Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Lukáš Kopenec

Notový editor

Music Notation Editor

Katedra softwarového inženýrství

Vedoucí: RNDr. David Bednárek

Studijní program: Informatika, Obecná informatika

2007

Rád bych na tomto místě poděkoval všem, kteří mi s mou prací pomáhali nebo mi vytvářeli podmínky pro to, aby mohla vzniknout. Za první patří velký dík zejména mému vedoucímu RNDr. Davidu Bednárkovi pro ochotu, s jakou mi poskytoval vždy velice užitečné konzultace a pro usměrnění mých, místy příliš divokých a vzletných, nápadů. Dále pak Petru Volnému a Vítu Kubánkovi, aktivním hudebníkům, kteří se mnou probírali podobu editoru a v neposlední řadě kolegům, kteří mi párkrát pomohli vrhnout světlo na některé temné zákoutí mého kódu. Za druhé pak děkuji především svým rodičům, kteří mne po celou dobu podporovali a snažili se udělat vše pro to, abych svou práci psal v pohodlí a dobré náladě.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

Lukáš Kopenec

V Praze dne

Table of contents

Chapter I - Introduction 7
Chapter II – Small introduction to music notation 9
II.1 - Common western music notation system 9
II.2 - Basic notation symbols 9
II.2.1 Note, beats, tempo – The duration characteristic 10
II.2.2 Staff, clefs, key – The pitch characteristic 10
II.2.3 Parts – The timbre characteristic 11
II.2.4 System, Measure – The time characteristic 12
II.2.5 Dynamic marks – The strength characteristic 13
II.3 - Other notational symbols appearing in this work 13
II.3.1 Ledger lines 13
II.3.2 Barline types 13
II.3.3 Rests 14
II.3.4 Symbols attached to notes 14
Chapter III – Basic Concepts 15
III.1 - The initial reflections 15
III.2 - The CMNSymbol Class 16
III.3 - The top-level aggregation structure 16
Chapter IV – The internal representation 19
IV.1 - Down the aggregation hierarchy 19
IV.1.1 System and staff connectors 19
IV.1.2 Part and staves 21
IV.1.3 Measure and barlines 22
IV.1.4 Note 24
IV.1.5 Environment modifiers 26
IV.1.6 Attachments 26
IV.2 - The score class 28
IV.3 - The common symbols interface and active symbol service 29
IV.5 - The undo / redo actions 30
Chapter V – Discussions 32
V.1 – Notation symbols classification 32
V.2 – Action objects 33

Chapter VI – Algorithms used in the editor 36
VI.1 – Durational symbols insertion 36
VI.1.1 The note insertion 36
VI.1.2 Discussion 38
VI.2 – Measures insertion / removal 38
VI.2.1 Measures insertion 39
VI.2.2 Measures removal 39
VI.2.3 Changing measures per system attribute 39
VI.2.4 Moving associated objects 40
VI.2.5 Discussion 41
VI.4 – Symbols layout / drawing 42
VI.4.1 Drawing 42
VI.4.2 Symbols layout 42
VI.5 – Active symbol detection 44
Chapter VII – Conclusion 45
References 47

Název práce: Notový editor
Autor: Lukáš Kopenec
Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. David Bednárek
e-mail vedoucího: David.Bednarek@mff.cuni.cz

Abstrakt:
Cílem této práce bylo navrhnout a implementovat notový editor se základními editačními funkcemi, ke kterým bude v budoucnu snadné přidávat další až do rozsahu použitelného pro hudebníky. Editor umožňuje vkládat a editovat základní hudební symboly. Hotové notové záznamy lze ukládat a tisknout. Důraz je kladen především na WYSIWYG editaci. Na přiloženém CD jsou k dispozici všechny zdrojové kódy, dokumentace, instalační program a tento text ve formátu PDF.
Práce je v anglickém jazyce.

Klíčová slova: notový editor, noty, hudební notace, WYSIWYG

Title: Music notation editor
Author: Lukáš Kopenec
Department of software engineering
Supervisor: RNDr. David Bednárek
Supervisor's e-mail address: David.Bednarek@mff.cuni.cz

Abstract:
The goal of this work was to design and implement a musical notation editor with basic edit functions. The program is easily expandable and prepared for addition of new functions, so that after further work, it would be useable for musicians. The editor allow the insertion of core notation symbols, the final scores can be saved or printed. On the attached CD you will find all the sources, documentation, setup program and this text in the PDF format.
The thesis is written in English.

Keywords: music notation editor, notes, music notation, WYSIWYG

Chapter I - Introduction

As an active musician I often come across a necessity to write down some of my ideas or just rewrite or arrange an existing musical score. Handwriting is not a very comfortable way so I use computer notation software. Several products are presently available on the market, but they are generally very expensive. There are still some free engraver programs but their input interface is typically text-oriented and if you tried to find a good WYSIWYG music notation editor for a reasonable price, you would very probably fail.

The best software I have ever used is Encore from Passport Designs Inc. I share this opinion with numerous musicians who extensively use computer for the notation. However, it is a very old product and in consequence presents some compatibility issues with recent operating systems. Moreover the Passport Designs Inc. no longer exists and the product is now sold by Gvox. Nevertheless this company does not develop the program anymore and offers still the old version of Encore (from approximately 1995) with just a new splash screen.

All these facts led me to the decision to create my own editor which would meet all my requirements and which I could offer to the amateur musicians who cannot afford a commercial product.

After a detailed analysis of the problem and study of materials cited in references I found out that the development of a fully functional WYSIWYG editor is a too big task for a bachelor thesis. Therefore I decided to design and implement the core program with most important functions and leave the rest for my master thesis.

Thus the purpose of my work is to design and implement software, which will allow WYSIWYG input and editing of notes and rests. The program should conform to the rules published in [7]. It will provide automatic symbols alignment and usual editing facilities as undo / redo or clipboard support. Naturally the basic properties of the notational system have to be coded as well. This includes the key and time signatures, clefs, accidentals, augmentation dots and different barline types. The internal representation should have higher abstraction level than a purely graphically-oriented one. That means it should understand the relations between musical notation objects and store them appropriately. It should be able to perform some integrity checks, although I will try to delegate most of them to the external program. The reason for that is that I would like to keep the internal representation as general as possible in order to leave the door open for the functionality I left for my master thesis and especially for the uncommon practices that may appear in modern music notation which does not every time follow the rules as strict as the classical notation stated.

I consulted the form of the user interface with a few musicians and we agreed that the most suitable is the one of Encore. Therefore I will adopt it with some small modernizations as a dockable categorized toolbox in place of old pallets and a context menus support.

When I collected the materials for my thesis I found a very interesting work of Kai Lassfolk [4]. In his text he compares some existing notation products and provides a deep object-oriented analysis of the common western music notation system. Finally he offers a model for a new notation editor. I will base my program on his model, although I made several more or less important changes which I will describe in further chapters.

