to make a clear distinction between the theory taught in a classic musical education, and the theory used by jazz musicians that is more thorough and includes the classical theory. As the jazz theory can be used to conduct the analysis of chord progressions (implemented by the back-end), its knowledge would imply a more in-depth education of the user. So here again three levels were provided for assign a user to:

1. Hardly any or no knowledge of music theory
2. Obtained a classical musical education or parts of it
3. Knowledge in jazz theory with the ability to read chord symbols

- **Experiences in group performances (collaborative skill)**

This aspect was used as another dimension to enable a prediction of the users' behaviour in a collaborative setting: it was expected that the more experience a user has in playing in groups, the more responsive she would be towards fellow users collaborating in a session. In addition to regular group performances, in which mostly pre-composed songs are played that have been rehearsed several times before, jazz performances needed to be accentuated in this category as they involve unique experiences associated with the communication of changes in a performance's structure (e.g., a solo change). To

incorporate the most important differences, this dimension was provided with the following four values.

1. No experience in group performances
2. Occasional performances with other musicians
3. Regular performances with other musicians
4. Experiences in jazz sessions

The categorisation of the individual dimensions was loosely based on the overall categorisation used in the last prototype (novice – advanced – expert) to offer similar levels in the different contexts. By deploying the different aspects in this new user level scheme, a bigger amount of cases could be covered: for example, self-educated keyboard players, who have no theoretical background could now adjust the system to fit his user level.

5.1.3 Graphical User Interface

The user interface for the new prototype was supposed to offer simple access to the functions of the system. Therefore, it was designed following some of Apple's *Human Interface Design Principles* [Apple Computer Inc., 1996]: different metaphors were deployed to offer representations of information and functionality that most users should be familiar with. The most important functions were placed clearly visible on the interface in order for users to find their way very fast. Less important functions were hidden, and control elements used to access them were clearly placed on the interface. As the system would react to the user's input, feedback was a very necessary tool to inform the player about events. Different modes that would allow access to several functions with one control element were avoided as far as possible to offer a consistent way to control the system. To make the system accessible to a wide variety of people, an English and a German version of the user interface were created. Other language can be easily added thanks to Cocoa's mechanisms for localisation. This section will present an overview over the most important design decisions made while designing the graphical user interface for the new prototype.

5.1.3.1 The Lead Sheet

A central element in the new user interface was the lead sheet: as a metaphor for the paper-based lead sheet that jazz musicians use as a starting point for their performance, it was placed in the centre of the main window occupying most of its space (as it depicts the most information), and all control elements were orientate towards it. No notes needed to be displayed but a visual indication of time had to be added. Accordingly, the chord progression is displayed in a time scale that uses bars and beats as units. The bars' numbers are noted underneath the scale. The current chord symbol and its duration are highlighted by an orange rectangle laid underneath the time scale. The time itself is represented by a red cursor that moved over the scale as the playback is performed to indicate the current point of time in respect to the song's structure. Chord symbols are noted on the lead sheet in accordance with their notation in the song file; a dotted line is deployed for each chord symbol to mark the point on which the change to that chord occurs.

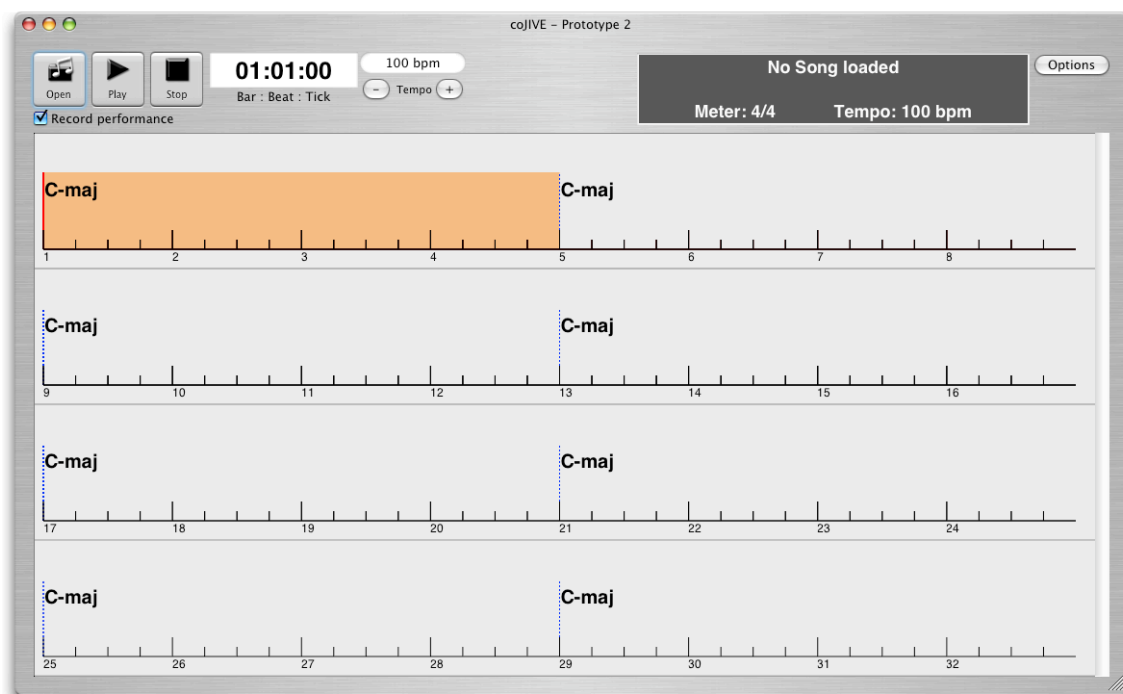


Figure 5.2: The prototype’s main window with lead sheet and controls.

5.1.3.2 Song Controls

The control elements for controlling the playback were designed using the tape machine metaphor, commonly used in applications concerned with the replay of media streams. Beside the buttons implied by that metaphor — play and stop —, a clock display (depicting the current time as *bar : beat : tick*), a tempo display with buttons for increasing or decreasing the tempo, and a display with information on the song (name, default tempo and meter) were deployed to arrange all necessary information on the interface. All these elements were arranged above the lead sheet, grouped in a rectangular area.

5.1.3.3 Player Fields

Each player is provided with an own area on the user interface to allow the displaying of player-adverted information: feedback on performance, information on the player’s current role in the session (playing solo or accompanying), and offering access to the settings for the player (MIDI and user level). Initially, the players’ fields are situated in a drawer attached the bottom of the main window.

To offer feedback similar to the last prototype, a virtual keyboard made of 88 buttons is placed inside the player’s field. This was designed to assure that the input interface of the keyboard is consistent with its graphical representation in the system. As the batons did not offer a visual input field, this musical interface was also provided with a keyboard representation. In contrast to the last prototype, only one keyboard was displayed for input and output to save screen space as several players needed to be provided with feedback this time. This depiction, of course, implied different ways to picture input and output. The buttons used here have a **highlight** function that

shades the whole button; this mechanism was used to depict the output. As the input involves the player pressing a key, a representation of a finger was chosen to mark a key: a pressed key receives an “O” to depict the touch of a fingertip. For the batons, input and output are always the same, as the gestures coming from the infrared sticks are not corrected but mapped.

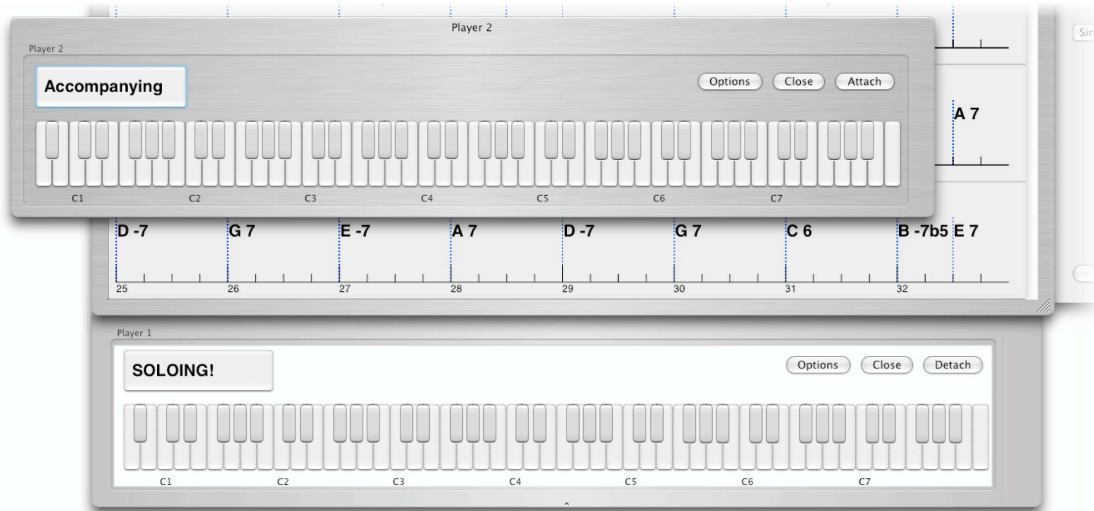


Figure 5.3: The players’ fields on a drawer attached to the main window (lower), and an additional window (upper).

For the information on the player’s current role in the session, the field was extended by another button that was meant to depict an emphasised field. It displays the strings “Soloing” or “Accompanying” in respect to the player’s role. An upcoming solo is announced by a countdown of the beats in the last bar before the beginning of the solo. The field of the soloist is further highlighted by a white rectangle in the background to allow a better orientation.

Some further buttons were integrated in the field: a “Close” button is provided to enable players to leave the system, and free the screen space occupied by her field. Access to the player’s options dialogue is granted via the “Options” button; the dialogue will be described in the next section. Finally, a “Detach” button offers to remove the player’s field from the drawer, and open an own window for it. With this mechanism, the players can arrange their fields on the screen to their liking, and take advantage of, for example, multiple displays. Here, two modes for that button were purposely deployed: originally, it “detaches” the player field to a window. After that, the mode (and labelling) changes to “attach”, reattaching the field to the drawer. These modes were deployed as attaching and detaching stand in relationship with each other, and either one leads to a change of context (window, drawer).

5.1.3.4 Configuration Controls

The decision was reached to hide all controls concerned with configuring the system. Yet, the buttons to access these controls were arranged clearly visible on the user interface. Two dialogues for configuration are used as one was concerned with the

options for a user, and therefore needed to be deployable several times (the number of players to be exact), while the general options of the system only have to be provided in one dialogue.

General options for the system were situated on another drawer that can be opened to one side of the main window. Therefore, another button that opens the drawer if pressed, was placed beside the control elements above the lead sheet. It was avoided to add another mode for that button that would close the drawer that instead was equipped with another button labelled “Close OptionsDrawer” for that purpose. The drawer further contains two pop-up-buttons: if pressed, the first one reveals a list of all possible numbers of players (based on the number of MIDI input ports), and is therefore labelled accordingly. The second is labelled “General Sound Source”, and contains a list of all available MIDI output ports, from which one can be chosen for sending all occurring notes to. Parting lines including names for the different sections were used to visually separate the sections from each other. Figure 5.4 shows a third section called *Accompaniment* that was created to include options on the drum and bass accompaniment (e.g., turning them on or off). This idea however was neglected in favor of spending more time on features more vital to the analysis.

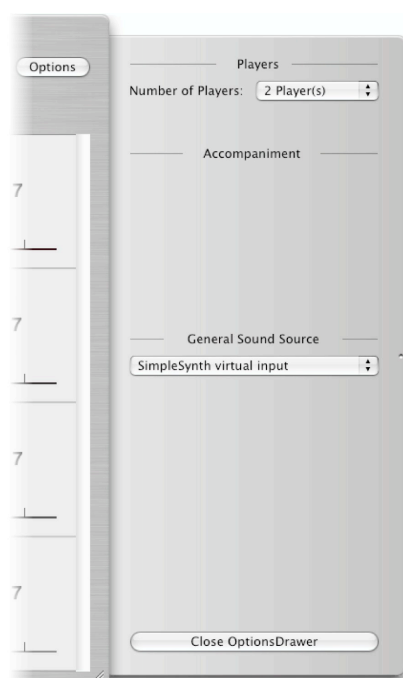


Figure 5.4: The drawer containing the settings for general options.

User-averted options are arranged on a special dialogue window that can be instantiated for each new player. It is accessible through the options button in the player field, and contains control elements for MIDI settings as well as a representation of the user level system for the user to adopt to his skills. In the MIDI section, the user can state which interface (keyboard or batons) he uses, and at what MIDI port it is connected. The section on skill settings is separated into two subsections: a

representation of the user level scheme is provided in the first subsection, each dimension/aspect represented by a field of radio buttons. These radio buttons depicted the levels of the aspect, and are labelled with a first-person description of the abilities summarised by that level. So the user can select the most suitable descriptions for his level of expertise. To offer a facilitation for adjusting these settings, a set of presets were created, and are offered in the second subsection by a pop-up-button.

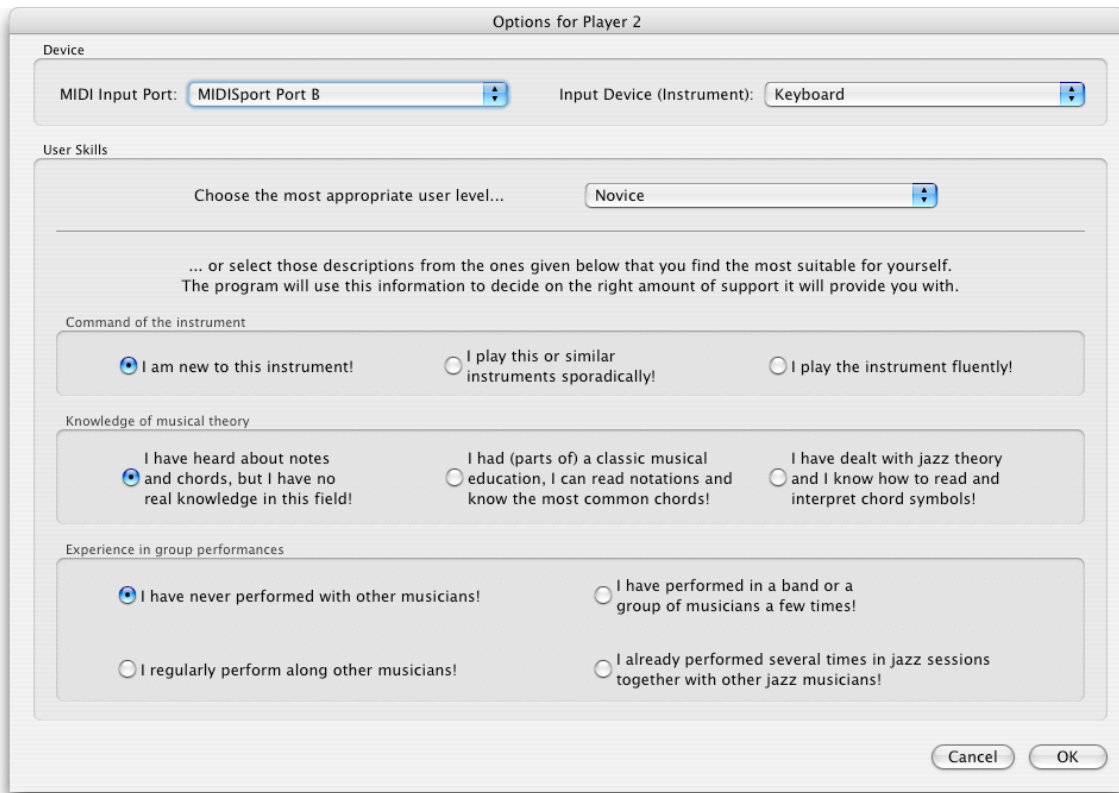


Figure 5.5: The options dialogue for adjusting input and skill settings.

5.1.4 Features

In this prototype the features were not allocated to the different user level; instead, they are configured with the settings a user adjusts in the options dialogue. The focus was shifted from the single features to a reasonable combination of their functionalities. Additionally, the mistakes users made using the keyboard when playing with the last prototype needed somehow to get rid of.

With the new musical interface — the batons —, a more differentiated interpretation of the input was inevitable: the notes from the keyboard still have to be corrected but the gestures coming from the Buchla system are not connected to a note in the first place; a meaningful way to establish this connection needed to be found. The use of the batons did not imply mistakes like the keyboard, and the missing orientation of users left more freedom for the mapping of gestures to notes. To reflect this clear distinction in handling input coming from the musical interfaces, this section will describe the processing of note and gestures separately.

5.1.4.1 Processing a Note

Notes coming from the keyboard are still interpreted in the area of notes that they originally came from. A strong cohesion between the played note, and the system's output needed to be assured particularly as one consistent combination of features was used for all user levels. The following sequence of features is used to process an incoming note:

1. **Filtering out mistakes:**

if a user is a novice in terms of playing the keyboard (*Command of the instrument*, level 0), the system checks if the last key pressed could actually have been played by mistake. Should a neighbouring key have been played within the last 10 milliseconds before, the system discards the last key pressed as being triggered by mistake: it assumes the user accidentally pressed two neighbouring keys with one finger, and does not further process that key (beside giving feedback that it had been pressed). Otherwise, the next step in the sequence of features is started.

2. **Finding the most probable neighbour:**

to check if a note connected to a pressed key is suitable based on the user's level of skill, and the current position in the song, a threshold is calculated that can be compared to the note's probability. The value of this threshold is defined to be about $\frac{1}{12}$ for novices, and decreases with increasing skill of the user. Thus, a novice is allowed to play all notes in an equal distribution of note probabilities. For the determination of the user's level of skill, only the *command of instrument* aspect is observed here.

If the probability of an incoming note is lower than the threshold, the note's neighbourhood is checked for more probable notes: the most probable of these notes in the is picked to substitute for the incoming note. The size of the neighbourhood is again dependent on the level of the *command of the instrument* aspect chosen by the user: for novices, the system looks up the next two notes in each direction, decreasing the size of the neighbourhood with increasing user level.

While the static mappings deployed in the first prototype stuck to the basic ideas presented by Nishimoto et al. [1998] in RhyMe, this feature depicts a mixture of these ideas and the probability-based corrections applied by *ism* [Ishida et al., 2004]. The probabilities still rather arise from the underlying chord structure than from comparing the input to a database of melodies. Yet, the corrections can be flexibly determined based on the current situation.

3. **Checking density of played notes in the neighbourhood:**

pressing too many keys in a small region leads to a chaotic performance and at least some dissonance: it is assumed to be another mistake made by novices as they might not know how to properly use a keyboard. Accordingly, the density of played notes in small note neighbourhoods is artificially limited. So, for an incoming note the two next notes to both side are evaluated by a weight function with decreasing values for increasing distance towards the incoming note. All weights of already playing notes in this neighbourhood are added up; if the sum surpasses a certain threshold, the incoming note is not processed

any further, and no output is created. The threshold is determined using the level selected by the user for the *command of the instrument* aspect, and was chosen to prevent a novices from playing a note if that note's next neighbours (to the left and to the right) are both already playing.

4. Finding a suitable chord voicing:

finally, the new mechanism for triggering chord voicings is deployed. For novices (measured with the *knowledge in musical theory* level), the same functionality already used in the last prototype is provided: if the note calculated by the preceding features has a value smaller than 60 (C5, middle C), the system looks up a voicing of the current chord build on that note (i.e., it has that note as its lowest member). A new level of chord triggering is used for more advanced players: the system searches for a voicing including the current note but only considers it to be valid for triggering if another two notes from that voicing are already sounding. So an advanced player can trigger a certain voicing with at least three suitable notes. This mechanism bridges the gap between classical players using chords with three notes, and the world of jazz piano playing that comprises at least four notes per chord. This mechanism is not limited to the lower half of the claviature: as it involves a more complex method to trigger chords, it is deployed for all octaves to offer more varied ways to use chords to advanced players while allowing them to play freely over all notes as well.

5.1.4.2 Processing a Gesture

In terms of the batons, the system recognises hitting gestures. Some efforts were made to also incorporate sweeping gestures (a downward hit followed by sideward movement) that would allow to emulate a beater sliding up or down a xylophone. But this extension resulted in additional (unwanted) notes invoked by normal hitting gestures. The gestures are reported with a corresponding position on the x axis (0 to 127). This position needs to be interpreted in the context of two octaves to map the gesture to a note.

To achieve this, the probability space is mapped to the position range of the batons. Since the back-end normalises the probabilities for one octave to add up to 1.0, the probability space used here comprises the range between 0 and 2.0. To determine the note belonging to the gesture, the note probabilities are first converted to the position range with the formula

$$width = (p(i)/2) * 128$$

to receive the width of the virtual target representing note i . These widths are then added up (which would correspond to lining up the virtual targets) until the sum surpasses the position of the gesture. In that case, the last width belongs to the target that was hit. The note connected to that target can then be derived and played.

5.1.4.3 Processing Velocity

As the collaboration between players was now an integral part of the system, a mechanism for the support of a session's structure was necessary. In order to point

out a soloist's leading role, the accompanying players needed somehow to be deterred. The idea of *restricting expressive parameters* developed but not used in the first prototype was taken up again, and deployed in this version of the system.

To complicate playing loud while accompanying a soloist, the velocity curve is bend: usually, if a note/gesture arrives with a velocity of i , the system's output is played with the same velocity. But while accompanying, the velocity is recalculated with the formula

$$newVelocity = \left(\frac{velocity}{128}\right)^{bendingFactor} * 128.$$

The bending factor is initially set to a value corresponding to the user's level in terms of the *experience in group performance* aspect, and further on determined based on the *temporal density* of the notes played: this parameter is recalculated whenever a note or a gesture is triggered by the user by adding 1 to represent the new event, and subtracting $\frac{time\ since\ last\ note}{150}$ (the value 150 was approximated by experimenting to receive useful results) to acknowledge the interval between the last two notes/gestures; it is reset to zero if the time since the last note surpassed 200 milliseconds. The new temporal density is compared to a threshold and if it is higher (i.e., the user had played "too much"), the velocity bending factor is doubled to complicate playing loudly even further. The threshold itself was derived from the user's level in terms of *experience in group performance*.

5.1.5 Architectural Design

With Objective-C and Cocoa, an object-oriented environment was employed for the development of the new prototype. This prerequisite was used to design a system structure that can preferably be easily understood, extended, enhanced, and maintained. For that purpose, several steps were taken: the architecture was divided into several layers to provide a rough structure. The relationships of classes were often designed using several well known *architectural design patterns* as defined by the *GANG of four* [Gamma et al., 1995]. These patterns describe reusable solutions to reoccurring problems in software development in form of micro-architectures (a few classes with defined roles and relationships) and offer a common knowledgebase to describe and understand object-oriented software architectures. This section will describe the different aspects of this architecture; simplified UML diagrams are used to depict the relationships between the different classes. More thorough diagrams were collected in appendix C. Figure C.1 shows an overview of the architecture described here.

Five layers were deployed to create a clear distinction between the different parts of the system: the **View**, **Controller** and **Logic** layers reflected the commonly used separation of a program's inner logic and its graphical representation. Classes encapsulating data used in the system were centralised in the **Data** layer to keep a clear distinction between the system's elements processing or displaying the data, and the data itself. Finally, special tasks were integrated in *Singleton* classes (a design pattern defined later on) in the **Singleton** layer. This section will provide descriptions of the inner structure of these layers and their interconnections.

5.1.5.1 System Foundation

To establish a basic structure for the system to build upon, the so-called *Model-View-Controller* pattern (MVC) was used: it encapsulates the application's state, and its

most fundamental functionality in a `Model` class and the user interface including the visualisation in a `View` class. To decouple these elements, a `Controller` class is added to negotiate between them. This results in a flexible combination: the `Model` can be changed, without having to adopt the `View`, and new `Views` can be added while leaving the `Model` unchanged. To keep the `Model` class's functionality independent from the depiction, it does not know of the `Controller` class.

For passing information from the `Model` to the `Controller` nonetheless, a `Controller` needs to register with the `Model` making use of the *Observer* design pattern: the *Observer* object (the `Controller`) deposits a reference to itself with the subject (the `Model`), which uses the reference to report changes of its states. Several *Observers* can register with one subject in this manner. For further de-coupling of the `Model` it uses `NSInvocation` objects to send a predefined set of messages that the `Controller` needs to interpret.

In **coJIVE**, this pattern was deployed two times thereby covering the first three layers: the first MVC is used to be a central base for the main application. The `MainView` is represented by the system's main window (an `NSWindow`) that is connected with a `MainController`. The latter plays the role of the mediator, receiving, formatting, and passing on information on events from `MainView` to `MainModel` or information on changes of the application's state the other way. `MainModel` is responsible for the structure of the performance, communication with the back-end, and controls the playback including the accompaniment. `MainView` is furthermore connected to a `LeadSheetView` object, a subclass of `NSView` depicting the lead sheet with time scale and cursor.

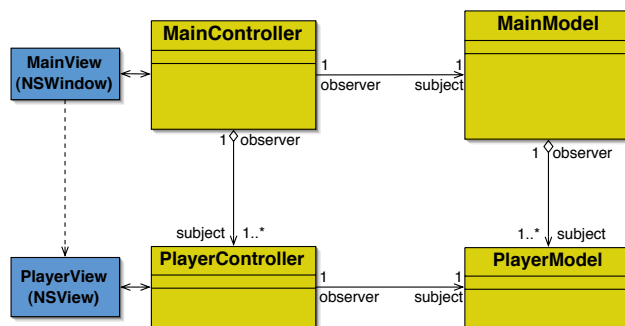


Figure 5.6: The system's foundation and the representation of players using MVCs.

The second MVC represents a player in the system: the `PlayerView` is an `NSView` object containing the player field, while the `PlayerModel` is equipped with mechanisms to receive, interpret, and send notes/gestures (described later on in this section). Again, the `PlayerController` is a mediator between these two classes. Both `PlayerController` and `PlayerModel` objects are created and collected by their main application counterpart (`MainController` and `MainModel`), which also register with them as *Observers* for several purposes: acknowledging a user leaving the system, receiving information on played notes, etc.

5.1.5.2 Centralised Tasks

Some tasks to be met by the system had special attributes: they needed to be executed in a centralised way as several solutions at once would lead to inconsistent

results, and be accessible for several modules in the system. Accordingly, those tasks were implemented using the *Singleton* design pattern that offers a general solution for these requirements: a class implemented as a *Singleton* itself sees to that only one instance can be created, and makes this object available throughout the system. Three tasks were identified that exhibited these attributes: creating a system clock to control the playback, enabling recording of the users' performances in MIDI files, and integrating the back-end into the system. Accordingly, three singletons were created for these tasks.