The rest of my thesis is organized as follows:

- In the Chapter II I offer a brief introduction to the common western music notation system, its basic symbols with their description and its most important rules.
- In the Chapter III I present the initial questions and problems I had to resolve before starting the internal notation representation development and finally I introduce its top-level part.
- Chapter IV is a description of the internal representation model. The principal goal of this chapter is to familiarize the reader with the model conception and the relations between its parts, so that he can easily follow my reflections and discussions developed in further chapters. I will also describe the changes I made to the original Lassfolk's model along with their motivation.
- Chapter V will be dedicated to discussions about alternative solutions of the model and the motivation I had to select the version described in previous chapter.
- Chapter VI will describe the algorithms used in the program along with the discussions about them.
- Chapter VII will bring the conclusion.

Chapter II – Small introduction to music notation

I will begin this work with a short dictionary of music terms and a brief introduction to common western music notation system and the symbols it uses.

II.1 - Common western music notation system

The first important term to define is the *western music notation system*. The music notation system is a graphical system used to encode music so that it can be interpreted and reconstructed by people. Western music notation system then encodes the western musical practice (common in Europe and United States) which differs in many ways from the oriental and other cultures (Chinese, Arabic, African etc.). According to Nelson Goodman and Kari Kurkela, music notation can be examined both semantically and syntactically as in case of natural languages, although it differs in many ways from them.

No official standards exist for the notation system. However the term *standard music notation* is used to denote a loosely restricted set of western music notation symbols and conventions. [4]

Finally the *common music notation* is defined as the standard music notation system originating in Europe in the early seventeenth century (Roads 1996: 708).

As a language, the common music notation is a highly complex system, more complex than any written natural language or even western mathematical notation (Roads 1996: 708). This complexity is due not only to the high number of different symbols but also to difficult rules that govern their coexistence. Moreover, as a graphical system, it is also a subject of rules that control its visual aspect. Entire books are dedicated to teaching of music writing and engraving (as for example Ivo Zelingr's work [7]) and entire chapters are devoted to their esthetical side. Many music publishers have developed their own layout conventions (often called "house-style"). Many music engravers have also developed their personal visual styles. Engraving and layout practices vary between individual publishers and engravers and they even disagree on the role and placement of common symbols such as barlines (Ross 1970, 151). The last important aspect is that musical practice and in consequence the notation system is subject to a very dynamic evolution. All the cited facts make the computer-based music notation a difficult and still open task.

II.2 - Basic notation symbols

The music composition can be regarded as a set of sound events – tones. The *tone* is a well defined term entirely described by 4 characteristics: Pitch, Strength, Duration and Timbre. The notation must encode these characteristics of all tones along with the fifth one which is the position in time (or the order of tones). Storing these five attributes would be roughly sufficient for a good interpreter to reconstruct the original composition. There are many other important aspects that have to be encoded which control the performance of group of tones or of the composition as a whole (e.g.: tempo, agogics,

slurs, etc.). But let's see for the beginning how the western music notation stores the mentioned basic characteristics.

II.2.1 Note, beats, tempo – The duration characteristic

The central element used in music notation is a note. The note corresponds to one tone and its shape encodes its duration. In western music notation, the term *beat* is used for a basic time unit. The beat alone does not have any particular duration. Its exact time measure is defined by the intermediate of *tempo* which tells how many beats belongs to one time unit (most often a minute is used). For example, the tempo may be given as 60 beats per minute which means that one beat durates for one second. The duration of the tone is stored in beats.

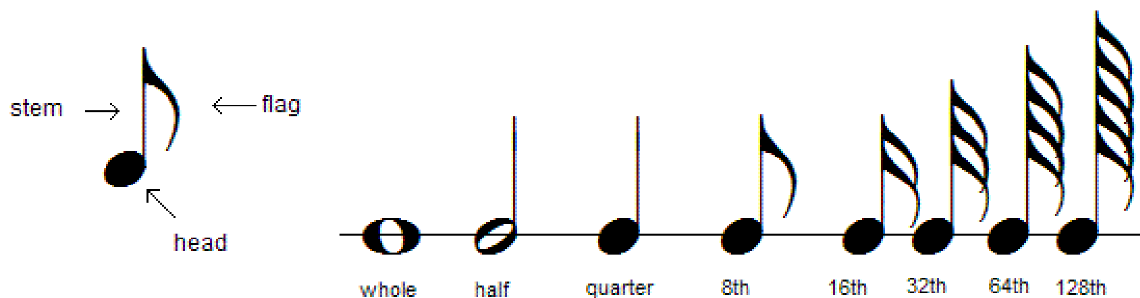

Figure II-1 Note and duration

For a note, the durations that are a power of two are used exclusively. Starting with *whole note* which is 4 beats long continuing to *half note* (2 beats), *quarter note* (1 beat), *eighth note* (2 in a beat) etc. Usually no shorter notes than 128th (32 in one beat) is used. If other duration (not a power of 2) has to be expressed, we can either *tie* more notes with the same pitch together (this is done by drawing an arc connecting their heads) which then sound as one tone or augment them by an *augmentation dot*(s). The augmentation dot is a small dot symbol drawn near the note head and changes its duration to one and half of the original value.

II.2.2 Staff, clefs, key – The pitch characteristic

To determine the pitch of the tones, the notes are placed on a *staff*. A staff is a set of horizontal lines – typically five and the notes are placed on the lines and in the spaces. The tone range used in western music is relatively large (approximately 100 tones). Hence, to be able to notate it on a small space of the 5-lines staff, we have to code the pitch in two steps. The position of the note on the staff indicates its relative pitch. The absolute highness is given by a *clef*. A set of notes without clefs does not have sense, we only know, which note is higher and which lower.

Figure II - 2 Staff and clefs

We use three basic types of clefs, each has the same name as the note it is determining by its position.

- F-clef indicates the position of f. Historically there were three types of F-clef, but today we use only one – bass clef on the fourth line.
- C-clef indicates the position of c'. Four were used in history, but only two are used nowadays and may be that only one will be used in the future: Tenor clef is placed on fourth line (and is used more and more rarely) and Alto clef is placed on third line.
- G-clef indicates the position of g'. Again, historically two types existed: French treble clef on the first line (which is not used any more) and Treble clef on second line which is the most often used today.

The cited music notation can only represent the old Church scales and moreover only those starting from c. To represent modern major and minor scales we have to introduce *accidentals*. The accidental is a mark placed before a note and valid for the *measure* (explained in II.2.4) containing this note or to the other accidental at the note with the same pitch. The *sharp* (#) augments the pitch by one semitone and *flat* (*b*) decreases the pitch by one semitone. There are also *double sharp* (*x*) which augments the pitch y one tone and *double flat* (*bb*) which decreases the pitch by one tone. The accidental can be canceled by a *natural* sign.

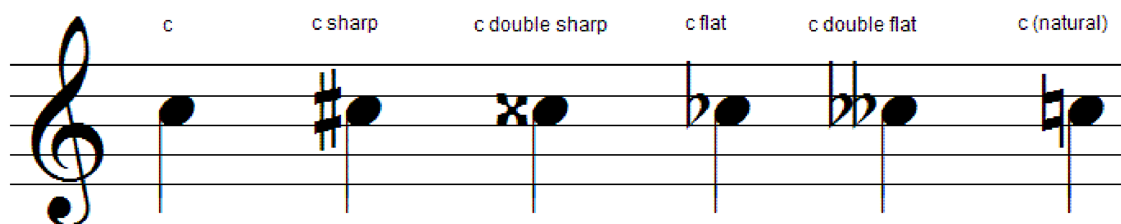

Figure II - 3 Accidentals

As I told, the accidentals have limited validity. To indicate that a composition is written in one concrete scale a *key signature* is used. The key signature is composed from accidentals placed in the beginning of the staff valid for the whole composition (or to the next different key signature). If you think for a while, you will find that the same tone can be expressed in more than one way when sing accidentals. For example C# and Db. This change is called *enharmonical*. In theory the tones are really equivalent. However in practice, this can lead to different tones due to the tuning system of the musical instruments.

II.2.3 Parts – The timbre characteristic

The timbre of the tone is made by the instrument which produces it. A flute produces a different sound than a piano or guitar, although they are all playing the same tone. The composition of instruments in the composition is also a very important characteristic. Western music notation typically uses *parts* to encode the instrument that is playing the specific set of notes.

As explained in previous paragraph, the notes are placed on the staves. Thus we can join the staves performed by the same instrument to groups called parts. To denote which instrument should play a specific part, the part has a name corresponding to the name of the instrument. The parts can also be grouped. This is typically done when their instruments belong to the same group (percussions, winds, etc.). To clearly denote the groups we can use *braces* and *brackets*. Braces are usually used to connect the staves of a piano part and the brackets denote a group of parts.

Figure II - 4 Brace and bracket

II.2.4 System, Measure – The time characteristic

The tones of the composition have heir fix order which has to be necessarily encoded in the notation. The time is basically represented in the horizontal position of the notes. The notes more on the right sound later than notes on the left. The notes which are above or below other notes sound concurrently.

A finer granularity has been introduced to represent other aspects of the performance. The notes are divided into *measures*. Measures are time sections of the composition where accented and non-accented notes regularly alter ([7]). Thus the measures can be distinguished by the accented notes when listening to the composition. To distinguish them visually, the barlines were created which separates them one from other. The measure has one other important characteristic – the *time signature*. It is represented by a quotient on the beginning of the measure whose nominator gives the count and denominator the value of basic beat. These two numbers determines how many notes we can put in one measure. For example a ¾ measure can contain three quarter notes (or six 8th notes or any combination whose total duration does not exceed 3 beats). The time signature is valid for all the following measures until another time signature appears.

The measures contain notes of all staves (all the notes in the same time section). Thus, they are not subordinates of staves. Not even of parts or the part groups. Their parent object is called a *system*. A system is a way to connect staves to indicate that they are performed in parallel ([4]). To clearly denote the system connects all its staves (parts) by a *systemic barline* which is a continuous barline drawn in the beginning of the system connecting all of its staves. So, the order of the tones is unambiguously given as follows: The composition is performed from page to page, from the topmost to the bottommost system, from the leftmost measure to the rightmost, from the leftmost note to the rightmost.

II.2.5 Dynamic marks – The strength characteristic

The last basic attribute of the tone to describe is its strength – it is roughly the loudness of the tone. It is represented by the *dynamic marks*. Basic marks are *ppp, pp, p, mp, mf, f, ff, fff* which are abbreviations of Italian expressions graduating the strengh from *piano pianissimo* – very silently, through *mezzo piano* – rather silently and *forte* – loudly to *forte fortissimo* – very loudly. These marks are placed below the staff (or between the staves of the part) near the notes where their validity begins and remain valid until another dynamic mark. However, these marks cannot represent a progressive change of strength. For this purpose we use *crescendo*, *decressendo* and *diminuendo*, which mean respectively: progressively more and more loudly, progressively decrease strength, progressively decrease strength to silence (or near the silence).

II.3 - Other notational symbols appearing in this work

In the previous section I described the basic characteristics of the music which have to be necessarily encoded in the notation in order to be able to reconstruct the composition. Nevertheless, the described symbols are far away of being sufficient for a good music description. Much other finesse appears in the practice and so many other symbols must appear in the notation system. They are typically attached to some of the "core symbols" described above. I will describe the most important which are modeled in my editor.

II.3.1 Ledger lines

Beside the clef and key signature which was already described the staff often needs another helper object to represent clearly the pitch of the notes – the *ledger lines*. As I told the notes are placed on and between its lines. However 5 lines do not offer enough space and to enlarge this space the notes can also be placed above or below the staff. To distinguish easily how far the note is from the last regular line, we draw the note on small lines in the same distance as regular staff lines.

II.3.2 Barline types

I have already told that barlines are visual separation of measures. However, there are several types of barlines each having different interpretation. The simplest is the *single barline*, it has only the basic separator function. Then we use a *double barline* which itself does not mean anything for the interpretation, but typically indicates that some important change occurs between the two measures it separates (such as key or time signature change). *Repetition barline* appears in pairs: opening and closing. It is draw as two lines one (the outer one) thicker, second thin and there are two dots (one in second and one in third staff space). The section between the opening and closing barline is repeated. The last type is a *final barline* which denotes the end of the composition. It has the inner line thin and outer thick.

II.3.3 Rests

I have talked about the way to represent tones, but we need also some symbols to represent silences – the *rests*. The duration of rests follows the same rules as in case of notes. Their symbols are on the figure below.

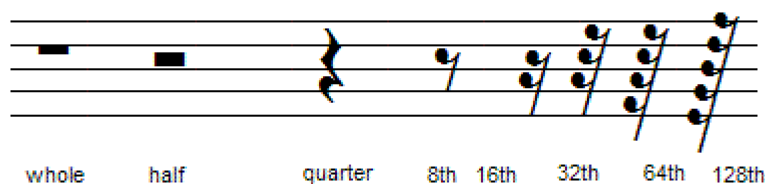

Figure II - 5 Rests

II.3.4 Symbols attached to notes

The greatest number of possible attachments holds, naturally, the note. The music notation offers symbols that can affect a single note (e.g. *ornament, fermata*), a chord which is a group of notes stacked one above the other having all the equal duration (e.g. *arpeggio*) or the entire group of notes (e.g. *tie, slur, crescendo*). The notes can be influenced in many ways: articulation, duration, strength…

There is also a special symbol attached to group of notes having the flags (8th note and shorter) – *beam*. Precisely the beam replaces the original flags and is used mostly for either clearly denoting the beats (the notes belonging to the same beat are beamed) or special rhythmical group of notes called *tuplets*. The tuplets are special rhythmical patterns which can be notated only by subdividing a standard duration note. The most often used are *triplets*. They are created by subdividing a nearest longer duration to three notes of the shorter duration (e.g. normally, there are two 8th notes in one beat, the triplet signify that three notes (of equal duration) will be played).