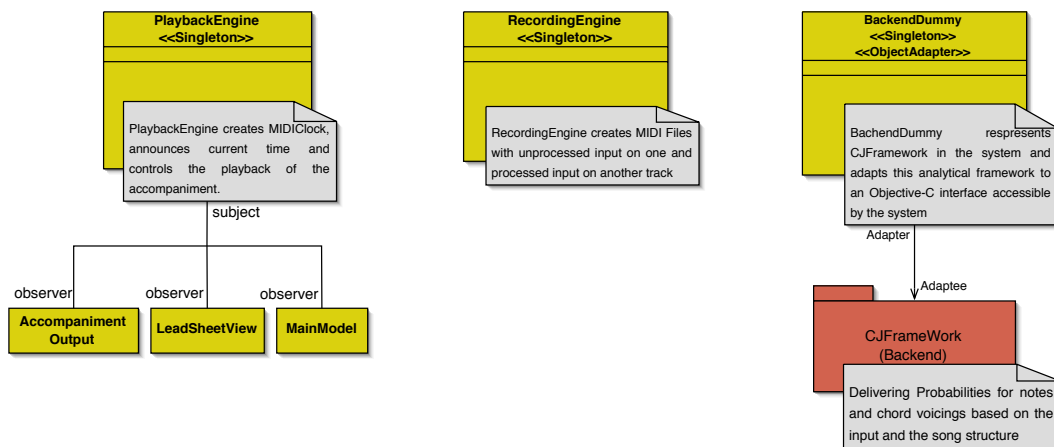


Figure 5.7: The singletons deployed in *coJIVE*.

The system clock was encapsulated in the **PlaybackEngine** class. It generates tick signals in a certain frequency (depending on the appointed tempo) that are counted and transformed into the system's main time format: Bars – beats – ticks, where a beat consists of 48 ticks, and the number of beats in a bar is determined by a song's metre. Bars and beats are counted starting with 1, while the first tick in a beat is 00. Other objects can register as *Observer* with the **PlaybackEngine**, and will then be informed of time changes (i.e., whenever a tick occurs) and whenever a new chord starts. For that purpose, the **PlaybackEngine** holds information on the current song including default tempo, metre, and the song's structure; the classes used to encapsulate this data will be discussed later in this section. The classes observing the **PlaybackEngine** are the **MainModel**, the **LeadSheetView** and the classes concerned with generating the accompaniment (described in the net subsection).

Besides the control of the playback, a mechanism to record the musical data in the MIDI file format has to be provided. Accordingly, the class to implement this mechanism is called **RecordingEngine**. To offer this service to different sources of musical data (e.g., accompaniment, user performances), **RecordingEngine** is deployed as a *Singleton*. The **MainModel** informs the **RecordingEngine** on starting the playback about the number of players in the system, and two tracks for each player are created (to record the original input and the processed output separately). In playback the **MainModel** reports every input event along with its processed version to the **RecordingEngine** that transforms the events into the MIDI format, and collects the obtained data until the playback is stopped. Then, the data can be written to a MIDI file.

Finally, a *Singleton* called `BackendDummy` was created that encapsulates and represents the back-end in the system. This class also uses the *Object Adapter* design pattern: it adapts the back-end's interface, which was written in C++, to an Objective-C-conform interface by instantiating the framework, and forwarding calls to and replies from the back-end. Here, the characteristics of the *Singleton* also reflect the back-end's behaviour towards its clients: the framework does not know its clients, and conducts its analysis independent from the calling client. Respectively, the `BackendDummy` must be called with an additional reference to the calling object for being able to inform the client about the various results. The `MainModel` calls the `BackendDummy` to initiate the analysis or when chord changes occur to receive new note probabilities and chord voicings.

5.1.5.3 MIDI Encapsulation

As several MIDI connections are necessary throughout the system, an inheritance hierarchy to encapsulate the necessary functionality provided by CoreMIDI and add system specific behaviour was created. A common root class for that hierarchy called `MIDIio` was build because CoreMIDI requires similar actions to connect to input and output ports. Derived from that class were `MIDIInput` and `MIDIOutput`, the first of which being abstract yet defining the mechanism to register with an input port, and receive MIDI messages, while the second one — a concrete class — establishes a connection to a MIDI output port, and can send messages to that port.

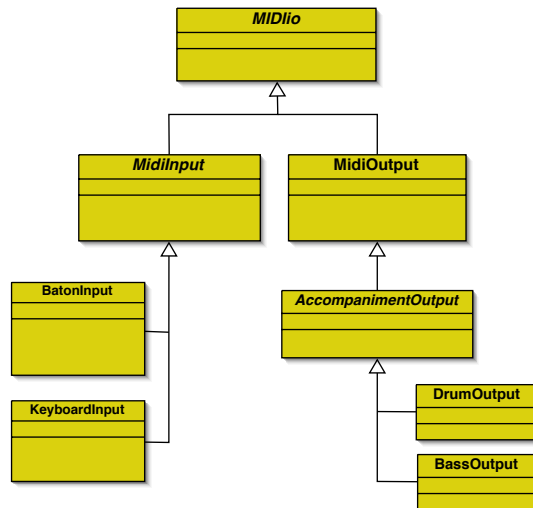


Figure 5.8: The MIDI classes.

`MIDIInput` is used as an abstract class because the input depends on the input device. Accordingly, two subclasses of `MIDIInput` with different behaviours were implemented: `KeyboardInput` and `BatonInput`. A `PlayerModel` object receives an object from one of the two classes for note input using them as instances of `MIDIInput`. `KeyboardInput` provides a very basic behaviour: it just passes on the note value and velocity for each incoming note, and ignores all other MIDI messages. Whereas

the `BatonInput` class encapsulates the gesture recognition using MIDI messages with data on the batons' positions: it waits for the first downwards movement, storing the initial co-ordinates and the time of its occurrence. When the baton move upwards again, the gesture is considered to have ended, and the coordinates are compared to extract gesture size and its duration (both used to calculate the velocity), as well as the position on the x axis. After that, velocity and position are passed on to the `PlayerModel`.

`MIDIOutput` is used for sending out the users' performances: a `PlayerModel` object receives an instance of that class, and sends note values and velocity to that instance, which turns this data into a MIDI message, and sends it to the output port. Furthermore, the class is able to change the channel on which to output, and set the instrument in a MIDI sound set to use. Another application for the output of MIDI notes is the accompaniment. As this performance is generated depending on the time of the playback, an abstract class called `AccompanimentOutput` was derived from `MIDIOutput`: it adds a mechanism to receive time signals from the system clock (by registering as *Observer* with the `PlaybackEngine`), and to store and play a melodic pattern inside a key-value-based `NSDictionary` object (with the time as key and note value as object).

Since the accompaniment consisted of bass and drums, both parts are represented by a subclass adding its specific behaviour: `DrumOutput` sends its output on channel 10, the default drum channel in MIDI, which offers a different element of a drum-set (Snare Drum, Kick Drum, etc.) on each note value instead of one sound in different pitches. Similar to the first prototype, it only uses a simple swing pattern to play. The `BassOutput` sets a bass sound to play with, and uses a simple method to calculate a bass-line: it plays the notes of the first chord voicing transposed to the fourth octave (C3 to B3) in a specific order (lowest note – second lowest note – highest note – second highest note) that way sticking to the current chord.

5.1.5.4 Processing Input Data

The decision was made to encapsulate the features used to interpret gestures and notes in an additional class hierarchy to keep it open to changes, without having to alter other mechanisms. The `InputProcessor` class serves as the root class of this hierarchy, defining almost all of the necessary behaviour. As different chord triggering mechanisms are deployed, one subclass for each mechanism was created: `InputProcessor` itself has no chord mapping for users capable of handling jazz voicings, while `NoviceProcessor` implements the one-key chord triggering, and `AdvancedProcessor` puts the chord triggering based on three keys into practice. A `PlayerModel` object receives one object from this class hierarchy depending on the user's skills thereby completing the chain of input → processing → output.

5.1.5.5 Data Representation

The central piece of data to be represented in the system is a song: its name, default tempo, metre, and structure are encapsulated in a class called `Song` for consistent representation and access. In the system, the `PlaybackEngine` creates a `Song` object on receiving information after a new song was loaded, and the `LeadSheetView` is handed a reference to that object to access the data for displaying.

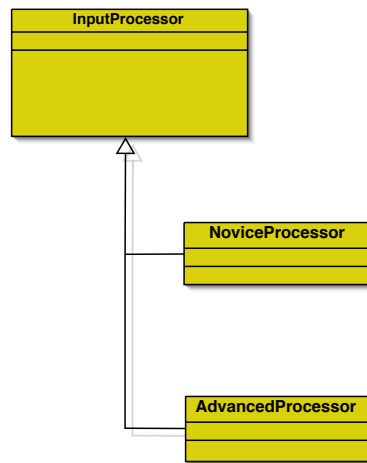


Figure 5.9: The class hierarchy for processing user input.

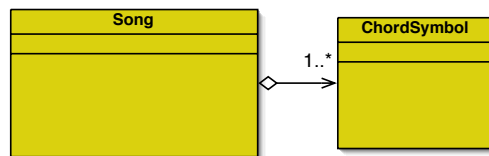


Figure 5.10: Classes encapsulating data.

The structure of the song is described by a progression of chord symbols. As each of these chord symbols has a textual notation as well as a starting point given in the system’s time format (bar – beat – tick), a class called `ChordSymbol` was created to encapsulate these pieces of information for every chord change. A list of such objects representing the sequence of chords in a song is created in the `BackendDummy` using data obtained from the back-end framework. This list is then handed to the `PlaybackEngine` to be included in the `Song` object.

5.2 Implementation

With the shift to Cocoa, several of the components that Max/MSP had provided needed to be newly implemented: besides creating a new user interface, MIDI connections needed to be established using the CoreMIDI [Apple Computer Inc., 2001a, pages 75–106] library, provided by Max OS X. Additionally, new hardware and the functionality of the back-end had to be accounted for in the new implementation.

This section will first present the aspects making up the new environment for the second prototype. After that, the particularities implied by Cocoa and the Objective-C language used in Cocoa as well as how they were dealt with in the implementation are presented. A description of implementation of the most relevant parts of the system makes up the rest of this section.

5.2.1 Environment

The new circumstances led to a change in the development environment and the extension of the system with new hardware. This section will describe the changes and additions made to extend the system.

5.2.1.1 The Back-end

The analytical back-end was developed in the form of an Mac OS X framework in C++ to be easily included in applications. Its goals were to analyse the chord progression defining a song structure, and calculate the probabilities of notes in one octave (so 12 values are calculated) for each chord based on all scales possible over that chord. Furthermore, it maintained a database of chord voicings that could be queried by a client.

To enable such clients to use the framework, they need to instantiate the framework, and can then use a public interface to access the framework's functionality: it can be told to load a specific song file, triggering the analysis by doing so. After that, the song can be accessed for information on its contents (chord symbols, tempo, etc.). The client can also tell the framework the number of a chord, and then query it for the probability distribution of notes and chord voicings for that chord.

A new song format using XML was created to represent and store songs to be later on loaded by the framework. Such files representing songs contain the name, tempo, and metre of the song, as well as the progression of chords, each stored as a textual notation. The framework can read these files, and extract all the different pieces of information. A detailed description of the framework was compiled by Klein [2005].

5.2.1.2 Hardware

The Batons

The most important new piece of hardware deployed in this prototype was the Buchla [1995] Lightning II system, consisting of the two batons, an infrared sensor, and a controller unit. This device can be programmed to recognize different gestures (e.g., sweeps, hits, curves), and transmit events and all available data via MIDI messages. In the prototype at hand, the internal gesture recognition was not used. Instead, the system uses the data on the batons' position to recognize the gestures itself. This data is transmitted by the device in different *ControlChange* messages.

Viscount Viva76 Stage Piano

To further approach the clavature-based interface used in real jazz sessions — the piano — another keyboard was deployed with this prototype in the user tests: the Viscount [2003] Viva76 Stage Piano. It has 76 half-weighted keys, and therefore offers a wider access to the available notes. Equipped with MIDI input and output ports, it can be easily integrated into a MIDI setup.

5.2.1.3 Cocoa

Cocoa [Apple Computer Inc., 2001b] is Apple's newest derivative of NeXTSTEP, an object-oriented development environment developed by NeXT Computer. It provides a wide variety of classes to use to in development: a basic class *NSObject*, from which all other classes are derived, offers all the basic behaviour, a Cocoa object needs: it has a unique ID, functions to query the object for the existence of a

specific function, functions to invoke a specific function, etc. Moreover, Cocoa offers several mechanisms for message sending, for example, the *NSInvocation* class that needs to be configured with a target object, the name of the function to be invoked, and an *NSDictionary* object including all parameters.

Furthermore, numerous container classes and additional helper classes facilitate data management in applications (e.g., *NSArray*, *NSDictionary*, *NSIterator*). Data itself can be encapsulated with special data classes (e.g., *NSData*, *NSString*, *NSNumber*). GUI-elements like buttons and text fields have their representative class (*NSButton* and *NSTextField* respectively) to allow programmatic access and configuration of these elements. All the classes representing GUI-elements are derived from *NSView*, an abstract class concerned with drawing the elements and handling events (e.g., mouse clicks).

Admittedly, the composition of user interfaces (e.g., arranging elements on a window) in Cocoa is usually done using the InterfaceBuilder, a tool that provides palettes with the most common GUI-elements, and allows arrangement of those elements on a window by just dragging them to the desired place. The obtained window is then stored in a *nib*-file from which it can later on be restored at runtime. This mechanism was also deployed to create the user interface for the new prototype. If an element (or the window itself) needs to be accessible in the program, the respective class must contain an instance variable of the element's representative class marked by the attribute `IBOutlet`. These variables can be recognised by InterfaceBuilder, and may be connected to the object used in the tool. Similarly, events occurring on an element can be connected to a function that needs to be declared with the attribute `IBAction` for that purpose. If the event occurs in the running system, the connected function is called.

To simplify the development, an integrated development environment was used: XCode is such a tool provided by Apple Computer Inc. [2003] to use when creating Mac-based projects. Besides common features like syntax highlighting and code completion, XCode offers mechanisms for easy integration of resources (e.g., images), libraries, and frameworks. A debugger is also provided with several functions to locate and remove errors.

The development in Cocoa can be conducted in Objective-C(++) and Java. For the reasons of performance and to enable a trouble-free integration of the back-end framework, Objective-C was chosen for the development of the next **coJIVE** prototype. A special feature of Objective-C is the reference counting: an object is maintained as long as some other objects retain references to that object. This mechanism facilitates dynamic memory allocation and release. Furthermore, the methods and functions declared in Objective-C are mostly described using a `selector`; this string contains of the method's name by leaving out the types and names of the variables, and indicating slots to fill in variables with colons (e.g., `performSelector:` indicates one variable in the method's signature). This syntax will also be used in this document.

5.2.1.4 CoreMIDI

In Mac OS X, MIDI functionality is a part of the overall audio architecture. To access the MIDI services provided by the MIDI server of the operating system,

an API was created in form of a framework, CoreMIDI. This framework defines a hierarchy of representations: a device represents an actual device connected to the system (e.g., MIDI controller, synthesizer), which can consist of one or several entities (the connection ports). Entities themselves can have several endpoints that are either sources of or destination for MIDI messages to enable communication in either direction.

Applications can connect to an endpoint and register a client reference with the MIDI server and a port either to listen to incoming messages (by specifying a callback function) or to send messages. With these mechanisms, a program is able to route MIDI data freely between different ports or just record or generate messages.

5.2.1.5 SimpleSynth

This Mac OS X application by Yandell [1995] is a software synth with a sound set adhering the MIDI standard. It uses the Quicktime synth provided by Mac OS ,and can be addressed by programs just like any other MIDI port via its *SimpleSynth virtual input*. The input port can also be adjusted to every input port present in the system. SimpleSynth is free-ware, and was developed by Peter Yandell. In the second prototype it was used as a general sound source for drums, bass and the user performances.

5.2.2 Implementation Principles

The system was implemented taking advantage of some of Objective-C's and Cocoa's special features: for data storage, `NSDictionary` and `NSArray` are used if possible. In relationships between *Observers* and their subjects, message passing mechanisms are deployed to remove the need for further knowledge of the *Observers'* classes: if several *Observers* are expected, `NSInvocation` objects are used, for one *Observer* its `performSelector:` method is called. Both methods pass a `selector` to the *Observers*, and tell the *Observer* to invoke the function corresponding to the `selector`.

Cocoa also offers mechanisms and classes for multithreading in applications: the `NSThread` class can be instantiated or used to create a new thread by calling the class function `detachNewThreadSelector:toTarget:withObject:` that creates a new thread automatically, and calls a given method on a certain object in that thread. To prevent critical pieces of code (e.g., reading/writing to a instance variable) from being entered by two threads at the same time, `NSLocks` can be used. These objects need to be locked before and unlocked after the critical code, preventing a fatal situation because they can be locked only once before being unlocked.

The reference counting mechanism provided by Objective-C is deployed to achieve a dynamic memory management: The memory for an object is reserved by calling `alloc`, and the object itself is created by triggering a constructor (usually called `init`). Another object can increase the reference count of that object via the `retain` method (defined by `NSObject`). Analogously, the count is decreased via the `release` method. If the reference count reaches zero, the runtime system automatically sends a message to the corresponding object triggering the `dealloc` method that disposes of the object, and frees the memory.

Nonetheless, Objective-C has some disadvantages in terms of information hiding: functions declared in the main interface of a class are always public and can neither be

declared to be private nor protected. Another special feature provided by Objective-C was used to resolve this problem, the so-called *Categories*. With this feature, the interface can be split up and categorised for different purposes (e.g., partitioning the functions by implemented tasks). Here, a second interface category called “private” was created in the implementation file instead of the header file (thereby hiding it) containing the functions that are meant to be used privately. Unfortunately, this procedure was not possible for allegedly protected functions, as they would also be hidden from the subclasses.

Furthermore, the underlying system allows unknown functions to be called leaving the developer with the task to check these calls: if an object sends a message to another object that is not declared in the second object’s interface, the compiler will merely give a warning stating that the second object might not respond to the message (so even hidden functions may be called and executed by other objects). To prevent serious runtime errors to happen due to this fact, all warnings were retraced rigourously, and their cause was removed. A similar problem arises from the use of the different message-passing mechanisms: `NSInvocation` objects and the `performSelector` method of `NSObject`s both pass `selectors` to an object to be executed. These parameters cannot be checked by the compiler when compiling the system. Here, errors are evaded by using `NSObject`’s `respondToSelector:` method to check if a `selector` is valid in the context of an object before sending it for execution.

A special emphasis was put on the readability of the code. Names of variables and functions were chosen to be significant, and they describe the purpose of the element. This sometimes resulted in rather long names. Furthermore, the naming followed the guidelines given by the numerous Cocoa classes to achieve a consistent style in the code: Names of classes start with a capital letter, while the names of functions and variables start with a word in small letters followed by capitalised words (e.g., `performSelector`, `respondToSelector`).

5.2.3 System Foundation

The `MainController` is automatically instantiated whenever the application is started, and its method `awakeFromNib` is triggered after the main window and the options dialogue have been loaded from the corresponding *nib*-file (whose creation was described in subsection 5.2.1.3). In this method, the `MainModel` and the `LeadSheetView` are created, and various default values are set including a list of MIDI input ports obtained from the `MainModel`. Additionally, the `MainController` implements various methods connected to the control elements that mostly trigger corresponding methods in the `MainModel`: `openPressed`, `playPressed`, `stopPressed`, `numberOfPlayersChanged`, etc. The latter method determines the number of players that need to be created, and instantiates the corresponding `PlayerController` objects and deposits them in an `NSMutableArray`¹.

As the `MainController` registers as *Observer* with the `MainModel` and the `PlayerController` objects, it needed to implement some further methods: a `PlayerController` can trigger `addPlayerView:` and `playerViewRemoved` to place a player

¹An `NSMutableArray` is a version of `NSArray` that can be dynamically resized by adding or removing objects.

field (an `NSView` object) in the drawer attached to the bottom of the main window or remove it again (this way enabling dynamical attaching and detaching of the fields) or `removePlayer:` to commence the removal of the respective player's representation in the system. The `MainModel` uses `setSongName:andMeter:andTime:andTempo:` to set and display information on a newly loaded song as well as `setCurrentTime:` to display the current time.

On instantiation, the `MainModel` creates two `NSMutableArray`s with the names of all MIDI sources and destinations for a later selection in the user interface, a `BassOutput` and `DrumOutput` object, sets various default values (time, etc.), and registers as *Observer* with the `PlaybackEngine`. If it is informed by the `MainController` about the creation of a new player (via `createPlayer`), it instantiates a `PlayerModel` object, registers with it as *Observer*, deposits it in a `NSMutableArray`, and returns it. Messages about control of the playback (whenever `startPlaying` or `stopPlaying` are triggered) are forwarded to the `PlaybackEngine`. If stop is pressed and the playback is not running, the `MainModel` tells the `PlaybackEngine` to reset the time.

Furthermore, the `MainModel` creates and ensures the structure of the session. The number of the current soloist — corresponding to the position of the respective `PlayerController` in the array —, and the bar at which the next solo change will occur are stored in instance variable. Whenever `setBar:Measure:Tick:` is triggered by the `PlaybackEngine`, the model checks whether the current bar is the preceding bar of the next solo change, in which case the `PlayerModel` corresponding to the next soloist is informed with the number of beats to go before the change, or if the time of the solo change is reached already. In the latter case the `PlayerModels` of the current and the next soloist are informed about the change of roles. On every chord change indicated by the `PlaybackEngine`, the `MainModel` informs the `BackendDummy`, and distributes the obtained probabilities and chord voicings — both received in `NSArray`s — among all `PlayerModels`. To allow the recording of performances, the model initially informs the `RecordingEngine` on the number of players taking part when the playback is started, forwards all note events reported by the `PlayerModels` (by triggering `triggeredNote:forNote:asActive:byPlayer:`), and reports the stoppage of the playback to the `RecordingEngine`.

`PlayerController` and `PlayerModel` have a connection similar to that of `MainController` and `MainModel`. The `PlayerController` is connected to the view depicting the player field, and reports changes in the player's options dialogue to the `PlayerModel` by triggering `setInputType:atEndpoint:` and `setInstrumentSkill:`

`andTheorySkill:andCollaborativeSkill:` that adjust the `MIDIOutput` object and create a new `InputProcessor` object based on the theory skill level (corresponding to the setting of the `Knowledge in musical theory` aspect in the options dialogue). The latter object is also fed with new note probabilities and chord voicings whenever `setProbabilities` and `setVoicings` are triggered by the `MainModel`.

When the `MIDIInput` object reports a note or gesture (by triggering `incomingNote:atVelocity:atTime:` or `incomingGestureOfType:atPosition:atVelocity:duration:`), the processing of the input data is commenced (which will be described further later on in this section). For that purpose, a `PlayerModel` has an `NSMutableArray`

Dictionary² to store the note actually played by user (*value*) in connection to the notes resulting from the processing (*keys*) for as long as it is held as well as a set to query, and change this dictionary (`isNoteSounding:`, `addNoteSounding:forNote:` and `removeNoteSounding:`).

A `PlayerModel` also reports a set of events to its corresponding `PlayerController` with the help of `NSInvocation` objects (as the `PlayerController` is its *Observer*): when

`upcomingSoloIn:` is triggered, an invocation triggers the method with the same name in the `PlayerController`, while a call for `setSoloing:` leads to the invocation of `showSoloing:` in the `PlayerController`. The model also reports note events after processing with an `NSInvocation` triggering `triggeredNote:forNote:asActive:byPlayer:` in the `PlayerController` that forwards the event to its own methods `notePlaying:forNote:` or `noteStopped:forNote:`, based on the note event being a *NoteOn* or *NoteOff* event that highlight or de-highlight the respective buttons in the player field.

5.2.4 The Lead Sheet and Data Classes

An instance of the `LeadSheetView` class acts as a canvas, while at the same time deploying the mechanisms to draw the pieces of information necessary to depict the lead sheet. It is created by the `MainController` and positioned inside a `NSScrollView`³ on the main window acting as a frame. The instance observes the `PlaybackEngine` (described in the next subsection) to receive the current time, and obtains a reference to the current `Song` object to access information on the song. To draw the basic grid, the chord symbols and the cursor, the `LeadSheetView` uses `NSBezierPath` objects. These objects describe paths (in a view) that can be assembled from lines and curves with the help of various methods (e.g., `moveToPoint:`, `lineToPoint:`, `curveToPoint:controlPoint1:controlPoint2:`), and then drawn to the lead sheet.

The basic grid is drawn by separating the lead sheet vertically into two to four parts each part depicting a line with four or eight bars, based on the total number of bars in the current song. Bars and beats are indicated by tick marks; their numbers are decided by the metre of the song. With the time of the chord changes stored in the `ChordSymbol` objects collected in the `Song`, the chord progression is drawn using the notation stored in the symbol classes. An orange rectangle highlighting the duration of the current chord is drawn by using the class methods `fillRect:` and `strokeRect:` of the `NSBezierPath` class. The current time is depicted by a red vertical line (the cursor) on the current part of the song.

`Song` and `ChordSymbol` are simple container classes with *setter* and *getter* functions. A `ChordSymbol` object encapsulates the time of the chord change as beat, bar and tick (all `int` variables) as well as a textual notation of the chord symbol in an `NSString` object. An instance of the `Song` class contains the song's name (`NSString`), its tempo, number of bars and chords, and the denominator and numerator making up the song's metre (all `int`) as well as an `NSArray` of the `ChordSymbols` defining

²An `NSMutableDictionary` is a version of `NSDictionary` that can be dynamically resized by adding or removing key-value pairs.

³`NSScrollView` is a subclass of `NSView` meant for contents of arbitrary size that can then be browsed by using the scroll bars.

the song's structure. The latter object can be queried by their index, as a sequential processing of the chords was expected.

5.2.5 Singletons

A *Singleton* class needs to ensure that only one instance can be created by itself. Therefore, constructors are declared to be private, and are only triggered by a static function that can be triggered by other objects, without having an instance to begin with. The single instance is managed by the class in a static instance variable: if the object is requested, this variable is checked to be empty (i.e., if the class has not yet been instantiated). In that case, the constructor is triggered, and the resulting object is stored in the instance variable, and a reference is sent to the enquiring client object.

5.2.5.1 PlaybackEngine

Objects can register as *Observers* with the `PlaybackEngine` by triggering its `addObserver:` method that is parameterised with the calling object's `id` to be stored in an `NSMutableArray`. To receive the notifications produced by the `PlaybackEngine`, the *Observers* need to implement a set of public methods: `setBar:andMeasure:andTick:` to frequently receive the current time, `chordChangedToNumber:` to get updates on which chord is current, `playbackStopped` to be notified when the stop button is pushed, and `songHasChanged` to know if a new song has been loaded into the system. These methods are triggered from the `PlaybackEngine` using one `NSInvocation` object per method.

The system clock was implemented in its own thread: when the play button is pressed, the method `startPlayback` is triggered in the `PlaybackEngine`. This method locks an `NSLock` object — an instance variable of the `PlaybackEngine` —, and invokes a private method called `play` using `detachNewThreadSelector:toTarget:withObject:` from the `NSThread` class to swap out the system clock to a new thread. The `play` method initiates an `NSTimer` object with a delay time corresponding to the current tempo, after which it will frequently trigger the method `nextTick`. This object is then attached to a newly created `NSRunLoop` object, and the `NSLock` is again locked, thus assuring that the thread is going to run until the main thread (in which the lock was initially locked) will unlock the `NSLock`. This was implemented in the method `stopPlayback` that is invoked when the stop button is pressed. After that, the thread running the clock can release the `NSLock`, and invalidate the `NSTimer` — thereby stopping the clock — and stop itself.

The method `nextTick` implements two tasks: initially, it increases the tick count on invocation, and adjusts the time that is stored in three `int` instance variables: `bar`, `beat`, and `tick`. After that, it calls the method `setBar:andMeasure:andTick` to inform the *Observers* of the new time. This method also checks if the time for the next chord change is already reached. If that is the case, the method `chordChange` will be called: here, the chord change is propagated to the *Observers*, and the time of the next chord change is extracted from the `Song` object.

Besides running a system clock, the `PlaybackEngine` also holds the `Song` object encapsulating information on the currently loaded song. On creation, the engine instantiates a default `Song` with a simple structure. When a song file is loaded, the

`BackendDummy` passes the song's name, numerator and denominator of the metre, an `NSArray` with the `ChordSymbols`, and the number of bars in the song through the `MainModel` on to the `PlaybackEngine`. Here, a new `Song` object is created with this set of data, and the *Observers* are informed that a new song had been loaded.

5.2.5.2 RecordingEngine

At the beginning of a performance, the `RecordingEngine` needs to be prepared for recording. Accordingly, the method `prepareRecordingFor:andDenominator:andNumerator:` was supplied that is triggered with the number of players and information on the song's metre (numerator and denominator). This method creates the main header for the MIDI file (by calling `createHeader`) and the header for all tracks (by calling `createTracks`), supplying two tracks per player (for input and output).

The main header and all tracks are kept in separate `char`-buffers. This separation is necessary because the MIDI file format defines that the tracks must be stored successively. The buffer for each track is initialised with 100 kilobytes of memory to offer enough space even for long and excessive performances. Furthermore, a position counter containing the current position in a track and an instance variable containing the time of the last event are created per track. The latter variable is necessary due to the fact that events are noted in the MIDI file format including not the absolute time of their occurrence but the relative time relating to the last event in the same track.

A note event is reported to the engine by triggering the method `noteTriggered:forNote:asActive:byPlayer:atTime` with a list of notes to be played, the original note value triggered by the player, a `bool` value identifying the event as a *NoteOn* (true) or *NoteOff* (false), the number of the accountable player, and the absolute time of the event in ticks. The method first calculates the relative time of the event, decides on a velocity value (100 for *NoteOns* and 0 for *NoteOffs*), and adds the event to the respective tracks: for player i , the original note is stored in track i while the list of notes obtained from processing the original note are stored in track $i+1$. The private method `addData:inTrack:` is used to add each piece of information (relative time of the event, type of the MIDI message, note values, and velocity) serially each time adjusting and checking the position counter of the track as some space is needed at the end of each track for special marks (end of track). If there is only space enough for these marks in one of the tracks, the recording is stopped.

Some data in MIDI files — like the relative time of an event — needs to be stored in a special format, *variable quantity bytes*. These bytes are used as seven-bit entities leaving the last bit to indicate the end of a sequence of these bytes. This allows to encode an arbitrarily-sized piece of data as a progression of these *variable quantity bytes*. To encode data in this format, the method `convertToVariable-Size:toTrack:` implemented a method described by Glatt [2001].

If the recording is somehow stopped — by pushing the stop button or if a track buffer is full —, the method `endRecording` is triggered. It first calls `closeTracks`, a method adding the end marks to all tracks, and then combines the main header

and all tracks in one `NSMutableData` object⁴. This object allows storing its contents in a file, which is done before the buffers are finally freed.

5.2.5.3 BackendDummy

As the `BackendDummy` is meant to reflect the back-end's characteristic to be unaware of its clients, a client (in this prototype the `MainModel` is always this client) needs to deposit an `id` — preferably its own — when querying one of the back-end's services through the `BackendDummy` that the *Singleton* uses to send the results to. Again, this is done via `NSInvocation` objects, expecting the client to implement a set of methods corresponding to the available services.

To load a song into the back-end and obtain information on that song, the `BackendDummy` supplied the method `openSong:forClient:` that needs to be triggered with the name and path of the song file to be loaded. This filename is then passed on to the instance variable of the `CJFramework`, calling the back-end's `load-Song` method. After that, a `CJSong` instance can be requested, from which the necessary information can be extracted. These are fed to an `NSInvocation`, including an `NSArray` with the `ChordSymbol` objects corresponding to the song's structure. The `NSInvocation` object, if invoked, passes this information to the client by triggering the client's `setSongName:andMeterDenominator:andMeterNumerator:andTempo:andChordSymbols:andNumberOfBars:`.

A client can also report a chord change to the `BackendDummy` by triggering `chordChangeNumber:occuredForClient:`. This method passes the number of the new chord to the `CJFramework` thereby triggering the analysis for that particular chord. Then the new note probabilities and chord voicings are passed to the client, both stored in `NSArray` objects. To achieve this, another `NSInvocation` triggers the client's `setVoicings:andProbabilities:` method.

5.2.6 The MIDI Classes

An application willing to connect to a MIDI port in Mac OS X needs to execute a few steps: initially, a reference to an endpoint needs to be received⁵, and a `MIDI-ClientRef` needs to be registered with the CoreMIDI server. Finally, the port to use that again differs for input and output ports needs to be created. The `MIDIio` class defines the basic behaviour for these steps deploying the *Template Method* design pattern that describes the method of an abstract class depicting the skeleton of an algorithm calling other functions to be defined by concrete subclasses. With this mechanism, every subclass can create its own version of the algorithm by redefining the different steps — the different functions called by the *Template Method* — while leaving the course of events in the algorithm unchanged.

Whenever an object from the MIDI class hierarchy — based on `MIDIio` — is created, `initWithEndpoint:` (a *Template Method*) is called with the number of the endpoint to use, which calls `initEndpoint:`. This method itself is a *Template Method* calling `getEndpoint:` (re-implemented by `MIDIInput` and `MIDIOutput` to either

⁴An `NSMutableData` is a version of `NSData` that can be dynamically resized by adding or removing data buffers.

⁵The function to use is dependent on whether the endpoint is a source (for input) or a destination (for output).

use `MIDIGetSource` or `MIDIGetDestination`) to receive the `MIDIEndpointRef` before creating and registering the client reference via `MIDIClientCreate`. After that, `initWithEndpoint:` triggers `initPort` that again is redefined by the classes derived from `MIDIio` to use `MIDIInputPortCreate` or `MIDIOutputPortCreate`. `MIDIInput` thereby registers a callback function — `readProcedure` — that can then be called by the `CoreMIDI` server in case a MIDI message is sent to the respective input port.

If such a message occurs, `readProcedure` receives a `MIDIPacketList` with a set of MIDI messages. This packet list is fragmented into messages by `handleMIDI-Packet:`, each time calling `handleMIDIMessage:onChannel:withData:withData2:atTime:`. This method is redefined by `KeyboardInput`⁶ as well as `BatonInput`⁷ to incorporate the respective behaviour.

For the recognition of gestures, `BatonInput` supplies several instance variables for each baton: two `int` arrays with two elements each (representing x and y coordinates) to store the initial coordinates of a gesture, and the last coordinates of the baton for comparison, a `bool` to indicate if a gesture is in progress, and a `UInt64` to store the start time of the gesture in. Changes in the coordinates of a baton are reported via `ControlChange` MIDI messages.

If no gesture is in progress, the `BatonInput` object waits for the first downward movement by comparing newly reported y coordinates with the one stored last. On the occurrence of this downward movement, a new gesture is declared to be in progress and the current coordinates and the time are stored. When the baton moves upward then next time, the gesture is declared to have ended, the velocity is calculated using the size of the gesture (initial y value of the gesture decreased by the current y value) and the duration (current time decreased by the initial time of the gesture) in the function `calculateVelocityForDistance:startedAt:`, and the gesture is finally reported to the `PlayerModel` deploying the last x value as position.

To enable the triggering of notes in a sound source (here: `SimpleSynth`), the `MIDIOutput` class provides two similar functions as the keyboard provides both `NoteOn` as well as `NoteOff` message, while the gesture recognition only triggers the start of a note. `playNote:atVelocity:` only forwards note value and velocity to `sendMIDIMessage:withData1:withData2:forMillisecs:` with a duration of 0 milliseconds, while `playNote:atVelocity:forMillisecs:` receives a specific duration to forward. `sendMIDIMessage:withData1:withData2:forMillisecs:` uses these pieces of information to create `MIDIPackets` to be sent to the registered port. If a specific duration longer than 0 milliseconds is given, a `NoteOff` packet is created and provided with a time-stamp calculated by adding the duration to the current time. The respective message is stored in the `CoreMIDI` server when sent off, and is forwarded to the port when the time stated by the time-stamp is reached (thus delaying the `NoteOff` for the given duration).

The `AccompanimentOutput` class adds the behaviour necessary for calculating and automatically playing notes: on creation an object registers as `Observer` with the `PlaybackEngine` so that `setBar:Measure:Tick:` is triggered whenever a new tick is produced by the system clock, and the object can check whether the `NSDictionary`

⁶In `KeyboardInput`, this method forwards `NoteOn` and `NoteOff` messages to the `PlayerModel`.

⁷In `BatonInput`, the redefined method triggers the gesture recognition and informing the `PlayerModel` on gestures.

containing the current melodic pattern includes one or several notes stored in an `NSArray` (*value*) for the current tick in relation to the current bar (*key*). If this is the case, each note is sent with a specific duration and a determined velocity using the functions inherited from `MIDIOutput`. The duration is calculated whenever is triggered by the `PlaybackEngine` to account for changes in tempo. It is set to the duration of a beat at the current playback speed to allow a fluid performance of the walking bass (so each notes passes into the next).

5.2.7 Input Processing

The processing of the input data is triggered whenever a note or gesture is reported to a `PlayerModel` object. This model then hands over the incoming data to the current `InputProcessor` object for processing (is further described below), reports all resulting notes to its `MIDIOutput` object to trigger them, and then reports the whole event (including original and resulting notes as well as the processed velocity) to its *Observers* for recording, displaying, etc.

5.2.7.1 Note Processing

When `incomingNote:atVelocity:atTime:` is triggered in the `PlayerModel`, this model calls the function `processNote:` of the corresponding `InputProcessor`. This function implements the algorithm described in subsection 5.1.4.1 as a *Template Method*. Each step of the algorithm is implemented in an own function, leaving the last step — the determination of a chord voicing — to be re-implemented by `InputProcessor`'s subclasses, since it is the only varying step. `processNote:` returns an `NSMutableArray` with the notes to be played.

`isError:` is triggered first by `processNote:` with the note value of the originally played note. It returns a `bool` value stating whether the note is assumed to be played by mistake (i.e., by accidentally hitting two keys with one finger) or on purpose, in the first case cancelling the algorithm causing it to return an empty array. For the determination, the notes are classified by the keys representing them on the clavature⁸ as their positions cause these unwanted occurrences: if a black key is situated next to a white key, this white key can either be unintentionally be pressed by slipping from the black key or by slightly missing the centre of the next white key (beyond the black key), and pressing both white keys with one finger.

The `InputProcessor` furthermore holds an array of 128 `UInt64` variables representing the time of the last invocation of each of the 128 notes (as defined by the MIDI protocol). For the incoming note, the notes that could be involved in its unintentional triggering are identified, and the times of their last invocation is compared with the current time. If one of the time differences is smaller than 10 milliseconds, the incoming note is declared to be played by mistake. If the whole algorithm is not cancelled by this or one of the other functions, the current time is noted in the array of invocation times for the incoming note value.

The next step is implemented by the function called `findMostProbableFor:`. If the probability of the incoming note falls below a threshold (based on the instrument skill), this function compiles a list — an `NSMutableArray` to be exact — of all

⁸Three classes of keys are defined: black keys, white keys with only one black key to the left/right and white keys surrounded by two black keys.

notes in the neighbourhood of the incoming note sorted by descending note probabilities. The size of the neighbourhood differs based on the instrument skill set by the user (corresponding to the *Command of the instrument* aspect in the options dialogue). Beginning at the top of that list (with the most probable note in that neighbourhood), `findMostProbableFor`: checks for each note if it is already playing by triggering `isNoteSounding`: in the `MainModel` until a note is found that is not sounding already. This note is declared to be the most probable note for the incoming note and returned for further processing by the algorithm.

`neighbourhoodTooDenseFor`:, implementing the next step, checks whether too much notes in the neighbourhood of the note determined in the last step are already playing, and cancels the algorithm if the result is positive. For that purpose, a weight for each note in the neighbourhood⁹ is calculated as

$$weight(i) = |1/(incoming\ note - i)|$$

using the distance of i to the incoming note. For all notes in the neighbourhood that are already playing, the weights are added up and compared to a threshold that is determined based on the instrument skill. If the result surpasses the threshold, the density of playing notes in the neighbourhood is declared to be too high. The threshold was configured to assure for novices that if the two neighbouring notes of the incoming note are already playing, the density is deemed to be too high, while experienced users are not affected by this function. A `bool` is returned stating whether the density is too high, cancel the rest of the algorithm if the value is `true` (also-called *YES* in Objective-C).

The last step in the algorithm is the determination of a suitable chord voicing for the note determined by the preceding part of the algorithm. For that purpose, `InputProcessor` declares the function `findSuitableChordFor`: that returns an `NSMutableArray`, and is implemented by the `NoviceProcessor` and the `AdvancedProcessor`. The first of these derived classes deploys the one-key chord triggering mechanism similar to the one deployed in the first prototype. Here, `findSuitableChordFor`: checks the array of voicings delivered through the `PlayerModel` from the `MainModel` on each chord change for a chord with the incoming note as its lowest member (thereby following the selection criteria used in the first prototype to manually determine chord voicings for the static mappings).

The version of `findSuitableChordFor`: implemented by the `AdvancedProcessor` class supplies the algorithm with the new three-key chord triggering mechanism. It checks the chord voicings for a suitable candidate by inspecting whether a voicing contains the incoming note. This process involves checking it for comprising the incoming note disconnected from its octave (leaving the value between 0 and 11) or the same value increased by one octave (between 12 and 23) as the chord voicings used can span over two octaves. If this should be the case, the other members of the octave are brought to the respective octave, and it is checked if they are already sounding with `isNoteSounding`: in the `PlayerModel`. The chord voicing is declared to be suitable if the number of sounding members surpasses 1, which means that two of its notes in the respective octave are already sounding extended by the note currently being processed (resulting in three sounding notes from that voicing).

⁹The neighbourhood includes two notes to either side of the incoming note.

Since the notes in the chord voicings are stored independent of the octave to use (i.e., as the values 0 to 11, and if a voicings spans over two octaves, the values 12 to 23), the incoming note needs to be disconnected from its octave (bringing it to the range between 0 to 11), and the resulting voicing needs to be transposed to this octave (by adding $12 * octave$ to each chord note) in both versions of the function.

In addition to the steps described in the algorithm, `processNote` also triggers `recalculateTemporalDensity` that is used to rate the number of notes played per time whenever the user is accompanying and not soloing. It calculates the so-called `velocityBendingFactor` that is used in the processing of velocity values (described later on in this section), based on this rating. The `temporalDensity` itself is calculated as described in subsection 5.1.4.3, and if its value surpasses a threshold based on the collaborative skill (corresponding to the *Experience in group performance* aspect in the options dialogue), the value of the `velocityBendingFactor` is doubled.

The `PlayerModel` first stores the result in the `NSMutableDictionary` thereby allowing the retracing and removal of the notes processed based on the original note as soon as the corresponding *NoteOff* arrives. After that, the `MIDIOutput` to send of the corresponding MIDI message for each of the newly determined notes, and the *Observers* (`MainModel` and `PlayerController`) are informed of the new note event. The sequence diagram in figure C.2 (Appendix C) depicts the flow of events during the processing of a note.

5.2.7.2 Gesture Processing

The processing of a gesture is started by the triggering of `incomingGestureOfType:atPosition:atVelocity:duration:` in the `PlayerModel` that in turn calls `processGesture:` in the `InputProcessor`. This function implements the procedure described in subsection 5.1.4.2 with a slight variation: in order not to lose the precision of the note probabilities stored in `float`-variables, the input area of the batons is mapped to the probability space of the two octaves, and not the other way around. Yet, only a probability space covering one octave is used to speed up the calculation, and the resulting note is transposed up one octave if the position of the gesture should be > 63 . To calculate the note, the position (subtracted by 64 in the case just described) is brought to the probability space by dividing it by 64. Now, the probabilities for the notes in the octave are added up one after another, each time checking whether the value calculated from the gesture's position is surpassed, in which case the note whose probability was added last is the note to be triggered. Figure C.3 (Appendix C) shows the course of events necessary for the whole process in detail.

5.2.7.3 Velocity Processing

The processing of the velocity of incoming events is not dependent on the type of event; both notes and gestures are treated equally in that respect. So, in both `incomingNote:atVelocity:atTime:` and `incomingGestureOfType:atPosition:atVelocity:duration:` of the `PlayerModel` object the `InputProcessor`'s function `processVelocity:` is triggered. In this function, the velocity is first divided by 128 to receive a `float` value between 0 and 1. This value is then raised to the power of the `velocityBendingFactor` previously calculated `recalculateTemporalDensity:`, thereby decreasing it if this factor is larger than 1. The result is multiplied

by 128, and rounded to obtain the corresponding velocity value that finally is returned by the function.

5.3 Analysis

Similar to the analysis phase of the last *DIA cycle*, a set of user tests was conducted to allow an evaluation of the system. Again, the users were asked to play with the system with no imposed time-limit. Yet, the users were asked to use the system in pairs to account for the newly introduced concept of collaboration.

5.3.1 User Tests

For this prototype, a more thorough user study was conducted with a total of 20 subjects. Four of these subjects tested the system on their own, which was mostly due to problems in the schedule, while the other 16 were grouped in pairs to cooperate in using the prototype.

The group of subjects was this time composed of a wider spectrum of people. As the level of support the system provides was adjusted to the users' skills, it had to be checked if that support was suitable for all the different user groups. Three of the subject had actual experience in jazz sessions, one with playing the trumpet, another one using a guitar, and the last one playing piano. Additionally, two people with some classical education in playing the keyboard or piano tested the system. The rest of the subjects consisted total novices in the field of music and some hobby musicians.

Again, questionnaires were handed to the subject right after the tests. Since the supporting mechanisms of the system were more sophisticated this time, a more thorough enquiry was necessary this time. The collaboration aspect needed to be covered as well as the appropriateness of the support by the system.

5.3.1.1 Procedure

At the beginning of a test, the subjects were asked to settle what instrument each subject would play. Some of the subjects were eager to try both instruments, which was not prevented, but to keep the tests within a certain time the pairs were only asked to perform in one setup. After that, they were asked to decide on a song to play. Since most of the subjects were not familiar with the songs provided by the system, the choice was usually made at random. Finally, they had to adjust the skill settings according to their own ability.

Before starting the performance, the subjects were asked to watch the prototype closely while playing. They were not explicitly told about the collaboration mechanisms to test how these mechanisms would affect the performance. The duration of the performance could freely be chosen by the subjects, which again was a matter of coordination between the two subjects. Single user tests were conducted very similar, as the activity was not very different in that case, and the collaboration mechanisms were not active for one person only. Similar to the last user tests, the subjects received no further information on the song they chose like a score.

5.3.1.2 Questionnaires

The questionnaires were divided into four sections starting with the enquiry of personal data. Here, the users were asked to name their main instrument (if any), how long they have been playing it, what other instruments they play occasionally, and the duration of their musical education so far. The second section asked for general information on the test like the song performed in the test, what instrument the subject had played, and if he had played that song before. Furthermore, the subjects were asked which skill settings they had used in the system to check whether they would make consistent statements on their abilities before and after the test (for that purpose, the settings in the system were noted at the beginning of the test).

The third section comprised statements on the test process with a consistent rating scheme. The subjects could rate each description by choosing one of five levels between “always applicable” and “never applicable”. Some of these sentences were related to question in the questionnaire covering the tests of the last prototype since similar or same aspects had to be checked again. The requirements and aspects of the new prototype had to be covered as well, and therefore, a set of additional aspects were covered by other statements. The first set of descriptions was meant to cover the subject’s conduct during the test, as well as their perception of the system’s reaction to the users’ input, and how it affected their performance:

1. **I followed the chord symbols during the performance.**
2. **I followed my own performance on the screen.**
3. **This feedback did help me in performing.**
4. **I perceived to play something suitable.**
5. **I perceived to have control over my performance.**
6. **The system clearly enhanced my input.**
7. **It was obvious how the system supported me.**
8. **The system corrected “wrong” notes in my input.**
9. **I would need more assistance to find nice melodies.**
10. **The assistance to play chords was really helpful for me.**
11. **The system has lowered my inhibition threshold to try new things.**
12. **The system has help me to enjoy the performance (even more).**

The subsequent set of statements was added to evaluate the collaborative aspect of the system. The subject should rate their behaviour towards each other, as well as how they co-operated and the systems support in that context.

13. **My partner and I played well together.**
14. **The system helped us with that by systematically deploying support.**

15. **It was obvious how the system structured the session.**
16. **I noticed in time if my solo was going to start.**
17. **I noticed in time if my solo was going to end.**
18. **My partner did not interfere with my solos too often.**
19. **I did not interfere with my partner's solo too often.**

Finally, two statements were appended to obtain the subjects' overall impression of the system:

20. **The system has sufficiently supported my performance.**
21. **The system has limited my performance too much.**

The questionnaire was concluded — similar to the last questionnaire — with a section asking for personal remarks, opinions, and suggestions for improvements by the subject. In contrast to the questionnaire used in the last cycle, this aspect was this time presented in an own section than just being one point in a list of questions. This classification was due to the fact that it had proven to be a beneficial source of further information and useful proposals leading to some of the design decision implemented in this prototype. Again, no time-limit was imposed on filling out the questionnaires.

5.3.2 Results

Some problems of the system became apparent very quickly during the tests just from observing the subjects: while subjects belonging to the group of absolute novices seemed to be pleased with the amount of support, especially intermediate users often had a hard time coping with assistance. The experienced keyboard players, for example, were very irritated by the fact that the system would sometimes play a different note than what they originally played.

It was also obvious that the deployed tuning of the musical assistance with the help of the skill factors did not cover all the cases present in the group of subjects. The jazz trumpet player, for instance, had thorough knowledge of jazz theory, and thus knew what she wanted to play. Yet, she only played the piano occasionally causing the system to correct some of her input, which led to her complaining about the prototype's behaviour. In Addition, some subjects stated that the levels for the skills did not include their special case by pointing out that they would rate themselves in-between two of the levels of an aspect in the options dialogue.

There were also problems concerned with the velocity: On the one hand, some of the novices were not used to the half weighted keys on the keyboard deployed for the tests. They had a hard time playing loud, and it often took them some time to figure out that more impact on a key would result in a louder note. On the other hand, the bending of the velocity curve for accompanying users was initially meant to detain them from playing too loud conveying them to perform more subtly. Yet, this mechanism often led to the opposite behaviour as some user tried even harder to make their performance heard once they noticed it to grow more mute.

This effect was intensified even more by two other problems of the prototype: On the one hand, the display of the users' roles and the announcement of upcoming solos were often overlooked. Some of the subjects would just play on independent of the display. These circumstances were very bothersome especially for the subject playing the batons as the keyboard was explicitly louder. On the other hand, the gesture recognition deployed for the batons did not work reliably. All subjects had a hard time to play a sequence of notes with one baton, and the condition even worsened when using the second baton in addition. Additionally, the feedback for the batons — also on a virtual keyboard — seemed to afford for most subject's to try hitting specific notes that sometime distracted them from creating melodies.

The second mechanism for chord triggering — using three keys — was not as accepted by the subjects as expected; often they did not notice its application in the performance, and some subjects did not even understand the statement on this feature in the questionnaire. Nevertheless, one of the subject explicitly complained about the mechanism being applied all over the claviature. He mentioned that even when trying to play a melody with his right hand the system would sometimes add notes to form chords.

The length of the solo seemed to be problem for most subjects. The jazz musicians explained that eight bars are inappropriate to develop a melody in especially as the time grows shorter in faster songs, while some of the novices seemed to be in a hurry just to keep up with the structure of the session. The later fact was also due to the announcements starting only one bar ahead of the next solo.

Besides these problems, the tests also showed improvements in the new prototype: the mechanisms to filter out mistakes (the unintentional pressing of a key and pressing too many neighbouring keys at the same time) were not discovered by any of the subjects, yet, the recordings of the performances show that they were applied in numerous performances. It can be argued that this fact proves the assumptions leading to the development of these mechanisms. Some of the subjects who had also attended the first user study expressed their opinion that the new prototype was a big improvement over the first one, as its behaviour proved to be more flexible (as intended by the new design). They stated to have had a much more satisfying experience compared to the first round of tests. The new user interface was mentioned to be concise, and the depiction of the lead sheet seemed satisfactory to all subjects since experienced users could get enough information, while novices often ignored the chord symbols.