Figure II - 6 Beams and tuplets

The symbols that affect the group of notes are modeled by the *connector* class (see section IV.1.6). The most important are slurs and ties. Ties were already described. Slurs are also drawn as an arc joining notes of different pitch and signifies that the notes should be played *legato*, it means continuously without interruptions in between. *Ottava* is used to indicate that the note (or group of notes) will be played one octave above or below the tone which is notated. It is used mostly for ease of reading when too many ledger lines would have to be used. *Ending* is used with repetitions to indicate that after a given repetition the composition will continue from a different point. *Pedal lines* and *pedal marks* are used for piano parts to indicate when to hold and release the left piano pedal.

Chapter III – Basic Concepts

In this chapter I will present the basic concepts and architecture of the editor internal representation. I will initially discuss the questions and problems I had to resolve before starting the creation of the model. Then I will introduce the top-level part of the representation whose description I will develop in further chapters.

When I collected materials for my thesis I came across a very interesting work of Kai Lassfolk [4] which provides a detailed object-oriented analysis of western music notation and offers a scheme of a notation model that could be adopted in new notation editors. I based my internal representation mostly on the Lassfolk's proposition, although I made several more or less important changes. In the following chapters I will present the final model with the explanation of alterations I made along with their motivation.

III.1 - The initial reflections

The first important question to consider before starting a development of an abstract representation of a real-world problem is its level of abstraction and the overall approach to the problem. For a musical notation three basic concepts come into question: graphically, logically or performance oriented approach.

It is necessary to realize that the music notation is not a direct equivalent of music. It is only a way to preserve musical ideas and share them between musicians (or more generally humans). There are other ways to accomplish this task as for example a sound recording. Both are using different means to transfer the musical information, but neither is able to conserve it exactly. Some information has lost, some was added. Music notation is a graphical system. Therefore it has to respect some special requirements that are not asked from the musical performance. Firstly, it is the esthetics of symbols and their layout, it does not seem important at a first glance, although the esthetical part is often accented in the teaching books of notography and it is an information that is completely lost in the performance. Secondly, there are many logical data which are also not necessarily present in the music. For example enharmonical change – G# is not the same note as Ab although they represent generally the same tone. In contrast, some pieces of information are not present in the notation and are deduced by the interpreter during the performance.

For all those reasons, I decided that a graphical approach will form the base of my representation. However, I will also need to keep evidence of the logical data and so I need to place my representation somewhere between the graphically and logically oriented one. The Lassfolk's model seemed to me as an ideal solution. It is based on following two assumptions:

I. Music notation represents music by interrelated graphical symbols.
II. Music notation does not exist without the presence of at least one identifiable graphic symbol.

The immediate consequence of these is that the central part of the representation is formed by graphical objects. However the logical information is modeled in the relations between them and thus can be easily retrieved. Further deliberation about these two statements lead us to the conclusion that ever purely logical information should be represented only in this manner. This type of representation is, in my opinion, closer to the music notation than any other and thus is optimal for a WYSIWYG notation editor.

III.2 - The CMNSymbol Class

As a consequence of above considerations, all the classes present in the model have their visual representation and no purely logical class exists. There will be of course some helper classes in the final program, which won't be able to render themselves. But these will be enforced by the implementation process and do not form a part of the abstract representation model. Following is the description of the basic class in the Lassfolk's model…

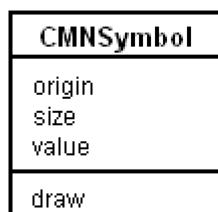

Figure III-1 CMNSymbol class

The *CMNSymbol* is a common ancestor of all other classes in the model. It has an abstract *draw* method, which causes the object to render its visual representation. Each *CMNSymbol* manages a two-dimensional coordinate system which will be called *internal coordinate system* further on. It is also assumed to be positioned in an *external coordinate system* which is managed by its aggregate object (typically another *CMNSymbol*). The *origin* attribute determines the position of the object's internal graphic origin within the external coordinate system. The *size* attribute determines object's size relative to the size and dimensions of its aggregate object (I made some violations to this interpretation which I will explain later). The position of all parts of the *CMNSymbol* is relative to its origin so that when it is moved, all of its parts move with it.

CMNSymbol also holds an abstract *value* attribute of unspecified type. Its purpose will vary between various subclasses. For example in the time signature class the *value* attribute holds a fractional number (e.g. 3/4, 2/4), in the note head class this can hold its type (open, closed, square, round).

III.3 - The top-level aggregation structure

The musical score is modeled as a hierarchical structure and in this paragraph I will present its top-level part. Also starting from here several important deviations from the original model will appear and will be explained.

The initial aggregation scheme (represented by an UML diagram) looked like this:

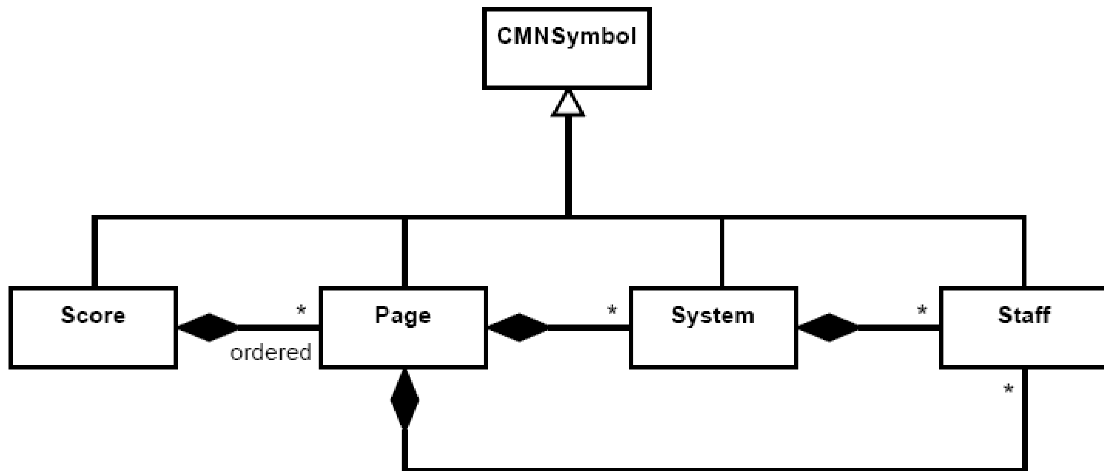

Figure III-2 Original top-level model

The Score is divided into an arbitrary number of pages which in their turn contain an arbitrary number of systems or staves. All four classes are inherited from *CMNSymbol*. The page can also contain non-musical symbols as text or conventional graphics but this is ruled out from the model for simplicity. The pages are defined as being ordered by the score and are not considered to be stored within a single two-dimensional coordinate system. This is done because the simple position does not define sufficiently their ordering and reordering of pages leads to a completely different and usually incorrect interpretation of the score. The system consists of an arbitrary number of staves.