Some of these results were also confirmed by the users' ratings of the statements in the questionnaire; the values derived suggested that the better the subject's command of the instrument is, the more likely she is to follow the chord symbol at the same time being less likely to use the feedback displayed by the system. Similar to the results of the first user study, the users' ratings of the statements on their perception of their own performance (i.e., feeling to be in control and of playing something suitable) were rather inconclusive since they varied heavily. The statements on sufficient support and being too restricted were rated analogously although a slight relation between the subject's knowledge in musical theory and those statements was visible (with more knowledge, the support was rated to be less sufficient, while the subject's felt more restricted). These observations still need to be confirmed by

an in-depth statistical analysis. Since this would go beyond the scope of this thesis, it is left for future work.

5.3.3 Discussion

With these results, the aspects in the system to be adjusted were rapidly identified. The discontent of the advanced users with the support was mainly due to the manner, in which the various mechanisms and features were tuned with the help of the skill parameters since most features were tuned with only one of the parameters. This resulted in some special cases (e.g., the jazz trumpet player not used to playing the keyboard) not being supported correctly. Additionally, the classification of the skill parameters still seemed to be too rough to account for most cases. The difference between classical musicians and jazz musicians¹⁰ also seemed not to be sufficiently covered by the parameters.

In terms of collaboration, the announcement of a solo change and the display of the roles was insufficient and clearly in need of a more highlighted representation. This new depiction, of course, needed to be accompanied by the introduction of longer and more flexible soloing times. A solo's duration should not be bound to the tempo of a song since this leads to a hectic change of soloist for fast songs, while in slow songs the accompanying players wait too long for their turn. This should also limit the effect of the velocity bending mechanism to animate users to play harder since they would be better informed on their own role. Still, this mechanism seems unavoidable to support the soloist in being the centre of the performance. To further ease the effect just mentioned, the velocity curve should not be bent as hard.

As for the chord triggering, the mapping of the mechanism and the feature itself needed to be revised. The mechanism should be able to decide on the coherence of several pressed keys. It could also be beneficial to deploy different mechanism distributed over the clavature since chords can be played over the whole keyboard, while being applied differently in certain areas.

The depiction of the feedback for the batons proved to be inappropriate, as it deploys a precise metaphor for this interface with rather fuzzy control. To reflect the characteristics of the batons, a fuzzy feedback would be more adequate, and should not mislead the users into anticipating precise control. Accordingly, a suitable metaphor for displaying the gestures should be found.

5.4 Changes in the Third Cycle

The changes necessary to eliminate the identified problems were viable, without a total restructuring of the system. The problems only indicated the rearrangement of some mechanisms and the addition of other aspects. Furthermore, the back-end was also altered with the help of the findings in the last user study. Some of these changes had to be accounted for in the front-end: First, the probabilities are calculated for all 128 notes as defined by the MIDI standard in the revised back-end, leading to some modifications in the mechanism for note and gesture processing. Secondly, probabilities are now determined for each player individually, since the back-end also uses the users' input to calculate the probabilities. Accordingly, a mechanism for reporting note events to the back-end had to be added, and the distribution of the probabilities needed to be changed for individual assignment.

¹⁰In contrast to jazz musicians, classical musicians are not used to improvising.

5.4.1 Design

The redesign of the system followed one basic principle: the original design should be maintain as much as possible, since the overall design of the second prototype (graphical user interface and architecture) had proven to be effective in regard to the main goals of the system. Therefore, the main mechanisms were retained and extended if necessary. The architecture was only slightly altered to reflect some of the changes made. Figure 5.11 depicts the interaction concept, and the output of the final system.

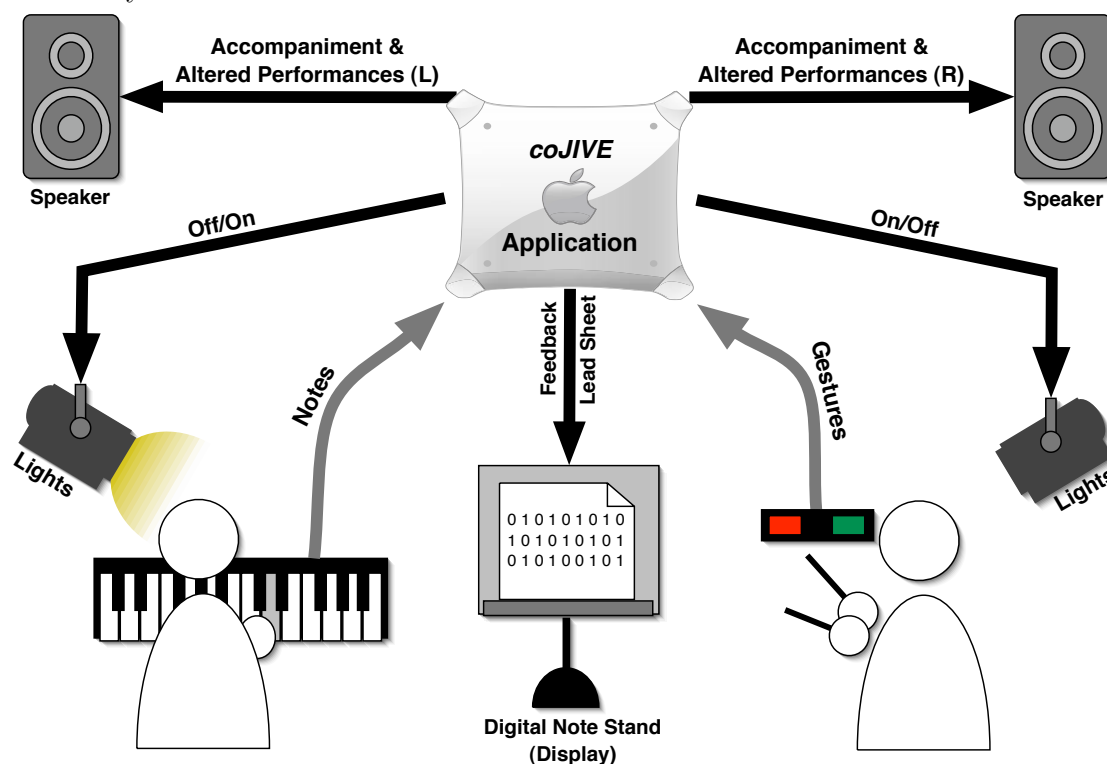


Figure 5.11: The interaction concept of the final system.

5.4.1.1 Performance Control

To further conform to real jazz sessions, a mechanism for reciting a song's main melody (theme) was needed; most users cannot be expected to play a given melody even with a notation. For that purpose, the melodies of the songs mentioned in subsection 5.1.1.3 were stored in MIDI files and referenced in the files describing the songs. In accordance with real jazz sessions, the revised system starts a session by playing the theme¹¹, after which the usual solo alteration commences. The repetition of the theme after the solos was left out intentionally. Since the users' performances would be over by then, the users would surely not be willing to listen inactively to the system performing the melody again. The bass creation mechanism was also altered to allow more varied bass-lines. The calculation algorithm was changed to approach the first bass note of the next bar with the last notes in the current bar to achieve more fluid transitions. Additionally, some bass-lines were stored in MIDI files and referenced in the song fields analogously to the themes. If such a bass-line

¹¹While *coJIVE* recites the theme, all users have the role of accompanying the theme.

is present for the current song, the automatic calculation of the bass performance is suspended.

The length of solos was changed from a predefined number to a dynamically calculated number of bars spanning between 30 seconds and one minute. The calculation is based on two aspects: the tempo of the song and the behaviour of the soloist. To evaluate the latter aspect, the number of note events and their mean velocity in the time of each beat is recorded. These values are especially recorded after 15 seconds into the solo as presumed peaks assuming the user to have realised his new role by that time (should they fall below certain values, they are substituted by default values). After about 25 seconds into the solo, and then repeatedly 5 seconds before a granted prolongation ends, a decision is made whether to increase the solo length for about 15 seconds based on the recorded values. This prolongation is not granted if the values mainly decreased over the last 10 seconds, their mean values are below 50% of the presumed peaks or the prolonged solo time would surpass the limit of 1 minute.

This new structuring of sessions, of course, needed to be supported by new ways of communicating the roles, and changes to the users since the depiction deployed had proven to be insufficient. A new metaphor was found to implement this communication: the spotlight. On a stage, the leading musician is sometimes highlighted by levelling a spotlight at her. To emulate this procedure in *coJIVE*, a Teleo system [Making Things LLC, 2002] was applied. This hardware module is connected to the computer via USB and controls two LEDs (one for the keyboard and one for the batons). The LED representing the current soloist is lit, and an alternating blinking of both LEDs is used to indicate an upcoming solo change. Beside this visual mediation of the session's structure, an audible signal was also included by altering the drum accompaniment. In the last bar before a solo change, the drum plays a short fill-in deviating from its usual pattern to imply the change.

5.4.1.2 User Support

The user level scheme was further adopted to deploy a continuous range of values for each parameter instead of discrete levels. Accordingly, a user's ability for each parameter can be approximated in more detail. As the old scheme also did not properly account for the differences between jazz musicians and classical musicians, an additional parameter (*Knowledge in jazz theory*) was appended to allow the estimation of the users' abilities in the field of jazz.

To approach the problem of the system not properly supporting advanced musicians, the thresholds used to configure the assistance features were adjusted. The formulas used to determine their values were redesigned to reflect the different skill parameters' influences on the respective aspects. Such a formula was initially set up to produce the correct values for the most basic cases — total novices and experienced jazz musician —, and then was altered step by step to cover some of the special cases experienced in the last user study (i.e., producing the values leading to the desired behaviour of the system).

The chord triggering mechanism was again extended by another mode: two key triggering. With this mode, a more fluent transition from one- to three key triggering was enabled as some of the advanced users had problems pressing a three key chord.

Similar to the one key triggering, the lowest member of a voicing to be played must be pressed to establish a connection between the keys pressed and the overall pitch of the voicing. For the two and three key modes, another aspect was introduced; keys had to be pressed within a certain time-limit to be considered in the calculation. The users are able to play freely with several keys with this mechanism, while a chord will only be triggered if the keys are pressed more or less simultaneously. The time-limit itself is calculated based on the instrument and theory skills of the user — it ranges from 0 to 400 milliseconds — to account for the fact that less experienced users are not as fast at playing a certain chord. A new mapping of these modes to the claviature (as shown by Table 5.1) was established based on one assumption: if a user plays three keys belonging to a voicing simultaneously, it can be assumed that he wanted to play a chord rather than a sequence of single notes.

Class of users (rough)	Mode used for notes $< C5$	Mode used for notes $\geq C5$
Novice	One-key	Three-key
Advanced	Two-key	Three-key
Classical Musician	Three-key	Three-key
Jazz Musician	none	none

Table 5.1: Chord-triggering modes used based on user level and position on the claviature.

Since the velocity bending mechanism did not have the presumed effect, it was mitigated to not bend the velocity curve as hard. For inexperienced users, an additional mode for this mechanism was introduced: during a solo, this mode bends the velocity curve upwards to facilitate playing loud notes, since often novices are not familiar with the principle of velocity on a keyboard.

A closer look at the probability values supplied by the back-end revealed that at least one of three subsequent note values is very probable in a given context. Accordingly, the size of neighbourhoods used to determine the most probable candidate based on a note does not need to be dynamic. So, the mechanism for looking up the most probable note was redesigned to only consider the direct left and right neighbours of the incoming note.

To further improve performances with the batons, the notes playable were further limited by using the same threshold deployed for finding the most probable note. Notes less probable than this threshold are not represented by a virtual target and cannot be played accordingly. This fact, of course, connects the batons to the users' skills; more experienced users are more likely to cope with occasional dissonant notes.

5.4.1.3 User Interface

The graphical user interface, too, was only slightly changed since it had proven to be effective for the most part. To provide a better overview, the button for opening the options drawer was provided with an icon, and was placed alongside the other buttons (open, play and stop) for a more consistent look. The previously neglected second mode of operation for that button — if the drawer is open, another push of the button will close it — was now added intentionally since some user had been expecting the button to work with the two modes; the additional button for closing

the drawer was still left in the drawer. Changing the tempo of the playback had been very cumbersome with the two buttons. Accordingly, they were replaced by a slider in the new design. Additionally, the labelling of the options button in the player fields was changed to “Settings”, thereby preventing the users from confusing it with the button for the general options.

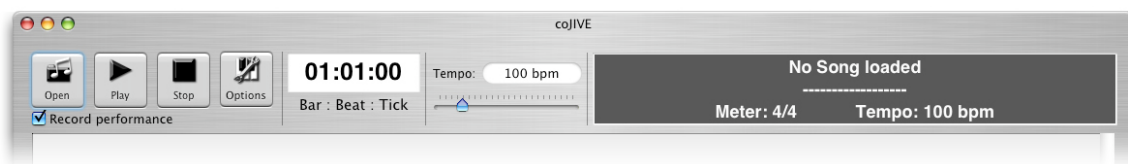


Figure 5.12: The revised controls in the final system.

The changes made to the user level scheme also needed to be reflected in the options dialogue. Sliders were used instead of the previously deployed radio buttons to indicate the range to be continuous; they were each equipped with four tick marks to imply levels similar to the previous version. To allow users to relate even more to the statements, the descriptions at the tick marks were changed to provide a more quantitative classification (e.g. “I play the keyboard once every six months”). The same depiction was used for the new jazz skill parameter that was furthermore added to the dialogue.

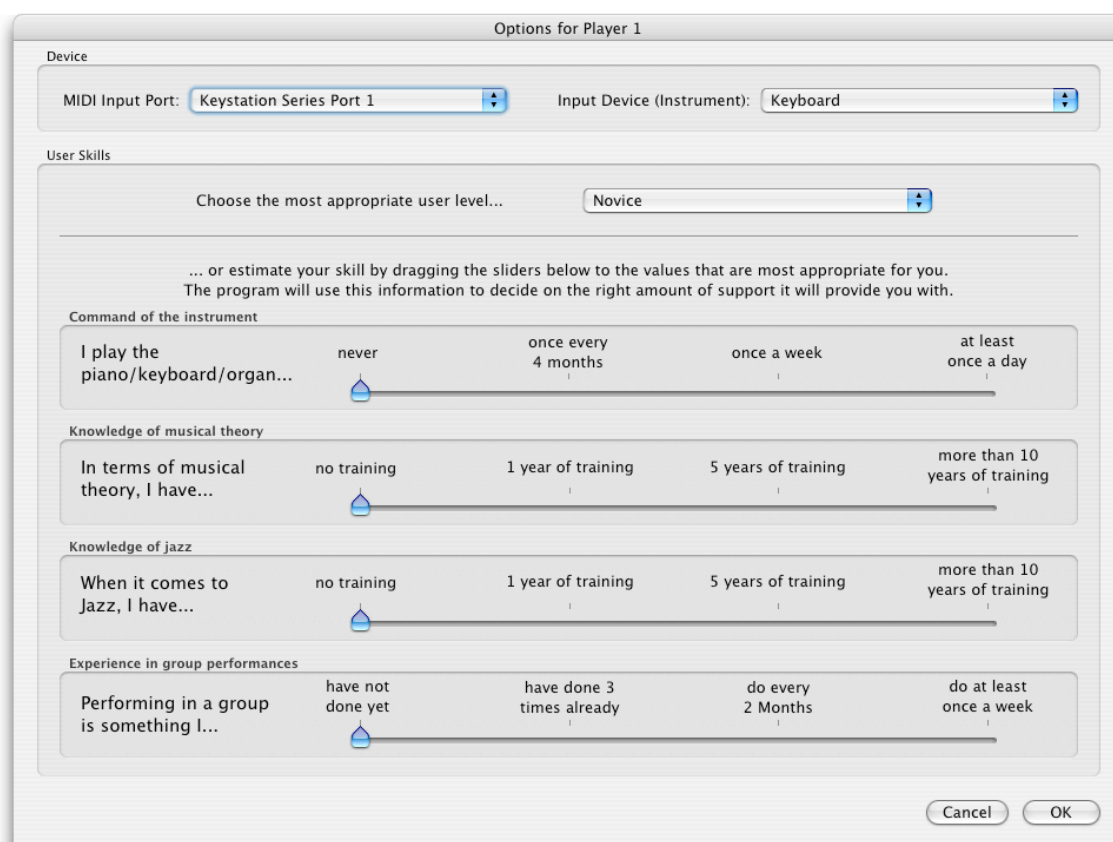


Figure 5.13: The new options dialogue for adjusting input and skill settings.

A more fuzzy feedback was created for the batons to reflect their type of control; the range is depicted by a black rectangle with a single tick mark (middle *C*) to offer rough orientation. Gestures are displayed as green circles in the determined note's area of the range, their radius indicating the velocity.



Figure 5.14: The feedback visualisation for the baton performances.

5.4.1.4 Altering the Architecture

For the playback of a song's theme, another subclass of `AccompanimentOutput` was created: `ThemeOutput`. After the back-end has loaded a new song, an object of that class is provided with the theme. Similar to the other subclasses of `Accompaniment`, it checks for the current time whether a note needs to be played (or stopped), and acts accordingly.

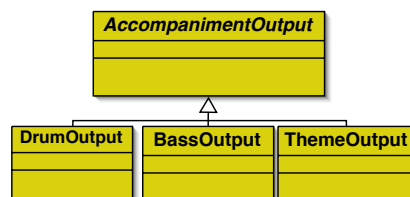


Figure 5.15: The extended inheritance tree of accompaniment classes.

Since the structuring of a session had become a more extensive task, the `MainModel` was relieved of it, and the `PerformanceStructurizer` class was created for that purpose. It includes `NSMutableArray`s for storing histograms on the players' behaviour in the session and implements the mechanisms for determining a session's structure. Therefore, it resorts to some of the functionality supplied by the `MainModel` for that purpose (e.g., informing the player classes).

To facilitate the sending of note events between objects, which usually includes the original notes values, the calculated notes, etc., a new container class was created. `NoteEvent` encapsulates the note originally pressed by the user, the notes determined by the system, the velocity, duration, time of invocation, and the player responsible for the event. In the revised architecture, this class is deployed whenever a note event leaves a `PlayerModel` and to represent note sequences (themes and bass-lines). Figure 5.16 shows what other classes use the `NoteEvent`.

The displaying of the new baton feedback was separated from the `PlayerController` class (responsible for the keyboard-based feedback so far) by creating a new subclass

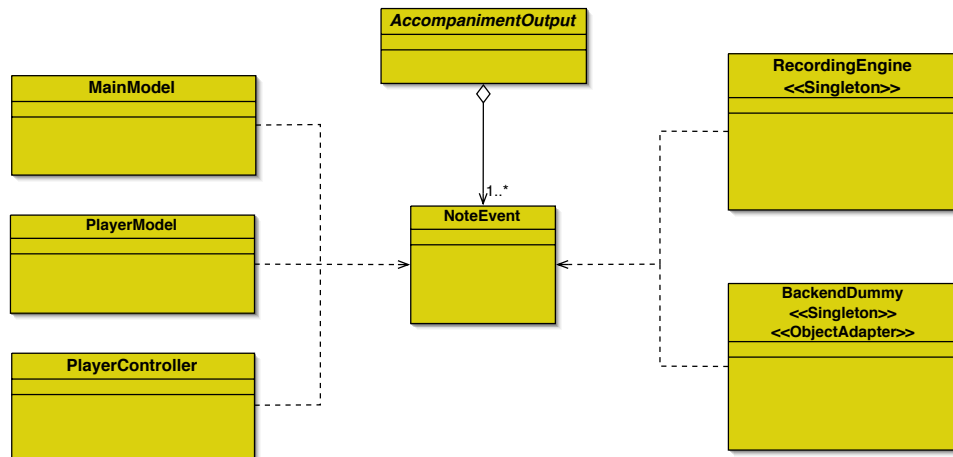


Figure 5.16: The new NoteEvent class and the classes that it is used by.

of `NSView`. `FuzzyFeedback` itself is a view, and is incorporated in the player fields. It contains the mechanisms to depict notes as described in the last section. The range of notes to display can be freely adjusted in an object of the class.

As the chord triggering mechanism — the only reason for the sub-classing of `InputProcessor` — was changed to be applied not only depending on the users' skills, one subclass per mode was inappropriate. Accordingly, the subclasses `NoviceProcessor` and `AdvancedProcessor` were removed, and all the mechanisms were integrated into `InputProcessor`. Figure C.4 shows the revised architecture of the final system, and it is followed by further figures depicting detailed descriptions of the classes present in this version of the system.

5.4.2 Implementation

The implementation of the changes, too, was created mainly by reusing and altering the existing methods and mechanisms. Some of these methods were just slightly extended or newly configured. Some changes in the system's environment were also put on for the implementation of the new concepts.

5.4.2.1 Environment

With some of the additional features to be implemented, new hardware was necessary. And again, a new keyboard was used to further approach the feeling of a real piano. This section will take a brief look at the new devices deployed in the final version of the system.

Teleo

Teleo [Making Things LLC, 2002] modules are hardware components for rapid hardware prototyping that can be connected to a computer via a USB port and programmed using various programming languages (e.g., C, Java). They offer the possibility to control other devices (like LEDs) or connect buttons and report events to all applications that registered with the driver.

Casio Digital Piano

The keyboard used for the analysis of the final version of *coJIVE* is a digital piano with 88 fully weighted keys. Its handling resembles that of a real piano very closely.

5.4.2.2 The New and Revised Classes

Similar to `ChordSymbol`, the new `NoteEvent` class is a container class with the data described in the last section and the corresponding *getter* and *setter* methods. An object of the class can be instantiated at two points in the system: as soon as the processing of a note or gesture is completed, a `PlayerModel` creates a `NoteEvent` to inform its *Observers*. `MainModel` forwards the event to the `RecordingEngine` and the new `PerformanceStructurizer`, while `PlayerController` uses the information for displaying feedback. The `BackendDummy` uses the information on the theme and bass-line retrieved from the back-end to create a representation of these note sequences in form of `NoteEvent` objects.

The `ThemeOutput` was based on the mechanisms supplied for the other subclasses of `AccompanimentOutput`. Yet, a theme is stored in an `NSArray` instead of an `NSDictionary` since the `NoteEvents` objects include the time of their occurrence themselves. The theme is delivered by the `MainModel` that also instantiates the `ThemeOutput` on the start of the application, by calling for the `setTheme:` method. In playback, the `ThemeOutput` stores the first `NoteEvent`'s time of occurrence in three `int` instance variables. As soon as this point in time is reached (i.e., when `PlaybackEngine` triggers `setBar:Measure:Tick` in the `ThemeOutput` object with the corresponding values), the notes of the event are triggered (i.e., played or stopped, based on the velocity), and the time of the next `NoteEvent` in line is stored.

The same mechanism is deployed for the `BassTheme` in addition to the bass calculation. To alter the calculation as described above, a set of `NSArray`s describing the bass sequence for one bar by the indices of the notes in the delivered voicings. Whenever a new voicing is delivered together with the root note of the next chord (`setNewChord:nextRoot:`), the member of the chord voicing closest to the next root note is calculated, and the respective `NSArray` ending with that member's index is chosen and used to create the `NSDictionary` for the "old" bass creation mechanism.

The `PerformanceStructurizer` is instantiated by the `MainModel`, and is informed on the number of player whenever they change (using `setNumberOfPlayers:` and `PlayerLeft:`). Most of the mechanism used to structure a session in the `MainModel` of the last prototype was re-implemented here, extended by the histograms mentioned above and the functions to analyse their content. For each player, four histograms are maintained: the density and velocity history for the current beat and the whole session. As soon as a beat is completed, the recorded value for density (number of events played during the beat) and the mean value of velocity is transferred as the next values into the respective session history. The `PerformanceStructurizer` is furthermore not directly connected to the `PlaybackEngine` but receives notifications on each new beat and bar from the `MainModel` (i.e., by triggering `nextBeat` or `nextBar:`).

To allow the decision on whether to prolong-a-te a solo, several values are maintained in instance variables and adjusted whenever the tempo is changed (as they approximate a certain time in bars): the bar marking the next solo change, the number of bars to inform the players on upcoming solos (about 3 seconds before the change), the number of bars to take the peak values after (15 seconds into a solo), the number of bars the current solo is running already, as wells as the minimum and maximum number of bars in a solo. To incorporate the Teleo system that controls the LEDs,

the corresponding library is used in the `PerformanceStructurizer`. With the help of functions provided by this library, the LEDs can be turned on or off. Further information on upcoming solos or actual changes of the players' roles are forwarded to the `MainModel` for distribution.

Like the `LeadSheetView` class, `FuzzyFeedback` is a subclass of `NSView` and uses the `NSBezierPath` class to draw the background and circles. It was connected to the player fields with the `InterfaceBuilder` tool. Therefore, an `NSTabView` was created, and the buttons depicting the virtual keyboard and a `FuzzyFeedback` object were placed in different tabs that could be selected by changing the instrument in a player's options dialogue. The range covered by the `FuzzyFeedback` can be set via `setRangeFrom:To:` that also triggers the calculation of the central tick mark depicting the *C* note nearest to the centre of that range.

The class also introduces a notion to adjust the feedback to the instrument used: an instrument factor can be set via `setInstrumentFactor:` to an `int` value describing the maximum number of notes simultaneously playable with the instrument (i.e., two note in the case of the batons). An `NSMutableArray` is filled with that number of `NSBezierPath` objects, each one depicting a single circle representing a note. If more than that number of notes is played, the first circle is deleted and replaced by another circle depicting the new note.

To provide the revised back-end with the input from the users and to reflect its individualised probabilities, the `BackendDummy` was extended. The `MainModel` informs it of played notes by triggering its `triggeredNote:` method, thereby passing the notes on to the back-end, and triggers the private method `setProbabilitiesForPlayer:` to send the new probabilities to the player responsible for the note.

To set the number of players in the back-end, the `MainModel` triggers `prepareForClient:withNumberOfPlayers:` in the `BackendDummy` as soon as the playback is started. Whenever `chordChangeNumber:` is triggered, the new voicings are distributed similar to the last prototype, and the probabilities are distributed for each player individually. The new private function `convertMidiTrack:` is used to transform MIDI tracks received from the back-end¹² into a series of `NotesEvent` objects.

5.4.2.3 Note and Gesture Processing

The new thresholds were implemented as instance variables of the `InputProcessor` class and are recalculated each time the skill parameters are changed (and reported to the processor object through triggering `setInstrumentSkill:andTheorySkill:andJazzSkill:andCollaborativeSkill:`). Since the values for the skill parameters come from a continuous range (0 to 3) as `float` values, the thresholds are also mostly stored in `float`-variables. The following thresholds are used in the `InputProcessor`:

- `probabilityThreshold:`
0.0078 ($\frac{1}{128}$) is the basic value for this threshold coming from an equal distribution. For more advanced players, 0.0048 is subtracted from that value to allow outside notes slightly raised in probability by the back-end based on the

¹²The back-end loads MIDI tracks with the help of the *jdkmidi* library provided by Koftinoff [1986-2004].

player's input (described by Klein [2005] for more information) to be played. The value

$$0.002 * \frac{\text{instrumentSkill} + 2 * \text{theorySkill} + 4 * \text{jazzSkill}}{7}$$

is further subtracted to adjust the threshold according to its usage in the system (e.g., the jazz skill is weighted higher than the other parameters since jazz musicians should know what notes sound good).

- **temporalDensityThreshold:**

This threshold is used to configure the velocity bending mechanism in terms the user is currently accompanying another user. It is calculated by the formula

$$2 + \frac{2 * \text{collaborativeSkill} + \text{theorySkill} + \text{instrumentSkill} + 2 * \text{jazzSkill}}{6}.$$

The collaborative and jazz skill parameters are weighted higher than the other parameters since the threshold is used in a collaborative context especially influenced by the characteristics of a jazz session.

- **neighbourhoodDensityThreshold:**

Since this threshold was deployed to determine whether a player properly used the keyboard, it was mainly based on the instrument and theory skill parameter:

$$2 + \frac{\text{instrumentSkill} + \text{theorySkill}}{2}.$$

The 2 as the basic value arranges for absolute novices that if the left and right neighbour of a note are sounding already, the note itself will not be played to prevent dissonance by overloading the performance.

- **chordThreshold:**

The only threshold stored as an `int` value¹³ is determined by a more thorough procedure based on the instrument skill parameter. Should this value be lower than 1, the threshold is set to 1 if the user exhibits advanced jazz and theory skills, otherwise it is set to 0. With a instrument skill parameter value between 1 and 2, the threshold is set to 1 for an advanced jazz *or* theory skill and it is set to 2 if both these skill parameters are set to an advanced setting. For an instrument skill value of 2, the threshold at least exhibits the value 1, incremented by 1 if jazz and theory skill are advanced or incremented by 2 if both skills are set to an expert setting. Should the instrument skill be even higher, the threshold at least contains a value of 2, and only an expert jazz skill can increment it to 3.

To implement the new chord triggering mechanism, more (private) functions were implemented: `findChordForOneNote`: reuses the implementation of the one key chord triggering deployed in the last prototype. `findChordFor:triggeredBy`: can be configured to, based on the given note, find a voicing that has at least a given number of its member sounding already, and therefore is used for the two and three key

¹³This fact is due to the discreet number of modes the chord triggering mechanism provides.

triggering mechanisms. This function also uses the private function `notesPlaying-InVoicing:equalOrMoreThan:onOctave:` that incorporates some of the functionality deployed in the three key triggering in the last prototype and is used to establish a distinction between the sub-tasks in the mechanism (i.e., finding a voicing and the octave it is based on, then checking how many of its members are sounding already). The function `findSuitableChordFor:` declared in the previous prototype is used to interpret the `chordThreshold` and the value of the incoming note in accordance with the mapping described in table 5.1 and respectively trigger one of the functions just mentioned.

The mechanism for mapping gestures to notes needed to be adapted to account for two major changes: since the revised back-end calculates probabilities for 128 notes instead of only 12, the accumulated probabilities for one octave are no longer normalised to a certain value. Additionally, not all notes in the observed range are counted based on the `probabilityThreshold`. To incorporate these changes, the sum of all note probabilities higher than the threshold is calculated whenever new probabilities are delivered. This sum is used when a gesture is reported to map the position of the gesture to the range of relevant notes. At summing up the probabilities of the relevant notes to decide on the note to trigger, the probabilities of the irrelevant notes are also left out.

The triggering of note and the gesture processing were consolidated in `incomingInstrumentData:` in the `PlayerModel`, which receives all necessary data collected in an `NSDictionary` object. Among the information is an `int` variable describing the type of event (note or gesture) that is used to decide on what processing method to use. This change was made to incorporate more event types or change the processing steps, while maintaining a central interface for the input objects and to centralise the steps similar to the processing of both events. The figures C.10 and C.11 (appendix C) show sequence diagrams depicting the sequence of methods and functions performed in the revised system for processing a note or gesture.

5.4.2.4 System Performance

Since the gesture recognition did not work properly in the last prototype, the revised system was changed to deploy the batons' internal recognition of gestures. The Buchla Lightning System reports each type of gesture by sending a `NoteOn` message with a specific note value, and calculates velocity and duration for each gesture by itself. Accordingly, the x axis value needed to still be observed to determine the position of a gesture. The note values for downward hits were identified and connected to the triggering of the gesture processing (i.e., calling `incomingInstrumentData:` in the `PlayerModel`).

A closer look at the CoreMIDI documentation revealed, that the courses of events depicted in the figures C.2 and C.3 in the second prototype were executed on a high-priority thread created by the CoreMIDI server. This thread could disrupt the execution of the application's thread for quite some time. To bypass this problem, the triggering of `incomingNoteData:` was made indirect by passing the method's selector to the `PlayerModel`'s `performSelectorOnMainThread:withObject:waitUntilDone:method`¹⁴, which then triggers `incomingNoteData:`.

¹⁴This method is defined by `NSObject` to call a method from arbitrary threads on the main thread.

This solution led to another complication: since the application's main thread was also busy with drawing the lead sheet, etc., the instruments could only be played with a noticeable latency. To solve this interfering problem, instead of using `performSelectorOnMainThread:withObject:waitUntilDone:` to trigger `incomingNoteData:` on the main thread, `NSThread`'s class method `detachNewThreadSelector:toTarget:withObject:` was used to create a new thread for each event. This new solution implied some more changes to secure the synchronisation of access to some objects: the `PlayerModel`'s `NSMutableDictionary` for the storage of triggered and calculated notes was secured by deploying an `NSRecursiveLock`¹⁵ since storing or removing a note event leads to several accesses to the container object. With these changes, the latency was minimised to rarely noticeable delay, and the overall performance of the system was improved.

5.4.3 Analysis

The effectiveness of the changes made to obtain this final version of the system, of course, needed to be evaluated. Once again, a user study was conducted for that purpose. Initially, 30 people tested the system in pairs in this final analysis phase, but as the latency mentioned in the last section was first discovered by some of the experienced subjects in the first few tests, the data retrieved from the first three tests had unfortunately to be discarded.

5.4.3.1 User Tests

Again, some of the subjects who had attended in the last series of tests were also asked to attend this user study, too. In addition, some experienced musicians could be persuaded to take part in the tests: an experienced jazz pianist, a jazz guitar player, and a jazz bass player, as well as four people with some experience in playing the keyboard or the piano were added to the list of subjects.

Similar to the last user study, the subjects were asked to perform in pairs with the system. Yet this time, the procedure itself was more guided as some statements on how close this version of the system gets to achieving the initial goals needed to be achieved. Each subject was to play each instrument in two phases: in one of the phases, they were supported by the system, while in the other phase, the support was turned off (by setting the skill settings to the setting for professionals). Half of the pairs started with the supported phase, and the other half started with the unsupported phase to ensure that the effects of both possible successions was covered in the user study. After setting up the system for the supported phase, the subjects were told about the kind of support the system would provide. This was done to see whether the subjects would use the support more deliberately once they knew how it worked. Since the solos were much longer this time, each phase was limited to two solos per players.

The evaluation of the tests was once more conducted with the help of questionnaires. These questionnaires were adapted to reflect the changes to the representation of the user level scheme, and included a more clear distinction between questions on the batons and the keyboard. It was separated into three sections this time. The first section, again, focused on the skills of the subject. It contained five lines, each one

¹⁵This subclass of `NSLock` allows being locked several times by the same thread.

was meant to be filled out with a certain instrument the subject plays, and how often she plays this instrument. Additionally, the subject was asked to state how long she was educated in music, and for how long she had played jazz. The second section contained statements that could be again rated in five steps between “I totally agree” and “I do not agree at all”. Another column was added to each statement that could be marked with a cross the subject wanted to abstain from rating a statement. For a clear distinction between the circumstance in question, the section was separated into four subsections, the first one covering the keyboard:

1. **I followed my own performance on the screen.**
2. **The feedback on the screen did help me in performing.**
3. **The system’s assistance in the supported phase was sufficient.**
4. **The assistance to play chords was really helpful for me.**
5. **The system has limited my performance too much.**

Similar statements were used in the next subsection on the performance with the batons:

6. **I followed my own performance on the screen.**
7. **The feedback on the screen did help me in performing.**
8. **The system’s assistance in the supported phase was sufficient.**
9. **The system has limited my performance too much.**

In both of these two subsections, the subjects were additionally asked to rate their own performance in the supported and the unsupported phase with a rating between 1 and 5 (1 being the best rating). This self-rating was added to obtain opinions on the question whether the system could help users creating subjectively “better” performances. The subsequent subsection was concerned with more general statements on the test:

10. **I followed the chord symbols during the performance.**
11. **I orientated my performance towards the chord symbols.**
12. **The system has lowered my inhibition threshold to try new things.**
13. **The system has help me to enjoy the performance (even more).**

The last subsection covered the collaborative aspect of the system with statements similar to the ones used in the last user study:

14. **My partner and I played well together.**
15. **The system helped us with that by systematically deploying support.**

16. **It was obvious how the system structured the session.**
17. **We abided the structure implied by the system**
18. **I noticed in time if my solo was going to start.**
19. **I noticed in time if my solo was going to end.**
20. **I would have liked to decide on the length of the solo myself.**
21. **I found the specification of the structure by the system to be bothersome.**

The final section of the questionnaire again offered space to write down comments, remarks, etc. for the subjects to collect further opinions. Similar to the last two user studies, there was no time-limit was imposed on filling out the questionnaire.

5.4.3.2 Results

With the revised handling of the batons, the users seemed to be really enjoying the performance. To mark the area in which the batons could be played, a piece of duct tape was put on the floor in front of the sensors. This courtesy seemed to enable most subjects to play more freely. Most of the subjects who had attended this user study as well as the last one mentioned the handling of the batons to have improved noticeably, and some remarks mentioned the new feedback to be more suitable. Yet, several subjects expressed their wish to choose the note to be played, even if they are not experienced in music. The threshold-based limitation of notes to be played with the help of the batons seemed to make no clear difference comparing supported and unsupported performance from just observing. The self-ratings by the subjects (discussed later in this section), however, indicated them to have sensed a difference.

The structuring of the sessions was clearly more reflected by the subjects behaviour, which was mainly due to two aspects: on the one hand, the subjects were told before the session was started how the system would structure the session, and how changes would be indicated. On the other hand, the LEDs proved to be very efficient. The subjects clearly reacted to the signals, which was also confirmed by their indication in the questionnaires. Analogously, the subjects behaviour was often reflected by the system. If a subject stopped playing in his solo, the system would often switch the roles of the subjects after a few bars¹⁶. The more experienced subjects, however, annotated that the structuring itself was still inappropriate since a solo could end or start in the middle of the song structure.

Although the subjects made better use of the chord triggering assistance, they generally were very cautious with deploying it in their performance. Yet, their ratings of the respective statement in the questionnaire implied this feature to be really helpful. This may indicate that with an insufficient education the use of chords in the performance is a difficult task even with facilitated access. The involvement of the user's input in the calculation of the note probabilities in the back-end seemed also to be beneficial to the performances of more experienced subjects. They were able to play chromatic sequences of notes, without creating dissonance, which seemed to be more appropriate to them according to their questionnaires.

¹⁶This would, of course, only happen after the minimum amount of time in a solo, 30 seconds.

The ratings of the performances made by the subjects themselves when filling out the questionnaires were regarded more thoroughly. Since both the supported and unsupported phases were evaluated, a direct comparison of the subjects' perceptions could be conducted. This analysis was executed separately for the keyboard and the baton performances.

Figure 5.17 shows the distribution of ratings over the five possible values for the keyboard performances. The diagram to the left depicts the ratings for the unsupported phases; most subjects rated their performance badly, and no rating better than 3 was given. In the supported phases, the subjects seemed to be slightly more satisfied with their performance. The right diagram shows a shift to the better ratings. There, the best rating is a 2, and the worst possible rating (5) was never given. This difference in the ratings indicate that the assistance provided by the system resulted in performances superior to what most subjects were able to do without support.

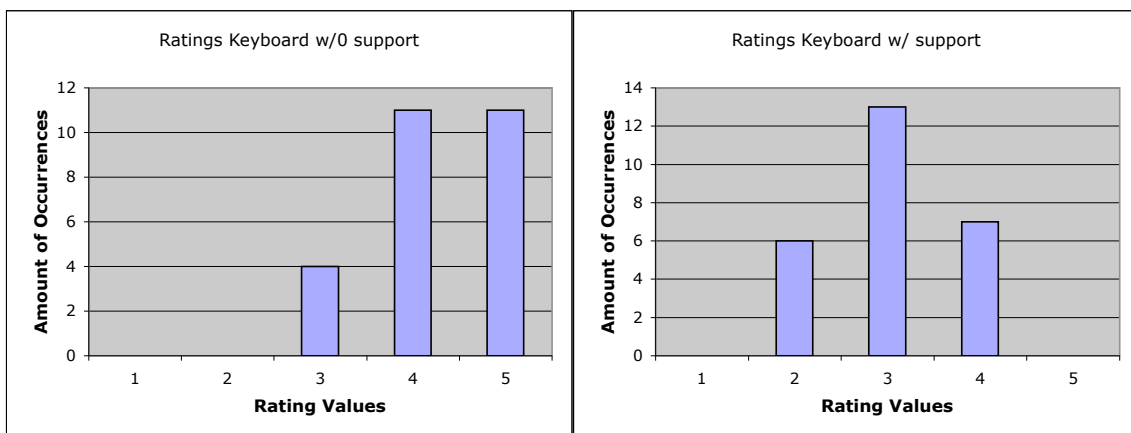


Figure 5.17: The amount of occurrences of the different ratings for the keyboard performances. The left diagram shows the ratings for the unsupported phases, while on the right the ratings of the supported performances are depicted. On an average, the users rated their performances in the supported phases slightly better.

As for the performances with the batons, the subject's ratings exhibited a similar relation. Figure 5.18 depicts the ratings for baton performances without (the diagram to the left) and with the support of the system (the diagram to the right). Again, a shift in the results towards the better ratings can be observed for the supported performances compared to the unsupported ones. In contrast to the keyboard performances, the ratings are more varied here. Although most subjects had expressed their opinion to have enjoyed performing with the batons, they exhibited a more diverse behaviour in rating their own performance.

The results for both keyboard and baton performances indicate that *coJIVE* has met one of its main tasks: the system can help most users to create performances beyond their normal abilities, and thus allows them to have a satisfying experience. Yet, the results presented in this section are not as explicit as they were intended to be. Accordingly, some more work invested in the system would be necessary to widen the gap between the normal capabilities of the players and the performances they can create with *coJIVE*.

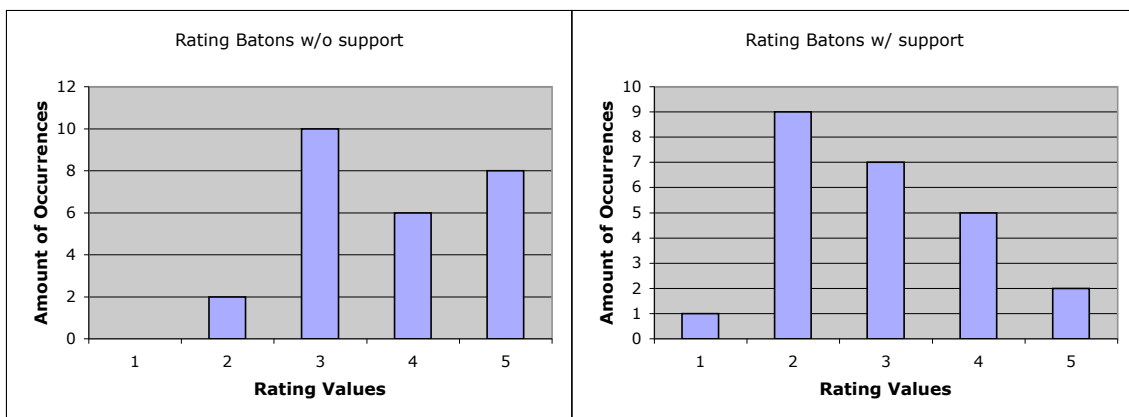


Figure 5.18: The amount of occurrences of the different ratings for the baton performances. The left side shows the ratings for the unsupported phases, the right diagram contains the ratings of the supported performances. Here, too, the users tended to rate their performance in the supported phases slightly better.

6. Conclusion

The work at hand has shown how the *coJIVE* front-end was developed to allow the participation in a simple jazz session to a wide variety of people. Most systems concerned with similar goals (as presented in the chapter on related work) only focus on few aspects of such performances: musical assistants, the creation of an accompaniment, etc. With *coJIVE*, a broader approach was taken that tried to incorporate several of the most important aspects. The results obtained from the numerous user tests indicate that the system was successful in covering these aspects and providing a satisfying experience to a wide spectrum of users; yet, the results also suggest that most aspects still need further development and improvement.

coJIVE's development was based on three main goals. To evaluate the system's efficiency, this chapter will again take a look at those goals and state the system's abilities leading towards their achievement, as well as areas lacking proper coverage by the system. The statements made in this chapter are based on the observations made in the different user studies. The data received in the tests can only indicate a certain coherence since the user studies were conducted with subjects that were not equally distributed over the range of abilities examined. For more quantitative results, a larger series of tests would be necessary, which was beyond the scope of this work.

6.1 Adjusted Musical Assistance

In its three stages of development, the system has greatly improved its capabilities to support different kinds of users in creating improvisations. Throughout the three user studies, especially the inexperienced subjects repeatedly stated to perform beyond their normal abilities when using the keyboard. The combination of features — context-based reduction of usable notes and filtering out the most common mistakes, while providing simple access to chords — has proven to be effective. Still, the tests have shown that the performances created by novices are not just far from what musicians are able to create, but only rarely exhibit a clear melody; further work is necessary in this area.

More advanced users proved to be less likely to be satisfied by the system's assistance; as most of them were more or less classically educated, they had a hard time turning

away from their usual way of performing (i.e., reciting pre-composed songs) and focusing on improvising. In addition, they were irritated by the corrections the system made during their performance. Accordingly, more work is necessary to further close the gap between playing a score and improvising new melodies.

The batons proved to provide an interesting new method of playing to most users; most subjects stated to have experienced an interesting performance when using the infrared sticks. Yet, almost all users expressed their desire to have visual clues to choose what notes to play. Of course, this visualisation would bypass the concept of fuzzy control.

6.2 Collaboration

The collaborative aspect was introduced into the system as a limiting aspect¹, and was revised to be more flexible and to comply with the users' behaviour. As for the direct interaction between users, *coJIVE* did not interfere with the performances to not further limit the players' freedom. To achieve such collaborative behaviour, the system would need to give clues on how to co-operate without disturbing the players.

6.3 Simulating a Jazz Session

The accompaniment provided by the system seems substantial enough to support the users' performances; however, it is still very basic and rudimental. For a more authentic experience, new and flexible methods would be needed to create more genuine bass and drum parts.

Restraining the session structure to the mode of alternating solos (preceded by the theme) seems justified since many subjects exhibited problems with the short solo time of eight bars in the second prototype; modes like the already mentioned trading fours would incorporate shorter solos and are much harder to communicate to inexperienced users. In spite of these circumstances, sessions with several modes of structuring would enhance and render performing with *coJIVE* more varied.

¹This notion still seems somewhat reasonable considering the target users of the system.

7. Future Work

As the previous chapter suggested, some additional work could further improve the system noticeably. Since these improvements were not feasible in the scope of this thesis, a set of ideas for the future course of work will be presented in this chapter.

7.1 Musical Assistance

Although the amount of dissonant notes can be observably reduced by the final version of the system, especially novices seemed still not able to create melodies like experienced musicians. Some more support is necessary in that respect.

7.1.1 Melody Guidance

The user tests have shown that even with *coJIVE*'s assistance, inexperienced players could not always create nice melodies. Klein [2005] suggests that the involvement of the player's input in the calculation of note probabilities in the back-end could be extended to recognise melodic patterns and change the probabilities accordingly. In the front-end this could be supported by returning to a wider search when looking for a more appropriate note to substitute for the incoming note. Of course, this mechanism would imply a cautious application to not irritate users by substituting a note by a distant note.

7.1.2 Rhythmic Guidance

The untrained users also seemed to have problems with finding a suitable timing to play their notes in. When playing the batons, some users just try to constantly hit on the beat of the playback. Since the method of quantisation was found to be inappropriate, a more highlighting approach could be taken. To indicate relevant rhythmic positions, for example, the system could be altered to display signals on each beat, between beats and on swing notes. This depiction of the rhythm could offer a visual relation in addition to the audible accompaniment, which some users may find easier to rely on.

7.1.3 New Chord Voicings

The chord voicings used for the chord triggering mechanism so far were all *narrow*; they span one or at most two octaves. In jazz, however, the piano players often use *wide* voicings spanning three or more octaves. Some of these voicings are even named after the musician who first found them. If these were included in the back-end's database, the two- and three-key chord triggering could be altered to include more sophisticated recognition mechanisms. The distance of the involved notes, for example, could be used to decide on what voicing to trigger. This procedure would, of course, involve more decisions based on the user's skills: What do users know about wide voicings? Who would intentionally choose a wide voicing over a narrow one?

7.1.4 Responding to Classical Musicians

Since most classically educated musicians are more familiar with playing scores, some changes to the system could help them to use this ability in improvisation. Besides displaying the names of the chord symbols on the lead sheet, they could also be depicted as notation; with these notes to read, the classical musicians could try to play different versions of the chord (e.g., by picking three of the notes to play with the left hand).

For the creation of a melody, the system could additionally display the most probable notes as a score in an additional display; this depiction may offer an appropriate representation of the repertoire of notes jazz musicians choose from after having analysed the chord progression. This kind of assistance would, of course, afford a musician to spontaneously choose one of the depicted notes, which may be perceived as a slight abstraction from the initial task of coming up with a whole sequence of notes. Additionally, the depiction of notes may lead to the musician remembering melodies she has played from a score before that she may then be able to recite.

7.2 Collaboration and Session Arrangement

7.2.1 Solo Arrangement

A necessary change of the possible lengths of solos was pointed out in the last user study; in normal jazz sessions a solo starts at the beginning of the song structure and ends together with the last chord of the song. Accordingly, the length of a solo needs to be a multiple of the length of the song's structure. Integrating this in the system will clearly bring it closer to a real jazz session.

Since the new `PerformanceStructurizer` class already introduced the notion of recording histograms of the players' behaviour, it could be used to further analyse the structure of solos. To receive more data on solos performed by real jazz artists, more jazz musicians could be asked to perform solos with the system, which then could record all relevant data. This data could be analysed for a common course of events in a solo (e.g., arc of suspense, distribution of note velocities), which may lead to new methods of supporting solos and sensing its structure. With these structure sensing mechanisms, the system could be able to recognise whether the solo is in a louder or softer phase; this information could be used to adjust the accompanying players' performance.

New modes of collaborative performing — beside the currently implemented alteration of solos — would enrich the system to provide more varied experiences. The already mentioned mode of trading fours would afford a clearer allocation of roles since the time to play for each player is very short (e.g., by displaying a clearly visible countdown to the next solo change) and may probably be only reasonable with more experienced users. With a faster change of roles, the users' performances could be used to influence each other; the solo performance could be fed to the back-end for the accompanying players to adjust the probabilities used for altering their input.

For a more authentic simulation of a real jazz session, the accompaniment provided by the system needs to be further improved. This enhancement could be achieved by using pre-recorded drum and bass performances stored in MIDI files; the accompaniment created with this method would of course be rather static and not responsive to the users' performances at all. A more interesting alternative would be to link *coJIVE* to another system concerned with creating an accompaniment. JOINME [Balaban and Irish, 1996], for example, presented in the chapter on related work would be interesting in the sessions created with the help of *coJIVE*.

7.3 Special Assistance for Jazz Musicians

The assistance provided by the system focuses mainly on abilities necessary for participating in jazz sessions; experienced jazz musicians, however, have already mastered these skills. The question remains what kind of assistance a system like *coJIVE* could supply for these jazz musicians. Based on talking to test subjects with jazz experience, two main ideas for potential features were developed.

7.3.1 The Retry Button

The first idea is based on the assumption that even experienced jazz musicians are not always satisfied by their own performance; a solo may not come to the interesting termination the musician had hoped to achieve (e.g., because he cannot transpose the complex melodic pattern he initially wanted to use for the final of his solo to the harmonic context of the song). In a normal jazz session, the solo cannot be repeated since the player usually has forgotten his ideas by the time the next player starts her solo.

Deploying the recording of the player's previous performance, *coJIVE* could make a difference in that respect. Each player could be equipped with a button, the *retry* button; should the player be unhappy with his performance, a touch of that button could set the session back to the start of the solo in question. The system would replay the player's previous performance to allow the user to retrace the ideas he had in during that solo. As soon as the player starts playing different notes compared to the recording¹, the system would fade out the playback of his previous solo. This feature may allow a musician to recreate the train of thought he had during the performance and thereby retry to create the performance he initially wanted to obtain.

¹At that point the player is assumed to pick up the performance and create new melodies for the parts he wanted to change.

7.3.2 Recording Statistics

Another interesting feature for jazz musicians would be the statistical analysis of the recorded performance. Musicians may be interested in seeing whether they tend to repeat certain patterns or stick to certain notes, while leaving others out entirely. A database with the melodic phrases a player uses could be established and the progress in extending the personal repertoire could be monitored by the player himself.