The first important change I made is that the *CScore* class is not a subclass of *CMNSymbol*. I wanted to separate the internal representation from the rest of the program in order to create an integral functional layer with a single entry point which is the score. As a result, the *CScore* class has specific methods and furthermore, it does not make part of the described two-dimensional coordinate system (What does the score position mean?). It lays between the editor user-interface and the internal representation. In summary, the attributes of *CMNSymbol* do not have a well-defined meaning in the score and from the logical point of view it is different from other *CMNSymbol* derivates.

If you look carefully to the diagram, you notice that the staff may be a part of the page or the system. The author adds a restriction which forbids the state where the staff belongs to both in the same time. I look at this as a superfluous design problem. Moreover I think that a more clean and systematic solution is to define the staff strictly an aggregate of the system and allow systems having only one staff. Which is not prohibited in the original model, by the way.

The third important change in this top-level aggregation section is the introduction of *CPart* class (inherited from *CMNSymbol*) which represents a musical part. It is true, that the part was one of the examples, which the author cited as purely-logical

information which should not be modeled by an object. Nevertheless, I believe that a part has its distinct visual interpretation: staff connector, part name. Moreover, the notes can pass from one staff of the part to another (which is not allowed between different parts). I went further in my reflections and finally considered the part as a holder for shared staves resources as the note font and notation objects: staff connector, part name and attachments (slurs, ties, etc.). I will present more details later.

You can see the final top-level aggregation model on figure 2-3…

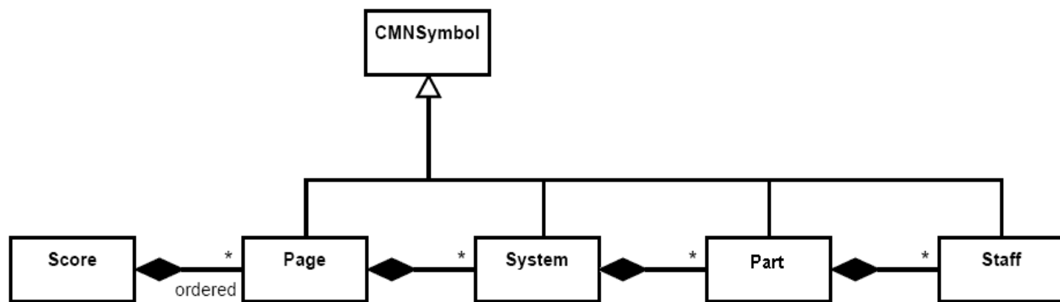

Figure III-3 Modified top-level model

A staff now always belongs to a part which, in turn, makes always part of a system. The score remains the root of the aggregation hierarchy, it is: all the other symbols are direct or indirect part of a *CScore* instance.

Chapter IV – The internal representation

In the previous chapter, I presented the top-most part of the internal representation model. This chapter will provide its detailed presentation in the form of UML diagrams and textual description.

IV.1 - Down the aggregation hierarchy

IV.1.1 System and staff connectors

When we descend a bit in the aggregation tree, we will talk about staff connectors, bars and barlines. Here comes probably the greatest and most important change I made to the original Lassfolk's model. Initially this part of the model had the following structure:

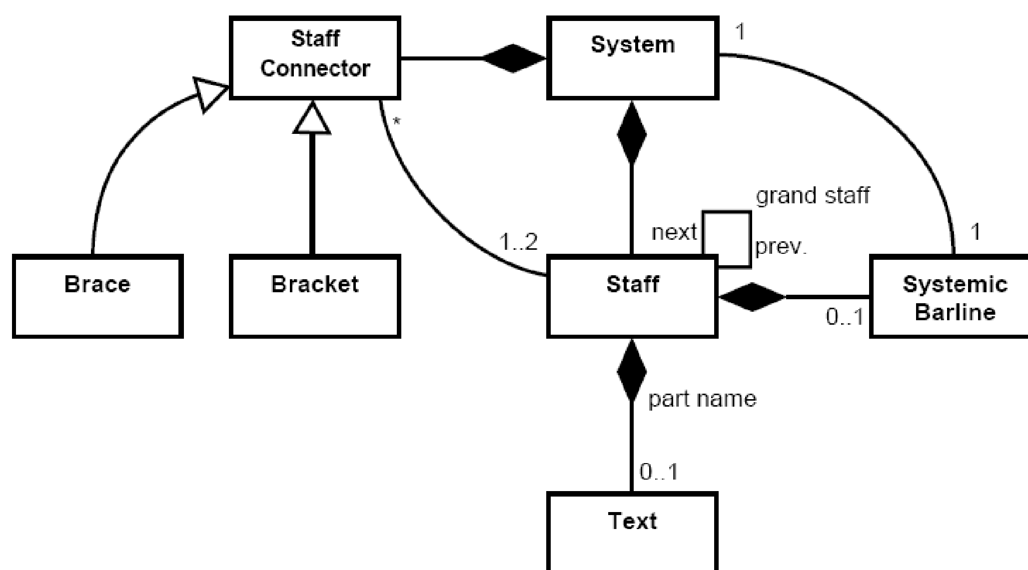

Figure IV-1 Original system model

In music notation, the staves can be grouped to form systems. Visually this is represented by a systemic barline which connects the staves. The staves can then be connected by braces or brackets to form groups or subgroups within the system. In [4] the model contains a systemic barline which is a subclass of a generic barline which is, in turn, aggregated in the staff. Thus, the systemic barline is owned by a staff and the specialized relationship to the system is realized by an association. The association type ensures that whenever the system is present there is always a systemic barline associated with it. The system can also contain an arbitrary number of vertical brackets or braces connecting the staves. These two classes have a common ancestor *CStaffConnector* which has an association with 1 or 2 staves. The staff connectors are not a direct equivalent of parts or sections. This model is general enough to permit a part to be written on more than one staff (as piano part) or, in contrast, include more parts on one staff. The *grand staff* association permits two staff instances to refer to each other. I add that this association reassured me about the introduction of the *CPart* class. The relation between the staves is

now defined explicitly and centrally and their communication is simpler to coordinate. The original model is more general, as it permits (as mentioned) to have more than one part on a single staff for example. Nevertheless, I do not need so universal representation and I find the cited benefits of the part class more valuable than to keep this capacity which would not be probably used very often. Moreover, the handling of parts would be much more difficult to implement in the original proposal.