Additionally, a comparison between the analysis of the back-end and the player's own interpretation of a song's structure could be of interest, too. An analysis of the played notes in the context of a certain chord could reveal, if the player often sticks to the chord notes, leaves the scales determined by the back-end or even uses other scales. These results may, of course, also be beneficial for further improvements concerning the back-end.

A. Max/MSP Objects

This appendix will describe the Max/MSP objects deployed in the first prototype for a more thorough insight in the workings of the prototype's different parts. The objects' individual tasks will be explained without addressing possible combinations.

An object in Max/MSP can have inputs and output (called **inlets** and **outlets**), with which it can receive and send messages or signals. The object itself transforms the input in a certain way to produce an output. Therefore, several parameters can be specified in the object, which sometimes are also represented by **inlets**. An **inlet** can be connected to an **outlet** of another object to achieve a combination of the transformation process of both objects.

A.1 Basic Objects

Message 

message objects do not process input but can store messages and, if clicked, send off these messages. The object's **inlet** can receive *bang* signals¹, which also trigger the sending of the message stored in the object through its **outlet**. If the **message** object receives the message *set XYZ*, its stored message is replaced by *XYZ*.

Button 

A **button** object sends of a *bang* through its **outlet** whenever it is clicked. It can also be triggered automatically by sending a *bang* signal to its **inlet**.

Loadbang 

loadbang objects send out a *bang* signal when the patch it is incorporated in is loaded. This object can be used to set default values or automatically start processes.

Inlet and Outlet 

Usual objects have a (dynamic or static) set of **inlets** and **outlets**. Sub-patches defined by a developer must be equipped with **inlets** and **outlets** to allow connections to the other objects in the patch. **inlet** objects have an **outlet**, through which all incoming

¹*bang* is a standard signal in Max/MSP used to trigger events and processes.


messages and signals are fed to the objects inside the sub-patch. Accordingly, **outlet** objects have an **inlet**.

Patcher 

With **patcher** objects, sub-patches can be defined that can be used like self-developed objects by connecting them to other objects. This is reasonable if the sub-patch serves an explicit task and is very likely to be used again or to arrange sub-tasks hierarchically in the main task. Accordingly, sub-patches itself can include further sub-patches.

Metro 

A **metro** object sends out ticks in a certain frequency through its **outlet**. The **inlets** are used to start and stop the **metronome** (left **inlet**) or to send delay between ticks in milliseconds (right **inlet**). The latter value can also be given as a parameter in the object.

Delay 

delay objects can delay an *bang* received in its left **inlet** for a certain time and then send it out through its **outlet**. The delay time can again be set as a parameter or set using the right **inlet**.

A.2 Math Objects

The objects depicted in this section all implement binary mathematical operations; accordingly, they provide two **inlets** for the operands. The second operand, however, can also be set to a specific value by a parameter in the objects (as depicted by the images). The result of the operation is sent out the **outlet**.

Addition 

The two operands are added to a sum.

Subtraction 

The second operand is subtracted from the first.

Division 

The first operand is divided by the second one.

Multiplication 

The first operand is multiplied by the second one.

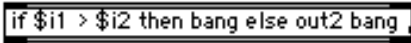
Modulo 

The first operand is taken modulo the second one.

Abs 

abs objects are the only exception to the description given above; they calculate the absolute value of the received value and therefore represent a unary operation. Accordingly, only one **inlet** and one **outlet** are necessary for this kind of object.

A.3 Control and Selection Objects

If Then Else 

To allow programming-like control structures in Max/MSP, this kind of object was

provided with a syntax similar to most programming languages. The amount of **inlets** and **outlets** is determined by the statements used in the objects; **inlets** can be referenced (e.g., if $\$i1 < \$i2...$) in the statements and **outlets** can be used to send signals or messages through (e.g., ... then out1 $\$i1$ else out2 bang).

Select 


A **select** object can check the incoming message for containing a special value (e.g., certain numbers, words). For each of the values given as parameters in the object, one **outlet** is created, which is used to send out a *bang* whenever the respective value is received in the **inlet** of the **select** object. An additional **outlet** is used to pass on the received messages or signals if none of the given parameters applies.

Gate 


gate objects can be used to route incoming messages to certain areas in the patch. A numerical parameter in the object specifies the number of **outlets** of the gate. The right **inlet** is used to receive the messages to be routed, while the left **inlet** receives numerical values specifying the **outlet** to send the messages through exclusively (0 closes all **outlets**).

Counter 

A **counter** object counts the occurrences of *bangs* received on its leftmost **inlet** that also allows to set the **counter** to a specific value. The next **inlet** to the right allows to set the counting to increasing or decreasing, while the next two **inlets** are used to reset the **counter** (either immediately or on the next *bang*). The last (rightmost) **inlet** allows the stating of a maximum for the count. The **outlets** send out the current **counter** value (leftmost **outlet**), set flags for under- and overflow and send out a carry count (rightmost **outlet**); this carry count counts each n^{th} bang, and n can be specified as a parameter in the object.

Send 

Message can be send throughout a Max/MSP application without using patch chords (e.g., from one sub-patch to the other). For that purpose, **send** objects are deployed; the message's identifier is given as a parameter in the object and the value to be sent is fed into the object through an **inlet**.

Receive 

To receive messages send by a **send** object, **receive** objects need to include the same message identifier as the respective send object. The received value is send out through an **outlet**. Arbitrarily many **receive** objects can be used to pick up a dispatched message, thus facilitating the distribution of specific messages to remote sections of an application.

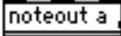
A.4 MIDI Objects

MIDI-related objects reflect the characteristics of the MIDI protocol (described in the next appendix): MIDI messages are received from a certain port² on a specific channel. The port and channel can usually be specified as parameters in these objects.

²In Max/MSP, MIDI ports are classified by a letter.

NoteIn 

To receive messages on note events (*NoteOn* or *NoteOff*) from a certain MIDI port (and channel), a **notein** object with the specific parameters can be used. Such objects have no **inlets** but three **outlets**. While the leftmost **outlet** sends out the note value to specify which note was played, the middle **outlet** sends out the velocity of the event — *NoteOffs* are reported with a velocity value of 0 — and the rightmost **outlet** sends out the channel, on which the event was received.

NoteOut 

A **noteout** object is the counterpart of the **notein** object; it can send *NoteOn* and *NoteOff* messages to a certain MIDI port on a specific channel. Therefore, it has three **inlets** corresponding to the three **outlets** of the **notein** object (for note value, velocity and channel).

CtlOut 

This type of object is used to send *ControlChange* messages to channels of MIDI ports; in the first prototype, this kind of object was used to send the *AllNotesOff* message to the receiving MIDI port. It has three **inlets** for controller value, controller number and the channel to send to. Max/MSP also provides the corresponding **ctlin** object, which, however, was never used in the context of this work.

PgmOut 

The change of programs (i.e., instrument) on one channel is transmitted via *ProgramChange* messages in MIDI; in Max/MSP, this kind of message can be sent to a MIDI port with a **pgmout** object. Two **inlets** are deployed by this type of object for the number of the program to change to and the channel on which to send the message.

**Keyboard**

The **keyboard** object depicts a claviature, which can be used to visualise notes but also to create them. Therefore, this type of object deploys **inlets** and **outlets** for both note value and velocity. Notes received through the **inlets** are displayed by highlighting the respective keys, which can also be clicked to trigger the corresponding notes. For the depiction of incoming note, the **keyboard** object can either be set to monophonic (i.e., only highlighting the key corresponding to the last received notes) or polyphonic (i.e. highlighting a key for as long as the respective note is sounding, independent of other notes).

MakeNote 

To facilitate triggering notes without the need to specifically send the corresponding *NoteOff* messages, the **makenote** object is provided by Max/MSP. It has three **inlets** for note value, velocity and the duration of the note in milliseconds. The last two values can also be set as parameters in the object. With these values, the **makenote** object passes on note value and velocity through its two **outlets**, and after the given duration it sends out the note value again with a velocity value of 0 (the respective *NoteOff* message).

B. MIDI Basics

This appendix will provide a short introduction to the MIDI standard. The information contained suffices to follow the implementation of the Cocoa-based versions of the *coJIVE* system. MIDI stands for *Musical Instrument Digital Interface*, and it provides a standard to connect electronic instruments and devices with each other. Besides hardware elements for physical connections, it also defines a protocol to transmit data between different ports; therefore, a set of messages and conventions was defined.

B.1 Basic Principles

Devices conforming to the MIDI standard can have input ports and output ports. A MIDI port usually has 16 channels to receive or send messages of different types on. Each channel has one program, which normally refers to the instrument that is played on the channel in different pitches (notes). Channel 10 is defined to be the drum channel; instead of using one sound, so-called drum maps are provided in this channels, in which each note refers to a different part of a drum set (e.g., snare drum, crash).

For each played note, two messages are sent: the first message (the *NoteOn*) indicates that the note has just been triggered (e.g., by pressing the respective key); as soon as the note stops (e.g., the key is released), a *NoteOff* message is sent. Both messages include a velocity value representing how hard the respective key was pressed (or released), which is usually used to determine how loud the note is played (corresponding to the handling of a piano). Most devices recognise a *NoteOn* message with a velocity value of 0 to be equivalent to a *NoteOff* message.

Besides predefined controllers for controlling the overall volume in a channel, etc., the MIDI standard also defines a set of controllers that can be individually mapped to a specific parameter by each device. The Buchla Lightning II system[Buchla, 1995], for example, uses four of these controllers to transmit the position of the two batons as x and y values.

B.2 Message Types

Each of the message types in MIDI has three values, the message type and two data bytes (which can either be used to further specify the message type or just contain data); the data bytes can range from 0 to 127. The value for the message types presented in this section is made up of the type and the channel, on which the message is sent or received. Accordingly, a *NoteOn* message has a different type value on channel 1 compared to the same message on channel 2.

B.2.1 NoteOn

The type value for a *NoteOn* message ranges between 144 to 159 to cover all channels. The first data byte contains the value identifying the note to play (the next section will explain how notes are encoded as numbers in MIDI). A velocity for the note is given by the second data byte. A triggered note is regarded to be active since the corresponding *NoteOff* message is received. Should a *NoteOn* for an active note be received, a device can choose to either re-trigger the note or to ignore the message.

B.2.2 NoteOff

NoteOff messages are constructed in analogy to *NoteOn* messages: a type value between 128 and 143, a note value and a velocity. The velocity represents how fast a key was released; some devices use this value to decide on how long the sound fades out after the message is received.

B.2.3 ProgramChange

With a type value ranging from 208 to 223, *ProgramChange* messages only use the first data byte to specify the program to change to. On channel 10, this type of message is used to switch between different drum maps suited for different types of music (e.g., jazz, rock).

B.2.4 ControlChange

The previously described controllers can be set with *ControlChange* messages with a type value between 176 and 191. The first of the two data bytes is used in these messages to identify the specific controller, and the second byte contains the value to set to the controller. In the Max/MSP-based prototype, for example, a *ControlChange* message with the two values 123 and 0 in the data bytes was used to send an *AllNotesOff* message, which causes the receiving device to shut off all active notes in the corresponding channel.

B.3 Note Values

A note is usually made of a pitch class (e.g., *C*, *D*, *E*) and the octave it belongs to. Table B.1 shows how a note's value is defined in MIDI based on its pitch class and octave.

Oct.	C	C \sharp (D \flat)	D	D \sharp (E \flat)	E	F	F \sharp (G \flat)	G	G \sharp (A \flat)	A	A \sharp (B \flat)	B
1	0	1	2	3	4	5	6	7	8	9	10	11
2	12	13	14	15	16	17	18	19	20	21	22	23
3	24	25	26	27	28	29	30	31	32	33	34	35
4	36	37	38	39	40	41	42	43	44	45	46	47
5	48	49	50	51	52	53	54	55	56	57	58	59
6	60	61	62	63	64	65	66	67	68	69	70	71
7	72	73	74	75	76	77	78	79	80	81	82	83
8	84	85	86	87	88	89	90	91	92	93	94	95
9	96	97	98	99	100	101	102	103	104	105	106	107
10	108	109	110	111	112	113	114	115	116	117	118	119
11	120	121	122	123	124	125	126	127				

Table B.1: The note values used in MIDI to identify a note's pitch class and octave.

C. UML Diagrams

In this appendix, numerous UML diagrams are collected to describe the inner workings of the Cocoa-based versions of the *coJIVE*-system. Therefore, the appendix is separated in two sections that both cover one of the two versions.

Besides an overview of the system's architecture and the detailed descriptions of the classes involved, sequence diagrams are used to depict the activities inside the system whenever a note is received or a gesture is recognised. To keep these diagrams clearly laid out, they are only concerned with the classes created for the system and those necessary for the message-passing; accesses to container classes like `NSArray` or `NSDictionary` are left out.

The section covering the final system includes several class diagrams with detailed class descriptions. The declarations in these diagrams were kept in the syntax of Objective-C to facilitate the retracing of specific selectors, mentioned in the section of the implementation

The diagrams presented in this chapter were created with OmniGraffle, a tool for creating diagrams and charts in Mac OS X. It was developed by The Omni Group [2000].

C.1 The Second Prototype



Figure C.1: The architecture of the second *coJIVE* prototype.

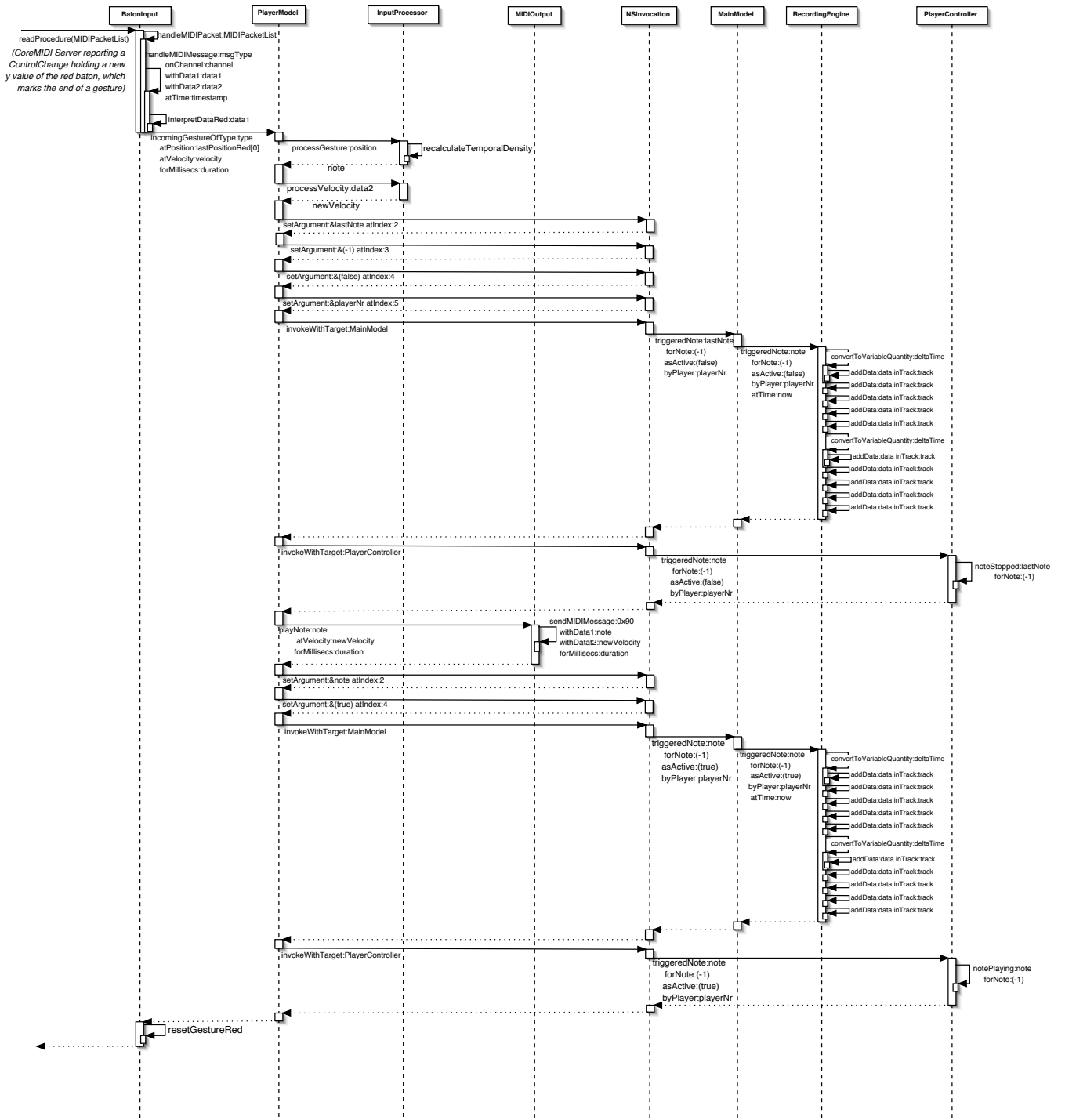


Figure C.3: The sequence of events necessary for processing a gesture in the second prototype.

C.2 The Final System

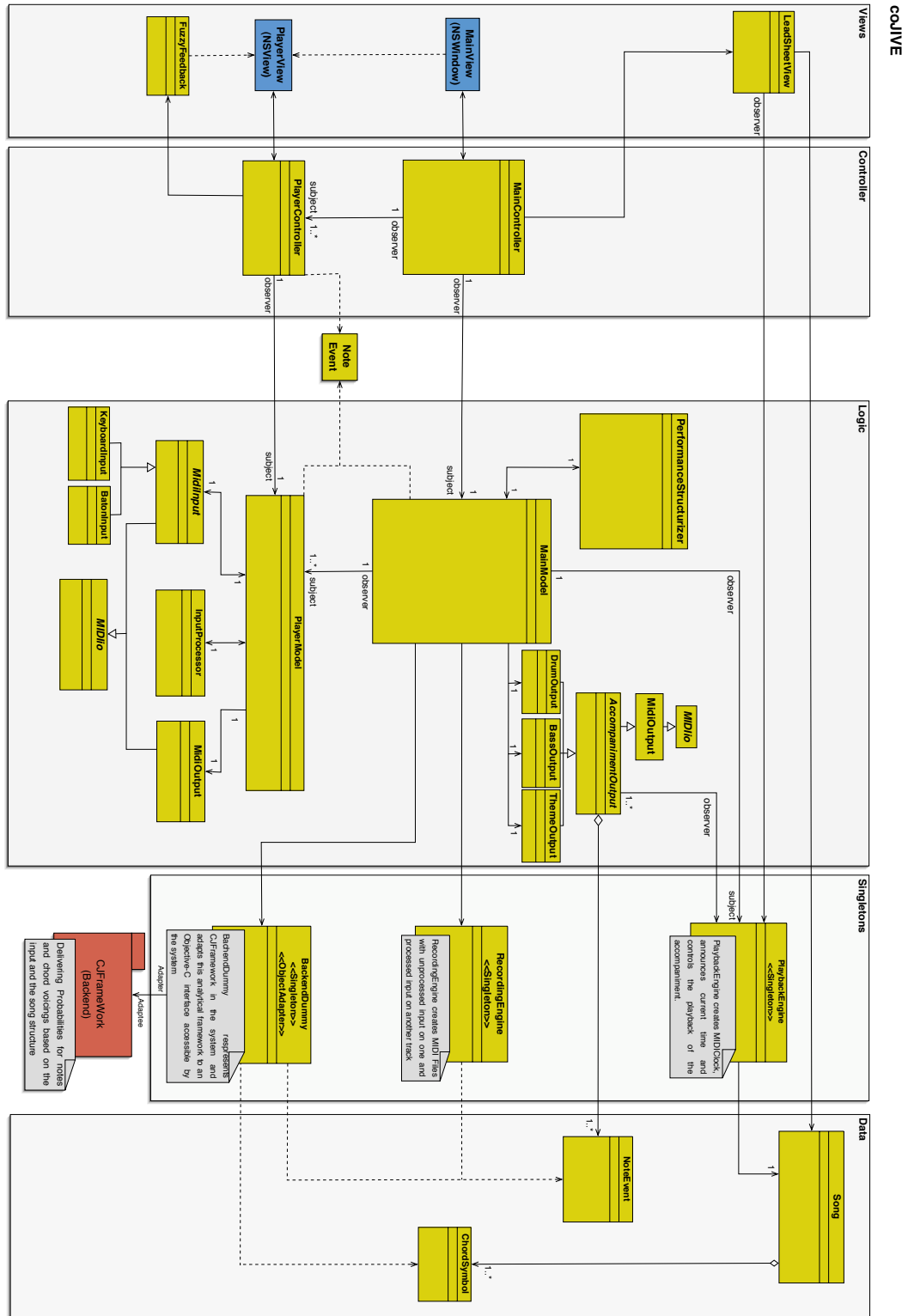


Figure C.4: The architecture of the final *coJIVE* prototype.

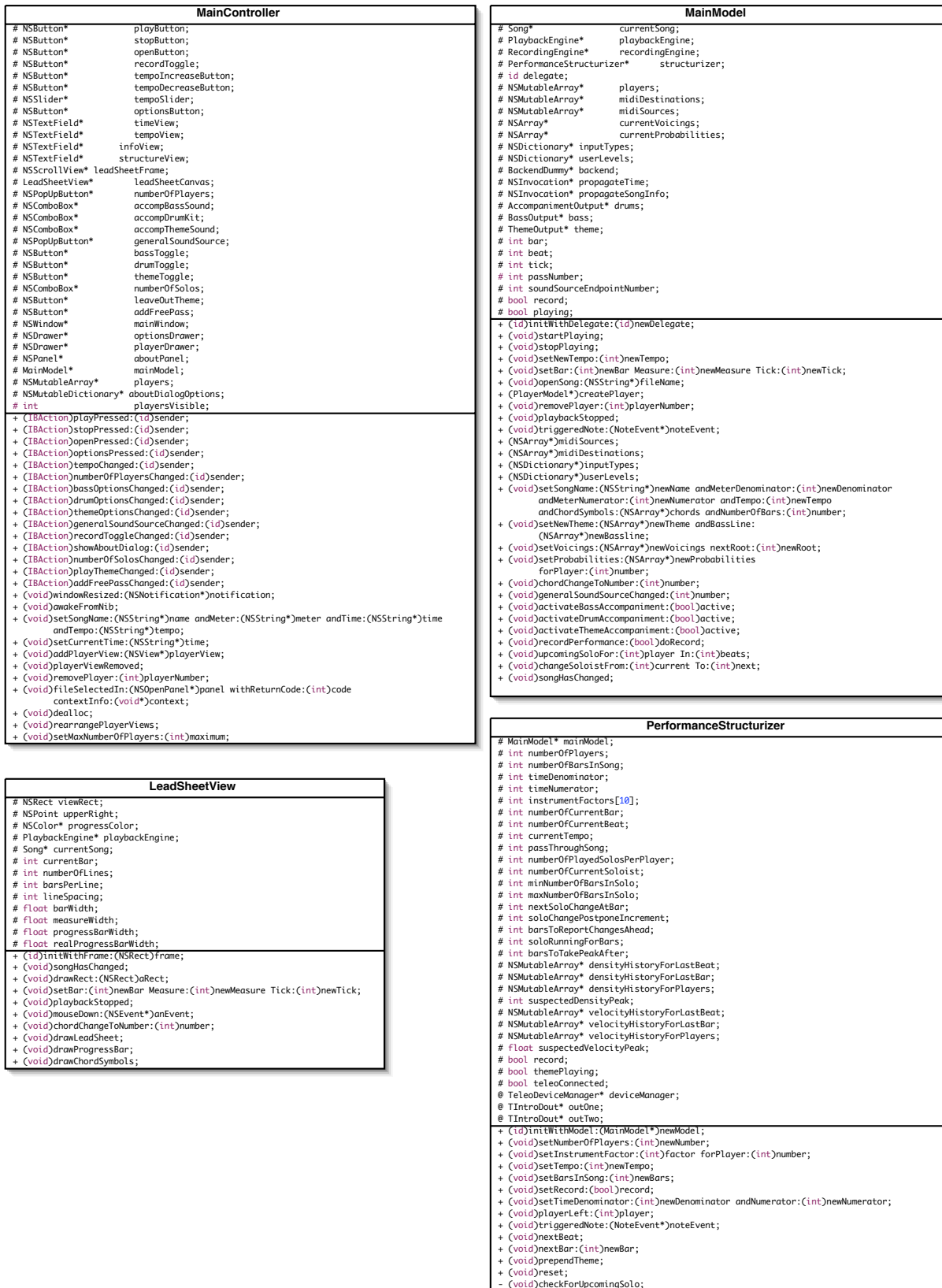


Figure C.5: The classes making up the main part of the application.

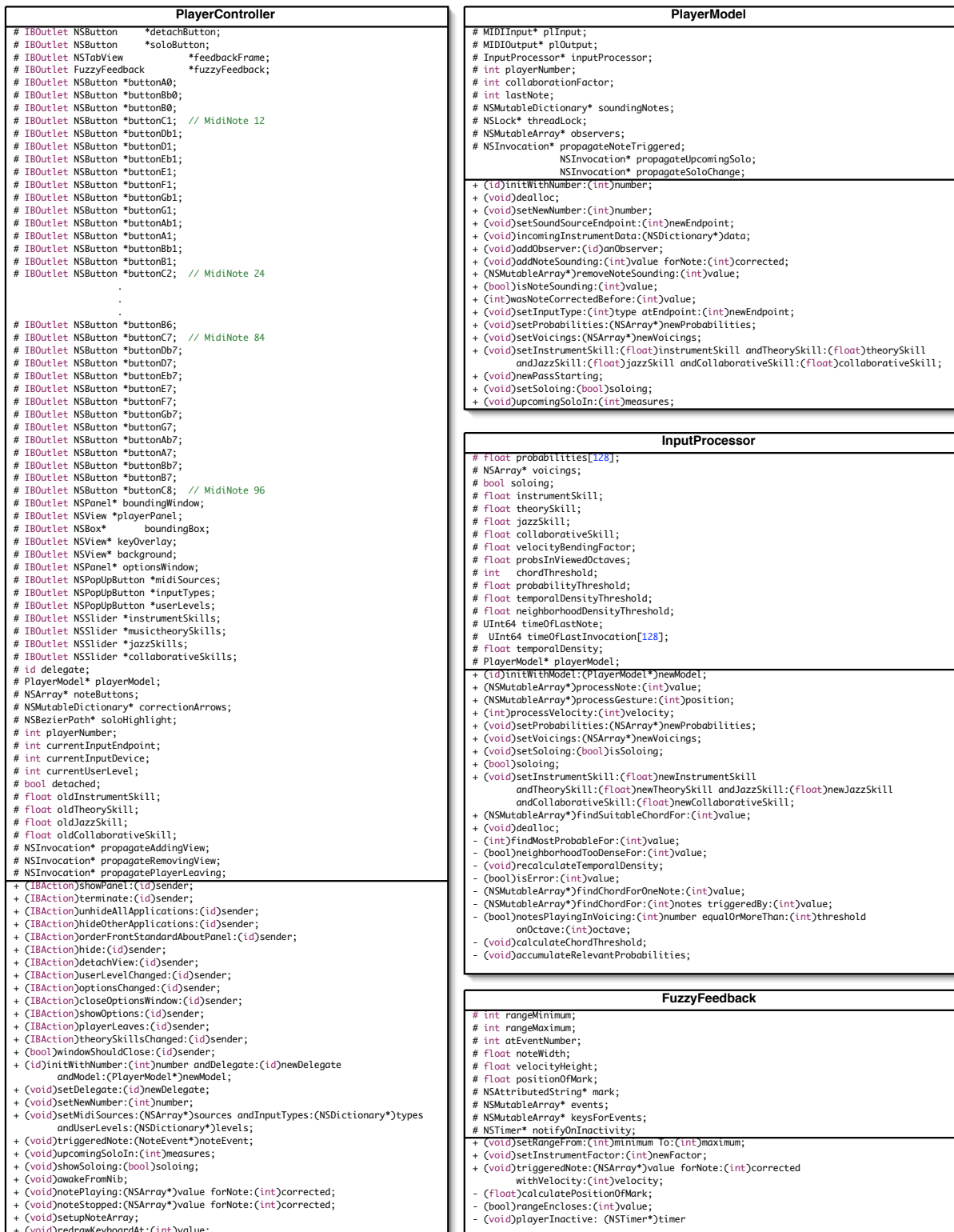


Figure C.6: The classes representing users in the system.

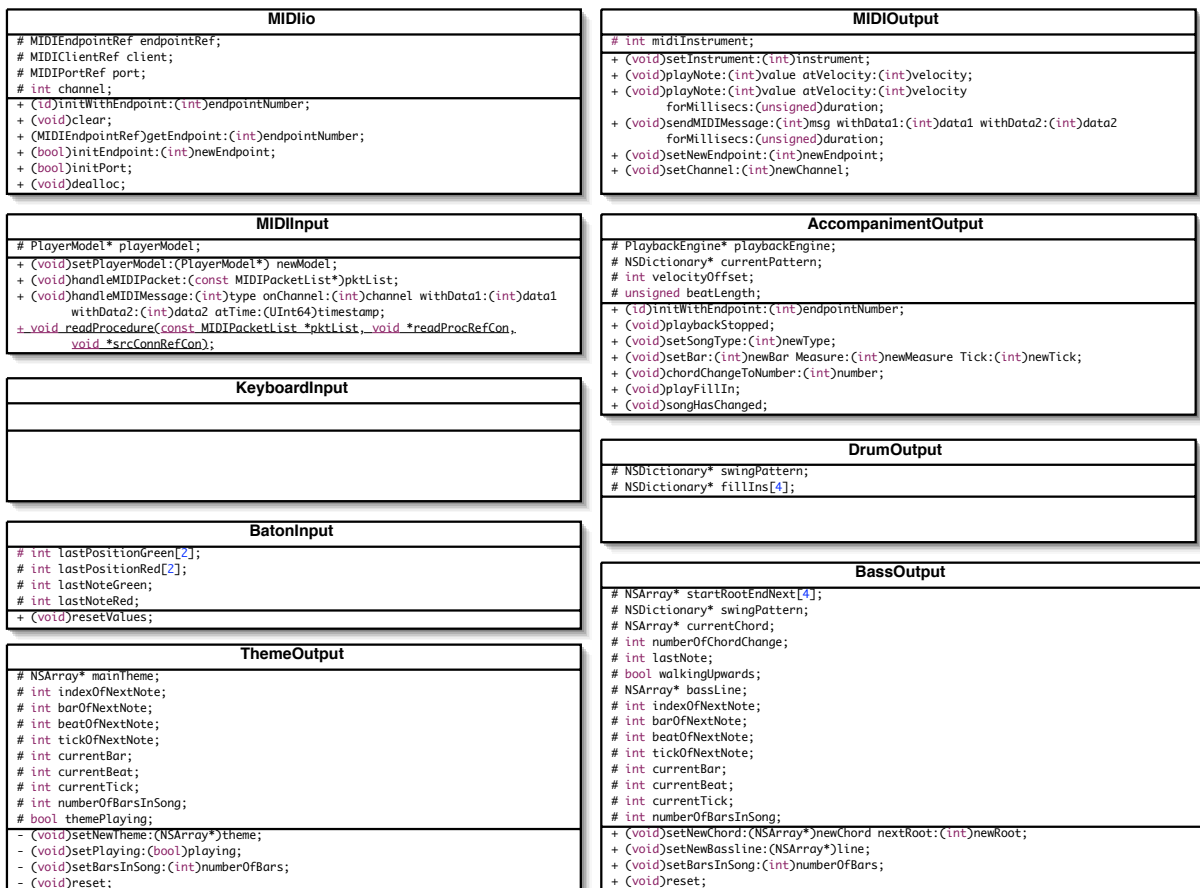


Figure C.7: The MIDI classes.

PlaybackEngine
<pre> # int tempo; # int delay; # int timeDenominator; # int timeNumerator; # int currentBar; # int currentMeasure; # int currentTick; # int nextChordNumber; # int nextChordChangeAt[3]; # NSTimer* timer; # bool playbackRunning; # NSThread* playThread; # NSLock* threadLock; # NSInvocation* propagateChordChange; # NSInvocation* propagateTime; # NSInvocation* propagateStop; # NSInvocation* propagateSongChange; # Song* currentSong; # NSMutableArray* observers; # id instance; + (id)getInstance; + (void)setSongName:(NSString*)newName andMeterDenominator:(int)newDenominator andMeterNumerator:(int)newNumerator andTempo:(int)newTempo andChordSymbols:(NSArray*)chords andNumberOfBars:(int)number; + (void)startPlayback; + (void)stopPlayback; + (void)addObserver:(id)observer; + (void)setNewTempo:(int)newTempo; + (id)getSong; + (int)tempo; + (int)numberOfBars; + (int)numerator; + (void)setBar:(int)newBar Measure:(int)newMeasure Tick:(int)newTick; + (id)init; + (void)setNewTimeDenominator:(int)newDenominator andNumerator:(int)newNumerator; + (void)playing; + (void)resetTime; + (void)nextTick; + (void)chordChange; </pre>
RecordingEngine
<pre> # NSFileHandle* midiFileHandle; # Byte* midiFileHeader; # Byte header[14]; # Byte** midiFileContent; # NSString* midiFileName; # int* positionCount; # int numberOfTracks; # int denominator; # int numerator; # bool recording; # int* timeOfLastEvent; # id instance; + (id)getInstance; + (void)prepareRecordingFor:(int)numberOfPlayers andDenominator:(int)newDenominator andNumerator:(int)newNumerator; + (void)triggeredNote:(NoteEvent*)noteEvent; + (void)endRecording; + (id) init; + (void) createHeader; + (void) createTracks; + (void) closeTracks; + (void) convertToVariableQuantity:(int)value toTrack:(int)number; + (void) addData:(Byte)data inTrack:(int)track; </pre>
BackendDummy
<pre> # NSMutableArray* chordVoicings; # NSMutableArray* basicProbabilities; # NSInvocation* propagateChordInfo; # NSInvocation* propagateSongInfo; # NSInvocation* propagateMIDIInfo; # NSInvocation* propagatePlayerInfo; # int meterDenominator; # int meterNumerator; # int numberOfBars; # int numberOfChords; # int numberOfPlayers; # bool songLoaded; # id instance; - CJFramework framework; + (id)getInstance; + (bool)openSong:(NSString*)fileName forClient:(id)client; + (void)prepareForClient:(id)client withNumberOfPlayers:(int)players; + (void)triggeredNote:(NoteEvent*)noteEvent; + (void)chordChangeNumber:(int)number; - (id)init; - (void)setAllProbabilities; - (void)setProbabilitiesForPlayer:(int)number; - (NSArray*)convertMidiTrack:(jdkmidi::MIDIMultiTrack*)track; </pre>

Figure C.8: The *Singleton* classes.

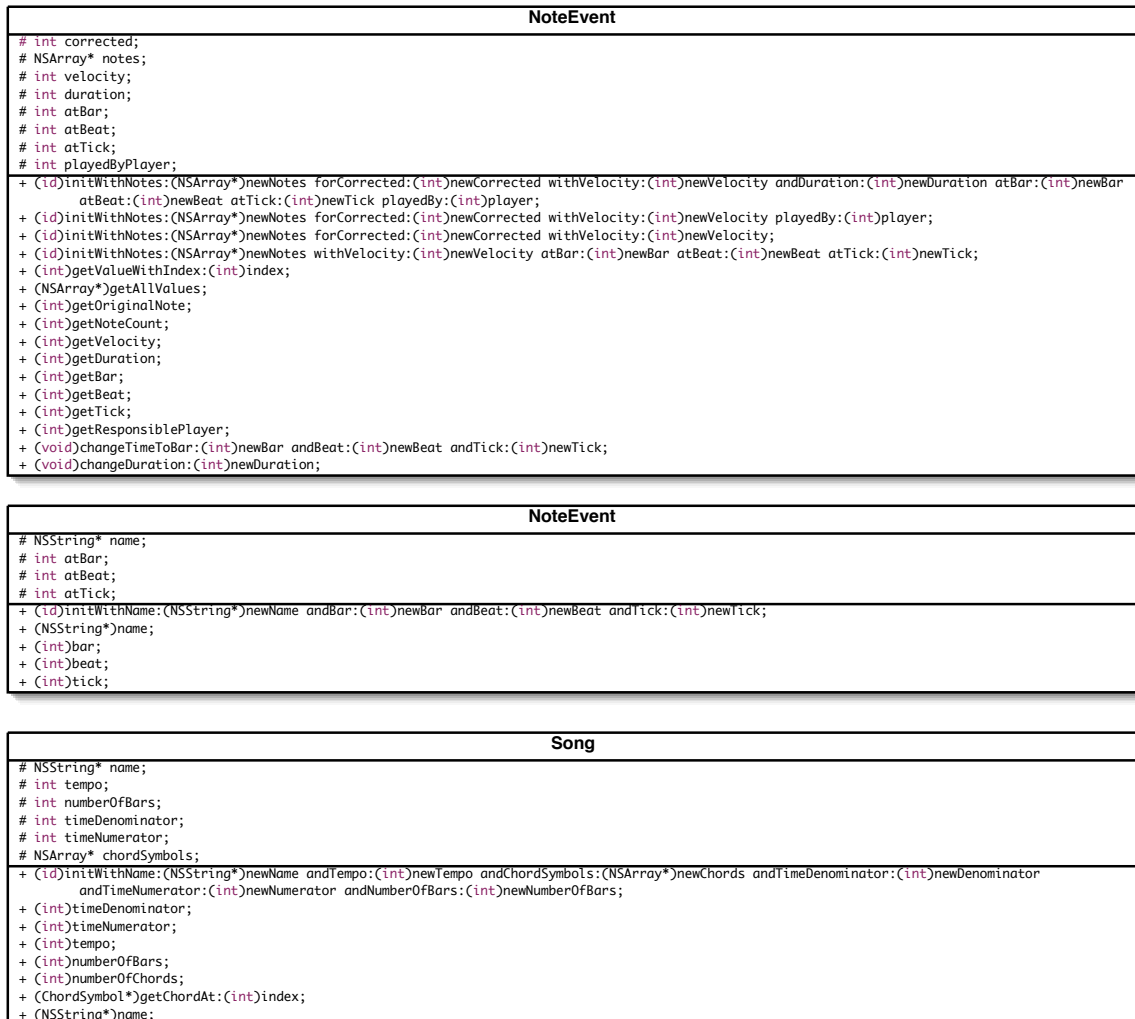


Figure C.9: The classes encapsulating data.

D. *coJIVE* in Use

This appendix will present a walkthrough of the final version of *coJIVE*. It will describe step by step how the system is set up, and how it behaves throughout simple session with two players. Figure D.1 shows two users playing with the system.



Figure D.1: Two users playing with *coJIVE*.

A MIDI keyboard and the batons are connected to the computer that runs the application. One LED is provided for each of the instruments. The application itself is controlled by the mouse or the computer keyboard.

D.1 Setting Up the System

Figure D.2 shows the system's initial screen with the depiction of a simple default song and no registered users. With a click of the "Options" button (the rightmost of the four buttons at the top left corner of the window), the drawer containing general options is opened. The number of users can be set by clicking the uppermost pop-up-button in the drawer and selecting the suitable number (figure D.3).

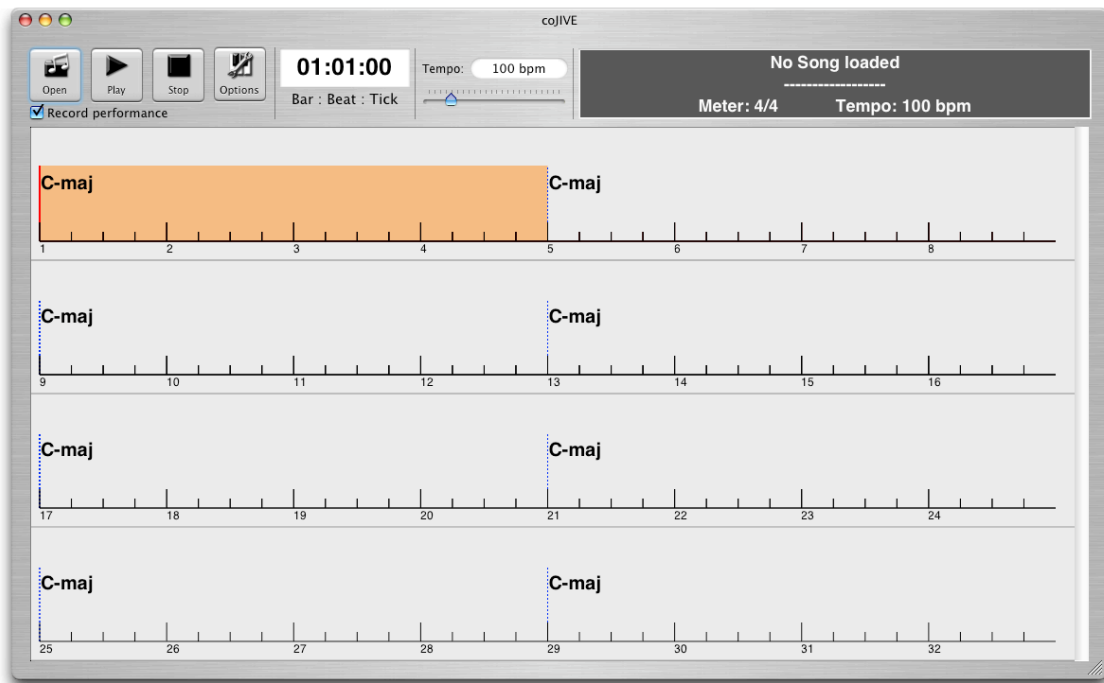


Figure D.2: The *coJIVE* window after start-up.

The system then creates the player fields and displays the options dialogue for each player one after another. Figure D.4 depicts this dialogue, in which the appropriate MIDI port and the type of the interface must be selected in the two pop-up-buttons in the upper box. In addition, the respective player’s skill needs to be set in this dialogue; the lower box is therefore separated into two parts, the upper one providing a simple way by using some default settings (accessible again via a pop-up-button), while the lower one presents four sliders to allow detailed adjustment to the user. By clicking on the “Ok” button, the settings are applied, and the dialogue is closed (the “Cancel” button discards the changes made to the settings and then close the dialogue). A user can later on access her options dialogue again by clicking the “Settings” button in the corresponding player field.

The second pop-up-button in the general options drawer allows the selection of a MIDI device to send all notes to (i.e., software synthesizer) as depicted by figure D.5. By clicking on the “Close OptionsDrawer” button, the drawer can be closed, which is also feasible by clicking the “Options” button again.

The player fields are arranged in another drawer at the bottom of the main window. Besides the “Settings” button, a field displaying the player’s current role (“Soloing” or “Accompanying”), the field for feedback and two additional buttons are given in the player fields; the “Detach” button can be used to reallocate the player field to a separate window — in that case the button is relabelled to “Attach” and can be used to reattach the field to the drawer —, and the “Close” button allows a user to leave the system (figure D.6).

Before the session can be started, the song to play needs to be selected. Therefore, the “Open” button is clicked (the leftmost of the four buttons at the top left corner of the window), which opens a file open dialogue depicted in figure D.7. If a song

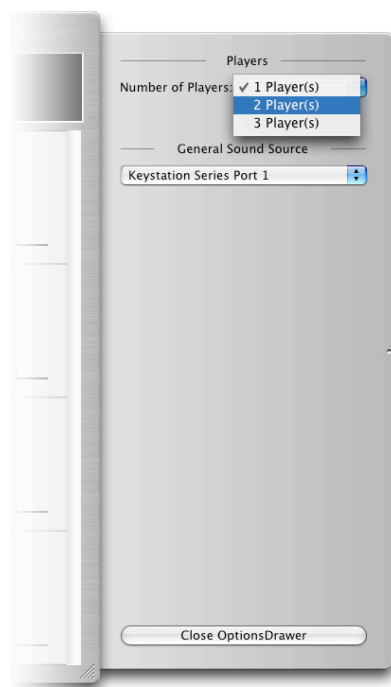


Figure D.3: Adjusting the number of participating players.

file is selected and the “Ok” button in the file dialogue is clicked, the song is loaded, its structure is shown in the lead sheet and further information is depicted in the display in the top right corner of the window.

D.2 The Session

Figure D.8 shows how a session is started by clicking the “Play” button that is situated to the right of the “Open” button. The system starts to play the accompaniment and the main theme of the chosen song. The player fields both display “Accompanying” to be the role of the players, and the LEDs attached to the instruments are turned of. While playing, the cursor moves over the lead sheet to indicate the current position in the song, and the current chord is always highlighted.

A few bars before the cursor reaches the end of the song structure (and the end of the main theme), a countdown to the upcoming solo of the first player is displayed in his field, and the LED attached to his instrument blinks (figure D.9). Once the end of the song structure is reached, the cursor jumps back to the beginning, and the playback of the theme is stopped. The field of the first player displays “Soloing” and his LED stays lit (figure D.10).

After between 30 seconds and one minute, the second player’s field displays a countdown, and her LED blinks in alteration with the first player’s LED. As soon as the countdown has finished, the second player’s field displays “Soloing”, while the first player’s field is set back to “Accompanying” (figure D.11). This change of roles occurs over again after the time-span mentioned above.

Throughout the session, the system displays feedback on the players’ performances in their fields. The field of the keyboard player depicts a keyboard, in which those

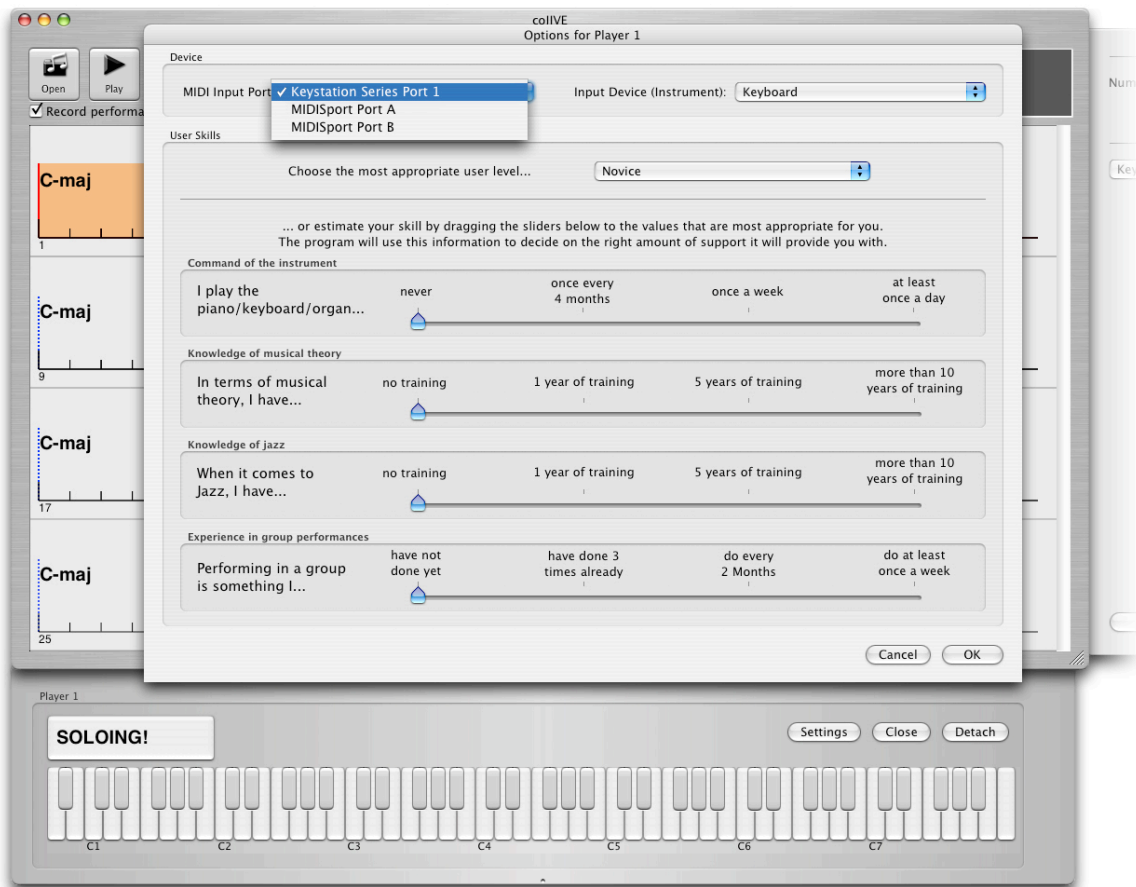


Figure D.4: The options dialogue for the players' settings.

those keys are highlighted that correspond to the currently sounding notes. The keys that the player is currently pressing are further marked by an “O” to depict the touch of a finger. Notes played with the batons are represented by green circles in a black rectangle. The size of these circles depict the applied velocity. Only two circles are visible at a time, corresponding to the number of batons available. Figure D.12 shows the keyboard and the batons in use, while figure D.13 depicts the corresponding two methods of displaying feedback.

The session can be ended by clicking the “Stop” button. This button is situated to the right of the “Play” button. If the “Stop” button is clicked again, the cursor jumps back to the beginning of the song's structure.

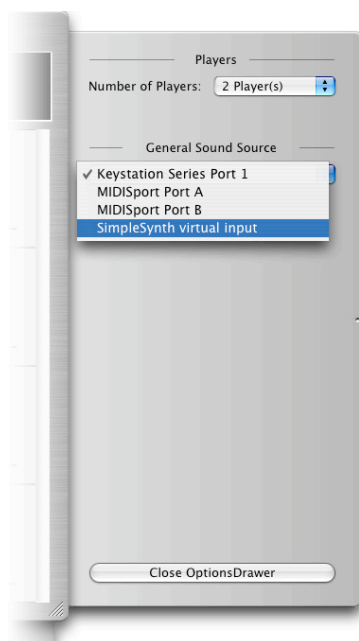


Figure D.5: Adjusting the general sound source.

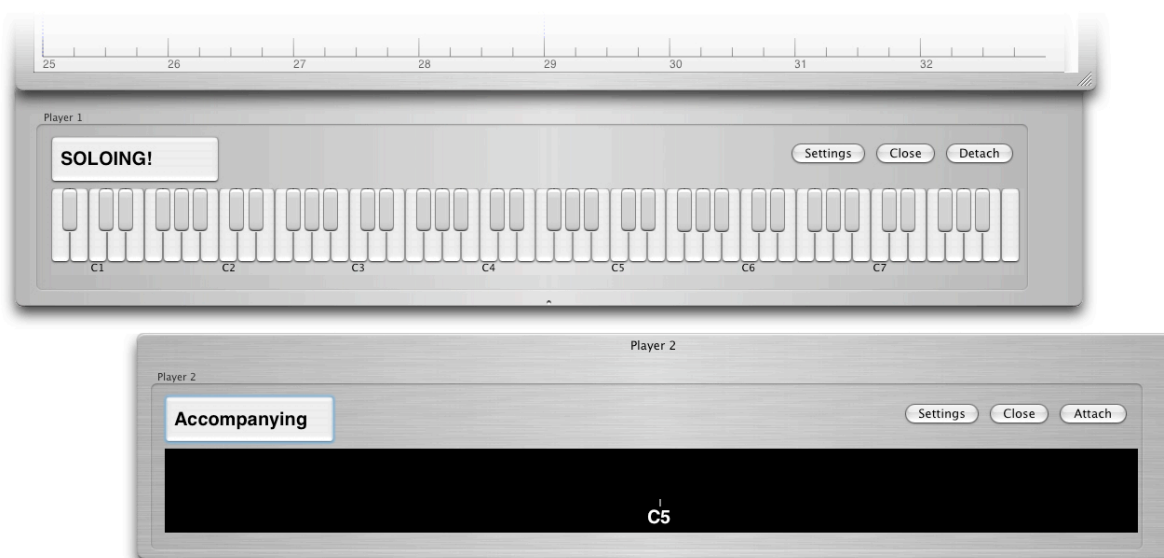


Figure D.6: The player fields on the drawer and on a separate window.