The first important thing I found objectionable on the described model was the barlines ownership. As stated above, each staff possesses its own barlines. However, the barlines are something to be shared among all the staves, since the measure has one opening and one ending line that are drawn on all the staves (possibly not continuously but it is still the same symbol). Even more this should be valid for a systemic barline. The association with system ensures that one and only one will always exist when a system is present. But which staff should be its owner? Lower in the aggregation hierarchy we will find some more similar questions which finally led me to a decision to introduce a totally new class to the model and split the main aggregation line into two branches…

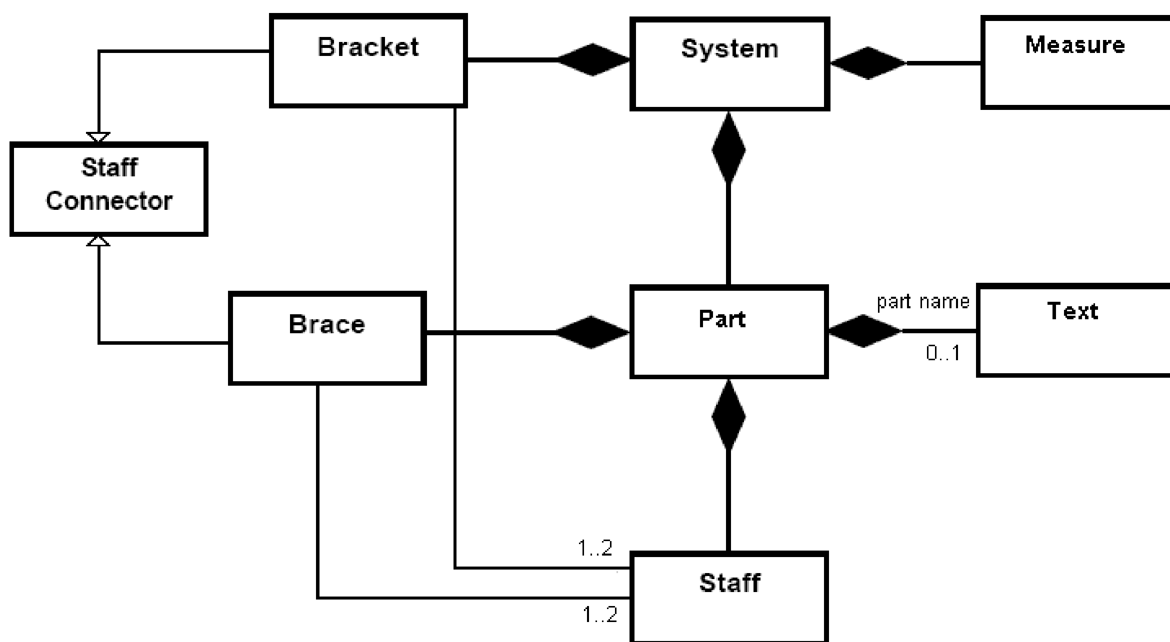

Figure IV-2 Modified System model

The *CMeasure* class represents one measure (bar) of the composition. It aggregates all the symbols that are shared among the staves (or parts) and also the notes and rests. For spatial reasons I couldn't have drawn them to the diagram, but all the barlines are now part of a measure class and not the staff. As for the systemic barline class, I discarded it completely and moved its drawing to the system's *draw* method. Firstly, it appears just once on each system, secondly it is an inseparable part of the system and finally, it is very simple to draw. My decision to introduce the measure class was also helped on by the fact that the measure seems to be the principal management unit in Encore. In this representation the automatic-alignment algorithms and the implementation

of basic user-actions (as resize the measure by dragging its barline) should be much easier than in the previous one.

The brace and bracket are now each owned by a different symbol. This is done because the brace is dedicated to join staves of piano part, while the bracket serves to connect staves from possibly different parts. However, they have still a common superclass, since they have still the same common properties as in the previous model.

IV.1.2 Part and staves

The staff is a basic part of every music score. It contains notes, rests as well as many other symbols as slurs, ties, clefs, dynamic markings etc. In my model, the staff always belongs to a part, so I will describe both classes in this paragraph. The ownership of durational symbols (notes and rests) was moved to *CMeasure*, so only attachments remain from the previous enumeration and they are owned by *CPart*. A staff has typically 5 lines, but this is not obligate and there are special cases (choral notation, percussion staff) which use a different number of lines. For that purpose the original model proposed a special class *staff line* which was aggregated in *CStaff* and represented one staff line. I do not use it. Instead I store the number of lines as a *value* attribute of the *CStaff* class and make it responsible of drawing all its lines alone. I do not think that a specialized class is necessary for such a simple function. Here is the UML diagram for this part of the model…

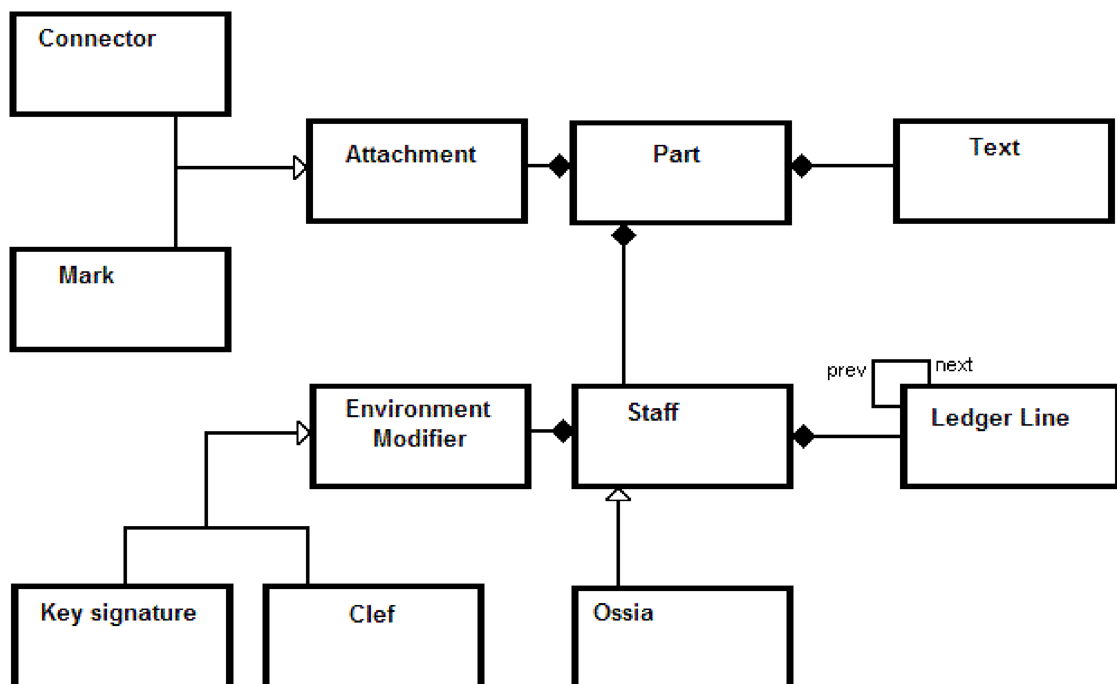

Figure IV-3 Part and staves

As for the attachments, we will see their detailed description later. The *environment modifier* is a common ancestor for symbols which have the property to affect the interpretation of notes. They are associated to measures in all the following measures until a next modifier of the same type appears. They are owned by a staff, not the part, because every staff can have its own key signatures or clefs independently to each other. There is also an *Ossia* class that inherits from *CStaff*. Ossia is a small staff placed near the original staff indicating an alternative passage of the composition.

The last class on the diagram is the *ledger line*. A ledger line is a short line drawn above or below the normal staff to indicate notes of extra-high or extra-low pitch. In my model the related ledger lines (used by the same note) are connected to a double-linked chain and they are associated with the note that lies on them. The motivation will be described with the note insertion algorithm.

IV.1.3 Measure and barlines

A measure is the central aggregate in this model. It contains durational symbols and measure modifiers. Every symbol placed on a staff is associated with a measure. It is the basic unit of the composition and also the editor's user interface will be measure-oriented. It means that most of the commands will take measure(s) as their parameters.

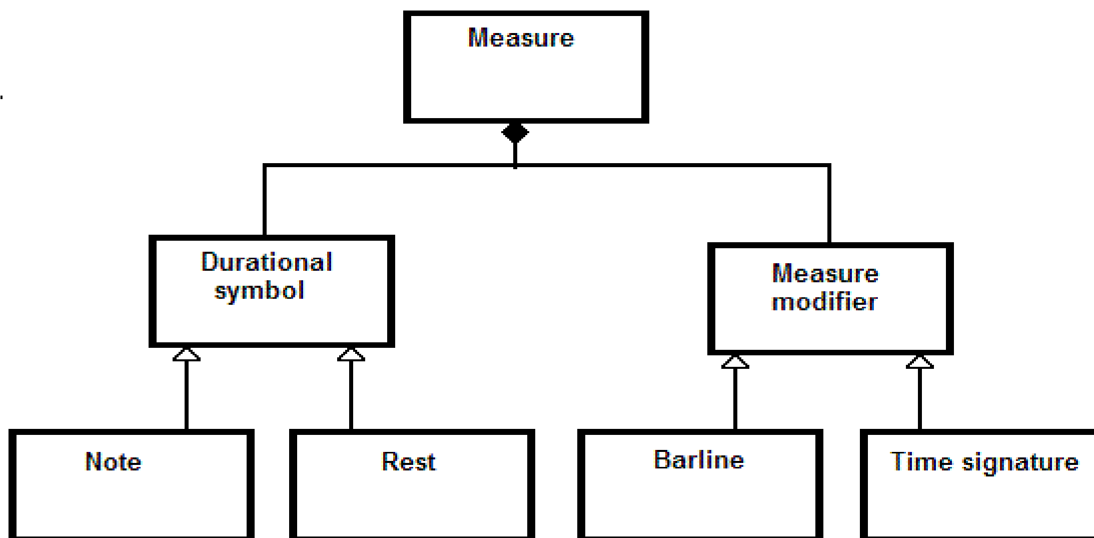

Figure IV-4 Measure and barlines

The *measure modifier* is the symbol which modifies the properties of its parent measure and is valid for all the following measures until next modifier of the same type is found. This is similar to the *environment modifier* with the exception, that *measure modifier* affects the measure, it is, all the staves in its section while the *environment modifier* has a local effect on its staff only.

The *measure* has always a type while it does not have to possess an instance of the *time signature* class. The time signature is only a visual representation of the measure's type. If it exists, it has only one instance for the whole measure and is drawn to all the staves in the system.

The measure also owns two *barline* instances. The barline type is determined in the following way…

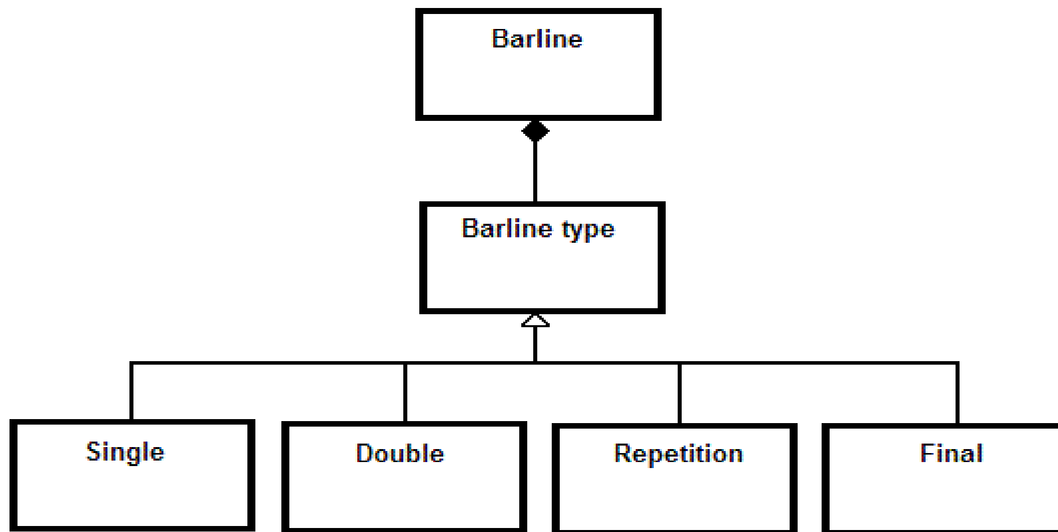

Figure IV-5 Barline type representation

For each type there is only class common to starting and ending barline, the correct drawing will be assured by a special flag indicating which direction the barline has. As in the case of time signature, the *barline* is drawn to all staves. More precisely, the barline may be interrupted to indicate the groups of staves in the system, but still exists in one instance only.

The *durational symbol* is a common ancestor for notes and rests. Its name suggests that it instructs the performer to produce a sound event or to hold a pause of a specified duration. It is a generalization that holds common notes and rests properties as well as the properties that affects the symbol placement and spacing on the staff. It can contain an arbitrary number of *augmentation dot* objects and is associated to the staff where it is placed. The *measure* is responsible of storing durational symbols in a way to be able to retrieve all the necessary logical information as the part and voice, the symbols belong to or the filling of the measure.