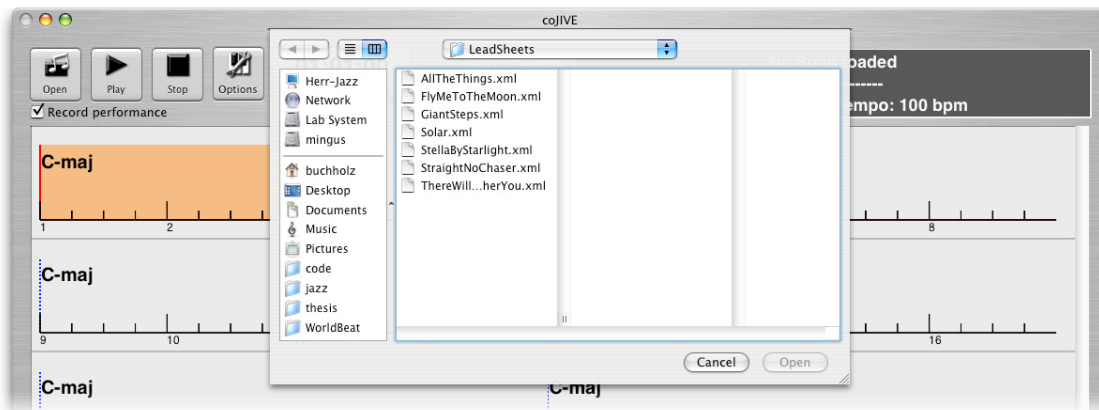


Figure D.7: The dialogue for choosing a song file.

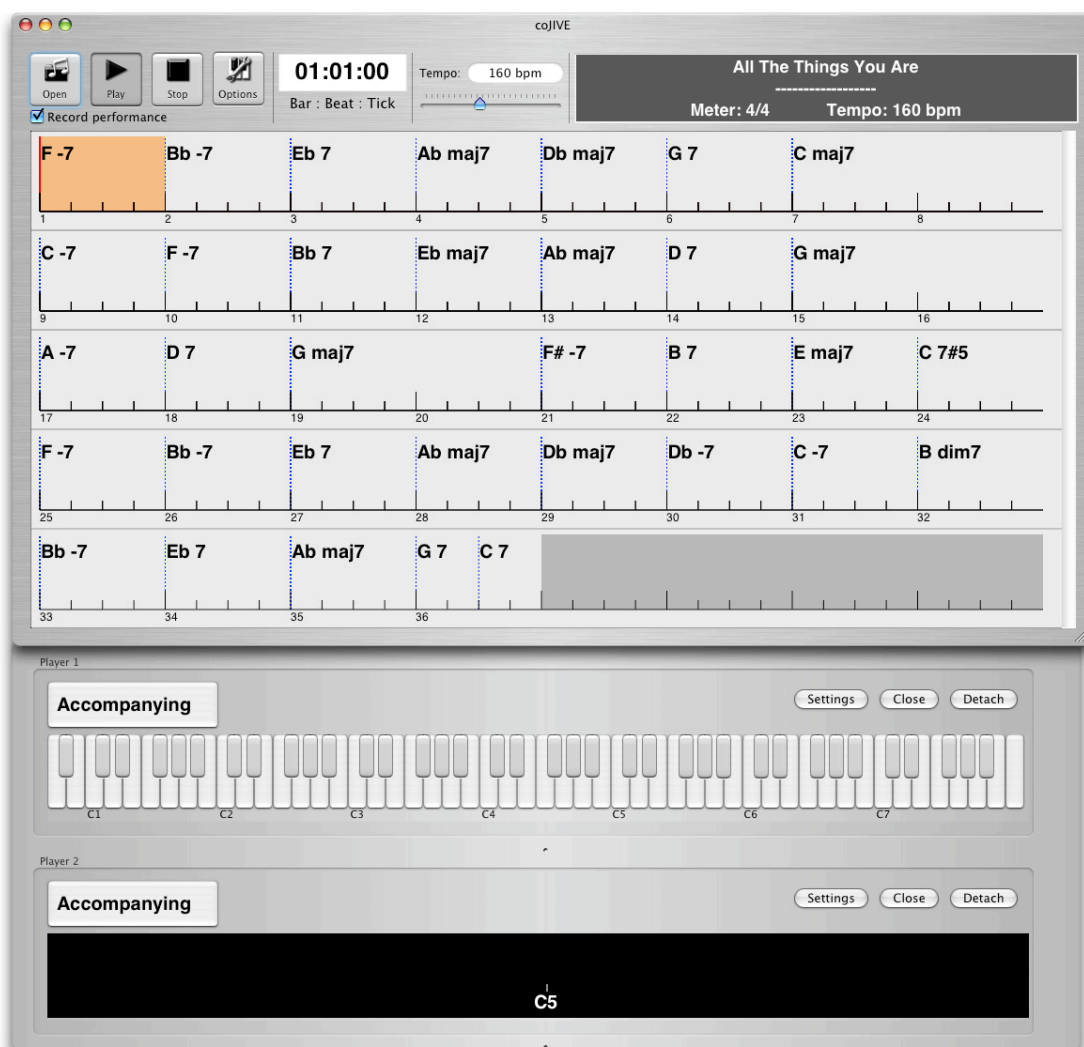


Figure D.8: The application on starting the session.

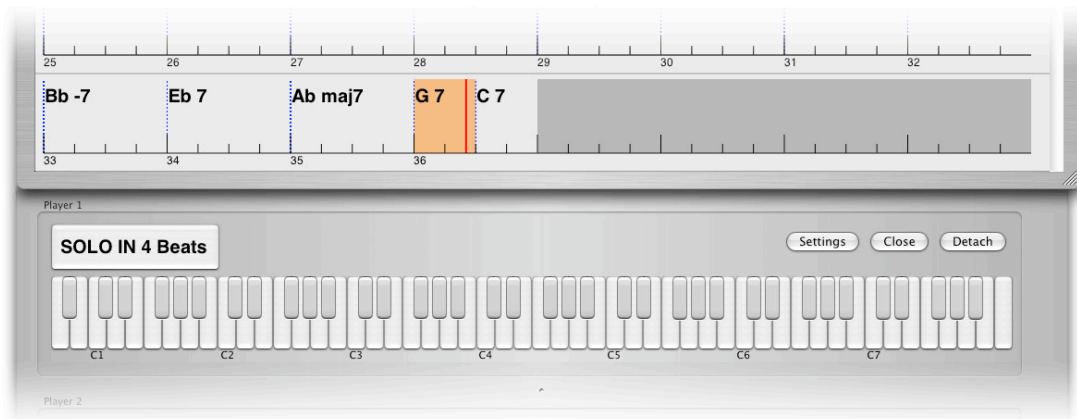


Figure D.9: The countdown announcing an upcoming solo.

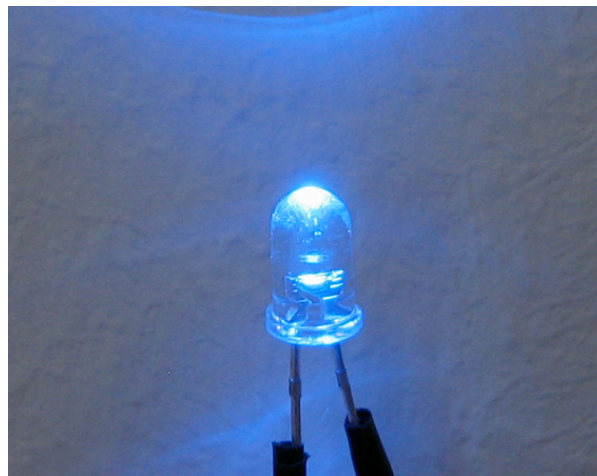


Figure D.10: An LED indicates an upcoming solo change by blinking.

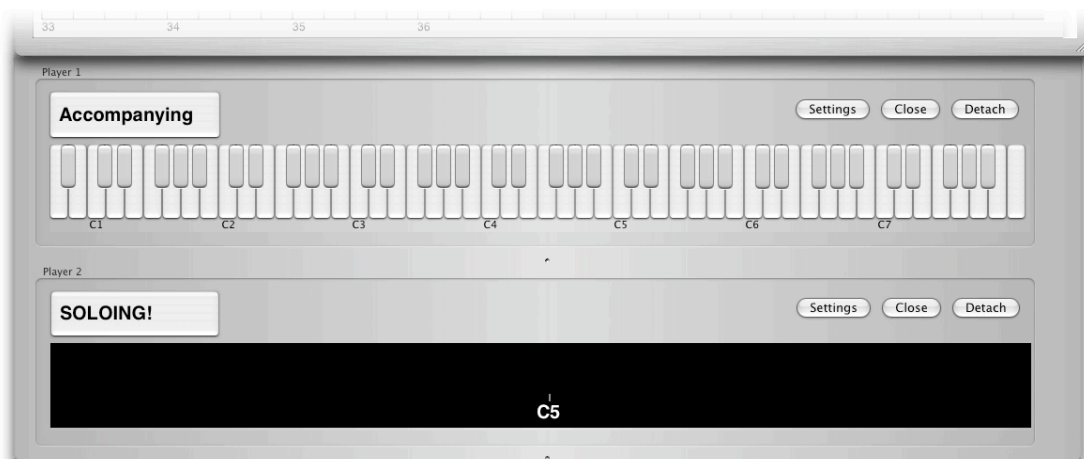


Figure D.11: The player fields depicting the roles after the change.



Figure D.12: The two musical interfaces in use.

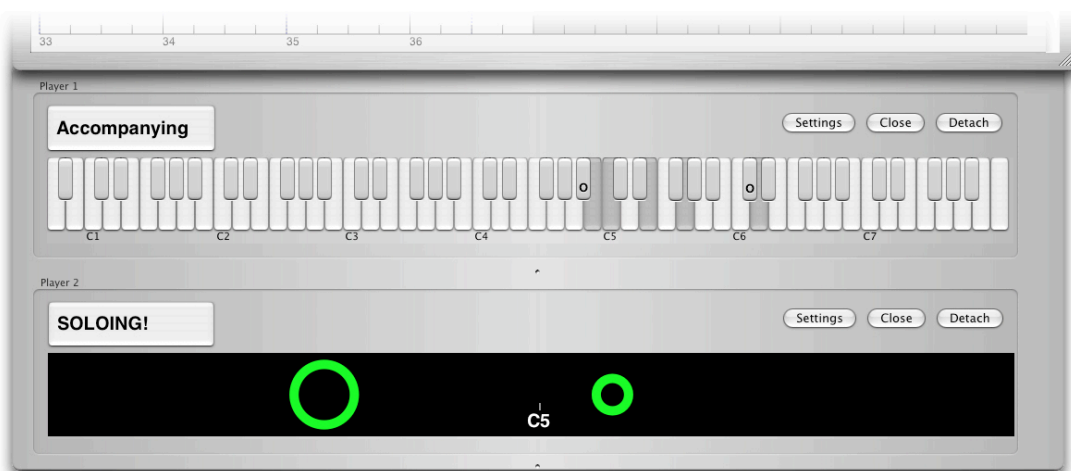


Figure D.13: The player fields depicting feedback on the performances.

References

- Apple Computer Inc. Human Interface Design Principles. <http://developer.apple.com/documentation/mac/HIGuidelines/HIGuidelines-15.html>, July 1996.
- Apple Computer Inc. Audio and MIDI on Mac OS X. <http://developer.apple.com/audio/pdf/coreaudio.pdf>, May 2001a.
- Apple Computer Inc. Cocoa. <http://developer.apple.com/cocoa/>, 2001b.
- Apple Computer Inc. Xcode. <http://developer.apple.com/tools/xcode/index.html>, 2003.
- M. Balaban and S. Irish. Automatic Jazz Accompaniment Computation: An Open Advise-based Approach. In *Proceedings of the ICCSSE '96 Israeli Conference on Computer-Based Systems and Software Engineering*, pages 67–76. IEEE, Washington, June 1996.
- F. B. Baraldi and A. Roda. Expressive Content Analysis of Musical Gesture: An Experiment on Piano Improvisation. Technical report, Department of Information Engineering, University of Padua, Padova, Italy, 2001.
- J. A. Biles. GenJam: A Genetic Algorithm for Generating Jazz Solos. In *Proceedings of the ICMC '94 International Computer Music Conference*, Aarhus, September 1994. International Computer Music Association.
- Jan Borchers. WorldBeat: Designing A Baton-Based Interface for an Interactive Music Exhibit. In *Proceedings of the ACM CHI'97 International Conference on Human Factors in Computing Systems (Atlanta, Georgia)*, pages 131–138. ACM, New York, March 1997.
- Jan Borchers. *A Pattern Approach to Interaction Design*. Wiley Series in Software Design Patterns. John Wiley & Sons Ltd, Chichestr, England, 2001.
- Don Buchla. Buchla Lightning System II. <http://www.buchla.com/lightning/index.html>, 1995.
- Casio, Inc. Light Guided Instruments — LK-90TV. <http://www.casio-europe.com/de/emi/lightguided/lk90tv/>, 2004.
- Cycling '74. Max/msp. <http://www.cycling74.com/products/maxmsp.html>, 2004.
- Roger B. Dannenberg and Jakob J. Bloch. Real-Time Accompaniment of Polyphonic Keyboard Performance. In *Proceedings of the 1985 International Computer Music Conference*, pages 279–290, February 1985.

- Judy Franklin. Multi-Phase Learning for Jazz Improvisation and Interaction. In *Proceedings of the Eighth Biennial Symposium on Arts and Technology*, March 2001.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Boston, 1995.
- Jean-Gabriel Ganascia, Geber Ramalho, and Pierre-Yves Rolland. An Artificially Intelligent Jazz Performer. *Journal of New Music Research*, 28(2), 1999.
- Dr. Peter Gannon. Band-in-a-box. <http://www.pgmusic.com/band.htm>, 1989.
- Jeff Glatt. Standard MIDI File Format. <http://www.borg.com/~jglatt/tech/midifile.htm>, 2001.
- M. Goto, I. Hidaka, H. Matsumoto, Y. Kuroda, and Y. Muraoka. A Jazz Session System for Interplay among All Players. In *Proceedings of the ICMC '96 International Computer Music Conference*, pages 346–349, August 1996.
- M. Grachten. JIG - Jazz Improvisation Generator. In *Proceedings of the MOSART Workshop on Current Research Directions in Computer Music*, pages 1–6, November 2001.
- K. Ishida, T. Kitahara, and M. Takeda. ism: Improvisation Supporting System based on Melody Correction. In *Proceedings of NIME '04 Conference on new Interfaces for Musical Expression*, pages 177–180. IEEE, Washington, November 2004.
- Axel Jungbluth. *Jazz Harmonielehre, Funktionsharmonik und Modalität*. Schott, Mainz, Germany, 1981.
- Jonathan Klein. A Pattern-based Software Framework for Computer Aided Jazz Improvisation. Diploma thesis, Rheinisch-Westfälische Technische Hochschule, May 2005.
- J. D. Koftinoff. C++ MIDI Library — libjdkmidi. http://www.jdkoftinoff.com/main/Free_Projects/C++_MIDI_Library/, 1986-2004.
- Making Things LLC. Teleo. <http://www.makingthings.com/teleo.htm>, 2002.
- Y. Nagashima, T. Hara, T. Kimura, and Y. Nishibori. Global Delayed Session Music - New Improvisational Music with Network Latency. In *Proceedings of the ICMC '92 International Computer Music Conference*, 1992.
- K. Nishimoto, C. Oshima, and Y. Miyagawa. Why always versatile? Dynamically Customizable Musical Instruments Facilitate Expressive Performances. In *Proceedings of NIME '03 Conference on new Interfaces for Musical Expression*, pages 164–169. McGill University, Montreal, May 2003.
- 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.

- C. Oshima, K. Nishimoto, and Y. Miyagawa. Coloring-In Piano: A Piano that allows a Performer to Concentrate on Musical Expression. In *Proceedings of the ICMPC '02 International Conference on Music Perception and Cognition*, pages 707–710, October 2002a.
- C. Oshima, K. Nishimoto, Y. Miyagawa, and T. Shirosaki. A Concept to Facilitate Musical Expression. In *Proceedings of the Conference on Creativity and Cognition*, pages 111–117. ACM, New York, October 2002b.
- François Pachet. Interacting with a Musical Learning System: The Continuator. In *Proceedings of the Second International Conference on Music and Artificial Intelligence*, pages 119–132, London, 2002. Springer Verlag.
- Roland. Juno D Synthesizer. <http://www.roland.com/products/en/JUNO-D/>, 2004.
- Marc Sabatella. A Jazz Improvisation Primer. <http://www.outsideshore.com/primer/primer/>, 1992-2000.
- Chuck Sher, editor. *The New Real Book*. Sher Music Co., Petaluma, CA, 1988.
- The Omni Group. Omnigraffle. <http://www.omnigroup.com/applications/omnigraffle/>, 2000.
- Viscount. Viscount Viva and Viva X Portable Pianos. <http://www.viscountus.com/pianos/Viva.htm>, 2003.
- W. F. Walker. A Computer Participant in Musical Improvisation. In *Proceedings of the ACM CHI'97 International Conference on Human Factors in Computing Systems (Atlanta, Georgia)*, pages 123–130. ACM, New York, March 1997.
- Michael Welzl. Netmusic: Echtzeitfähige konzepte und systeme für den telekooperativen austausch musikalischer information. Diploma thesis, Johannes-Kepler-Universität, Linz, Austria, December 1992.
- Pete Yandell. Simplesynth. <http://www.pete.yandell.com/software/simplesynth/Read%20Me.html>, 1995.

Glossary

Accompaniment

The performance of the drum and bass players as well as musicians other the improviser — the accompaniment — creates a rhythmic and harmonic basis and is necessary to build an improvisation upon. 2

Avoid Notes

Although a scale was determined by an analysis of a song's chord progression to be suitable for a certain chord, not all notes inside this scale are necessarily consonant in the context of that chord. Notes that are dissonant despite being a member of the appropriate scale are called avoid notes. Jazz musicians usually try to avoid them. 34

Bar

A measure that is used to divide a certain time in a musical song or performance into smaller sections. 26

Beat

A bar can be divided into beats, smaller units of time in music. The metre of a song states, in how many beats a bar can be divided (e.g., $\frac{4}{4}$ means four beats per bar). Usually, the tempo of a song is given as beats per minute. 26

Chord

A set of notes that, if played together, form a certain harmony is called chord. In jazz, chords are deployed to define a specific harmonic context for a certain part of the song's structure. 2

Chord Note

A note that is a member of a certain chord. 9

Chord Symbol

The textual notation of a certain chord. 2

Chorus

The entire structure of a song is called chorus. The length of a solo is often measured in the number of choruses it spans. 26

Comping

The activity of taking part in the accompaniment is called comping. 2

Consonance

If a certain note is perceived to be melodic in a specific harmonic context, it is called consonant. 2

DIA Cycle

A DIA cycle is one self contained part of a user-centred software design process. It is divided into three phases that all contribute the first letter of their name to the term DIA: in the design phase, the basic concepts and user interfaces are stated and laid out, followed by the implementation phase that contains the actual generation of the system. Finally, the system is evaluated in the analysis phase often involving tests with potential future users. Since the results of this analysis are used for further development in the subsequent cycles, the user's wishes are a main criteria in this kind of software development process, and bugs as well as usability issues can be addressed very early in the development. 4

Dissonance

Dissonance is the opposite of consonance. 2

Harmonic Context

The harmonic context defines what notes sound melodic in that particular situation and what notes do not. Over the course of a song, the harmonic context changes very often and is mostly bound to the underlying chord. 2

Jazz Standards

The term jazz standards describes a set of several hundred jazz songs that are commonly used in jazz sessions. 56

Lead Sheet

The lead sheet usually is a piece of paper that contains the structure of a given song, further information like tempo and meter, as well as the notation of its main melody. The structure is described by a progression of chord symbols. 2

Metre

The metre of a song defines its rhythmic structure and is usually given in the form of a fraction. Its denominator describes the measure that a bar is divided into (e.g., 4 stands for quarter notes), while the numerator states how many of these measures make up a bar. 2

MIDI

The Musical Instruments Digital Interface is described in the second appendix. 3

Model View Controller

A software architectural template that is mainly concerned with separating the data and logic from its depiction in the user interface. Data and logic are encapsulated in a model class that is disconnected from the depicting view

class by an additional controller class. The advantage of this arrangement is that view and model can be changed or extended without the need to change the respective other part. Also, several views can be incorporated for one model. 67

Object Adapter

This design pattern describes a solution for the problem that a given class must be adapted to another interface. This is done with the use of an adapter class; it references an object of the class that needs to be adopted and implements the interface by forwarding its calls to the referenced object. 70

Object-Orientation

A style of designing, implementing, and maintaining software systems. The name is based on the objects, small modules encapsulating data and related functionality, that make up a software system developed under the object-oriented paradigm. An object's structure and behaviour is defined in its class, a template, which can inherit attributes from other classes, and it can hand down its attributes to other classes as well. 56

Observer

A design pattern that allows an object to register with another object to be informed on certain events. Therefore, it must implement one or more functions that the subject (the observed object) can call, whenever the respective event occurs. 68

Octave

An octave is the smallest interval in music that includes at least one note from every pitch class. Accordingly, an octave contains twelve notes, which are also called semitones, dividing the octave into twelve steps. The interval is also defined to be the distance between the frequencies f and $2f$. 19

Pitch Class

Notes with the same identifier (e.g., C, D, E) are collected in a pitch class. The theory of Western music knows of twelve pitch class: C, C \sharp (also: D \flat), D, D \sharp (also: E \flat), E, F, F \sharp (also: G \flat), G, G \sharp (also: A \flat), A, A \sharp (also: B \flat), and B. 118

Scale

A scale is a set of notes all described by a degree (e.g., the root note is degree I) that is used to classify notes over a part of a song as being suitable or unsuitable. If a scale is given in the structure of a song, musicians tend to mostly use the notes inside the scale since notes outside the scale mostly sound disharmonious in the given context. While a song usually sticks to one particular scale in classical music, jazz pieces often have no pre-defined scales; jazz musicians analyse the chord structure of the song and decide on what scale to use over what chord usually leading to numerous changes between different scales in the performance of the song. 2

Singleton

This design pattern describes a class that has two main attributes: at most one instance of this class can exist at any point in time, and this instance is available throughout the application. 69

Solo

The part of a performance, in which one musician is at the centre, while the other musicians accompany him is called solo. In most musical styles, musicians use solos to create special performances emphasising their personal abilities. 2

Template Method

The template method design pattern describing a method defined by a super class that defines an algorithm by calling several other functions or methods. These other functions are implemented by the subclass such that each subclass implements another variant of the algorithm. 81

Theme

In jazz, the term theme is usually used for a song's pre-composed main melody. 2

Trading Fours

The collaborative mode, in which musicians take turns in soloing for four bars, is called trading fours. In this specific mode, the musicians draw much more from the solo of their predecessor compared to usual alternating solos. This behaviour is often compared to a conversation. 26

Voicing

A specific arrangement of the notes making up a chord. 3

Index

- Accompaniment, 2
- AccompanimentOutput, 71
- AdvancedProcessor, 71
- Architectural Design Patterns, 67
- Avoid Note, 34

- BackendDummy, 70
- Band-in-a-Box, 13
- Bar, 26
- BassOutput, 71
- BatonOutput, 70
- Beat, 26
- Buchla, 4

- C++, 56
- Casio LK-90TV, 22
- Category, 76
- CHIME, 10
- Chord, 2
- Chord Note, 9
- Chord Symbol, 2
- ChordSymbol (class), 72
- Chorus, 26
- CiP, 18
- CJFramework, 81
- CJSong, 81
- Cocoa, 56
- Collaborative Skill, 59
- Coloring-In Piano, 18
- Command of the Instrument, 59
- Comping, 2
- Confine to Scale, 33
- Consonance, 2
- Continuator, 11
- ControlChange, 73
- CoreMIDI, 72

- DIA Cycle, 4
- Dissonance, 2
- DrumOutput, 71

- Experiences in Group Performances,
59

- Gang of Four, 67
- GenJam, 8
- Global Delayed Session Music, 15

- Harmonic Context, 2
- Help Triggering Compound
Structures, 34
- Human Interface Design Principles, 60

- Improvisation Supporting System, 20
- ImprovisationBuilder, 9
- Improviser, 2
- Improvisession, 15
- InputProcessor, 71
- Instrument Skill, 59
- ism, 20

- Jazz Improvisation, 1
- Jazz Skill, 94
- Jazz Standards, 56
- JIG, 8
- JOINME, 12

- KeyboardInput, 70
- Knowledge of Music Theory, 59

- Lead Sheet, 2
- LeadSheetView, 68

- MainController, 68
- MainModel, 68
- MainView, 68
- Max/MSP, 32
- Melody-Based key Illumination, 22
- Metre, 2
- MIDI, 3
- MIDIClientRef, 81
- MIDIEndpointRef, 82
- MIDIInput, 70
- MIDIio, 70
- MIDIOutput, 70
- MIDIPacketList, 82
- Model View Controller, 67

- Musical Design Patterns, 17
- Neighbourhood Density, 65
- NeXTSTEP, 73
- NIB, 74
- Nishimoto Design Principle, 28
- NoteOff, 31
- NoteOn, 31
- NoviceProcessor, 71
- NSArray, 74
- NSButton, 74
- NSData, 74
- NSDictionary, 71
- NSInvocation, 68
- NSIterator, 74
- NSLock, 75
- NSMutableArray, 76
- NSNumber, 74
- NSObject, 73
- NSRunLoop, 79
- NSScrollView, 78
- NSText, 74
- NSTextField, 74
- NSThread, 75
- NSTimer, 79
- NSView, 68
- NSWindow, 68

- Object Adapter, 70
- Object-Orientation, 56
- Objective-C, 67
- Observer, 68
- Octave, 19
- Offer Automatic Structuring, 35
- One-Key Chord Triggering, 21

- Pitch Class, 118
- PlaybackEngine, 69
- PlayerController, 68
- PlayerModel, 68
- PlayerView, 68
- Prohibit Avoid Notes, 34

- Quantisation, 38

- RecordingEngine, 69
- Restrict Expressive Parameters, 36
- RhyMe, 19
- Roland Juno D Synthesizer, 21

- Scale, 2

- Selector, 74
- SimpleSynth, 75
- Singleton, 69
- Solo, 2
- Song (class), 71

- Teleo, 96
- Template Method, 81
- Temporal Density, 67
- Theme, 2
- Theory Skill, 59
- Trading Fours, 26

- Velocity, 31
- Velocity Bending Factor, 85
- VirJa, 14
- Viscount Viva, 73
- Voicing, 3

- Walkthrough, 133
- WorldBeat, 17

- XCode, 74