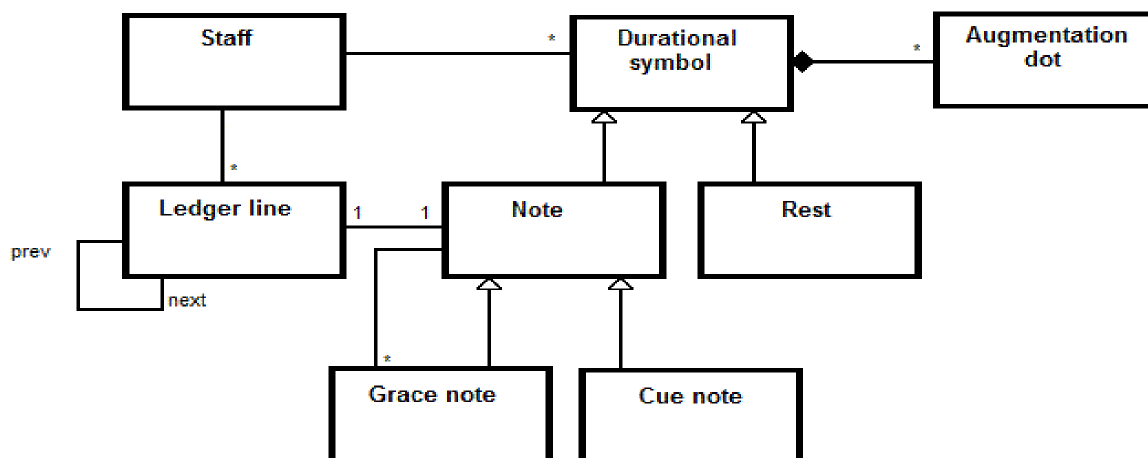

Figure IV-6 Durational symbol

IV.1.4 Note

Note is a very complex class which relates with several other classes. Some of its properties are inherited from *durational symbol* (as association with augmentation dots). Its scheme is shown in figure 3-7.

Figure IV-7 Note

This class was taken nearly without changes from the original Lassfolk's model. The position of the note is determined by halfspace on y-axis (and the association with staff inherited from *durational symbol*) and the beat on x-axis (and the relation with its parent *measure*). Thus, the note does not store its pitch. It is logical information that can be easily computed from its position (what the performer actually does when he reads the score). On the contrary its duration is directly stored in the *value* attribute as it is in *durational symbol*. Not because this value could not have been recomputed from other characteristics, but this attribute will be very probably often accessed (in contrast to the pitch) and its recalculation on each demand would be ineffective. This is a small deviation from [4].

The central but not mandatory part of this class is the *notehead*. Note head is modeled as a dedicated class. Its type is represented in the same way as in the case of *barline*. There is an abstract class *NoteHeadType* whose derived classes stand for the different note head types. The only true difference in this case is its visual appearance; the type of the head has no other impact on the note's interpretation.

Note can have from zero to two *stem* instances. Music notation some times uses notes with two stems pointing in opposite directions as in two-voice parts where both voices share a common note head.

In the original model, the *stem* can have an arbitrary number of *flag*s. One implementation issue forced me to modify this structure in the following way: The *stem* contains just one *flag* instance whose *value* determines how many flags will be drawn. This alteration was made, because I use the Anastasia note font (used in Encore) for the drawing of most of the symbols (including the flags). I could not make a new font on my own. And there are separate symbols for the flag of the eighth note, a double flag symbol and another flag to use in combination for shorter notes. This would be difficult to implement in the former model. Stems of different note objects can be beamed. Beams are not implemented, but they will be possessed by the part class.

Notes can be related with other notes through the *chord* association. A chord indicates a group of notes that are stacked one above the other, are of equal value and, if the value requires the stem, share the same stem.

Note has two subclasses: *Cue note* and *grace note*. They differ from the plain notes by their visual appearance (smaller size and in case of the grace note also the spacing rules) and interpretation. Grace note is also associated with one note.

Note may be also associated to a ledger line and, as described above, the ledger lines can be linked in a chain. Therefore, the note can refer to the ledger lines it is laying on et versa.

Other musical symbols which directly affect the interpretation of the note they are attached to (accidental, ornament, articulation and arpeggio) are modeled as optional aggregates of their *note* instance.

Tremolo beam may be associated either with a *stem* or directly with a *note*. The former association refers to situations in which stems are connected with tremolo beams. The latter association applies when no stems exist. A note may contain *TremoloLine* objects, which may be associated with a s*tem*. Tremolo-Lines, which are part of the same *note*, are also associated with each other, as defined by the *piled* association ([4]). The tremolos are not implemented, but they will be also owned by the part.

IV.1.5 Environment modifiers

The *environment modifier*s affect the section of the staff which owns them from the point where they are placed till the next modifier of the same type. Hence they are associated with the measure where their validity begins. There are two environment modifiers in my model: *Key signature* and *Clef*.

The *clef* represents a notational clef. Its type is once again determined by the intermediate of an abstract *ClefType* class as it was in the case of *barline* or *notehead*. Sometimes, musical notation uses superoctave or suboctave variant of the clef to indicate that the notes are in reality pitched one octave above or below their notated tone. For this purpose I added an *OctavesShift* attribute to the class. It can hold in theory any integer while the negative values denote suboctave variants (1 octave below, 2 octaves below, etc.) and positive values superoctaves. In practice, only 1 to 2 octaves shift would normally make sense. I created only three types of clef: *G-clef*, *C-clef* and *F-clef*. They all store the *base tone* and *baseline* attributes that stand for most important characteristics of the clef: The tone it notates and its position relative to the position of the clef. The position is controlled by the *origin* attribute whose x-axis represents the beat, where the clef validity begins and y-axis signifies the halfspace. Consequently, as a combination of the basic type and position, the user can create all the existing clefs and, if he wants to, also some new.

The *key signature* indicates the scale in which the composition (or its part) is notated. This is done by the intermediate of constant accidentals. I modeled only the common 12-tones European scale. As a result, the key descriptor has 7 items which corresponds to tones C through B and accept integral values from the range -2…2 (double flat…double sharp). Thus, the user can represent all the common scales in western notation and, if necessary, some new as well. The position of the key signature is again controlled by the *origin* attribute, but only the x-part (beat) is considered. The y-part is ignored and the vertical position of the accidentals is controlled by the clef that is valid for this location.

IV.1.6 Attachments

This part of the model is not implemented in this work. It is left for the master thesis. The attachments will group practically all the remaining notation symbols. Their common property is that their meaning is not defined without the presence of note or group of notes they are attached to. They are not implemented, but I have already prepared a set of their classes and associations into the internal representation, so I will briefly describe them.

The attachments are divided into two subclasses *Connector* and *Mark*. Connectors are joining a group of two or more symbols. Mark represents an instantaneous event which, however, may begin a gradual progress. Semantically we can describe the difference between connector and mark as follows: Connector is a symbol with explicit beginning and end while the mark is either an instantaneous event or beginning of a process which has either an implicit end or whose end is explicitly stated by another

symbol. As an example of connector, I can mention the slurs or ties. As for the marks, the dynamic marks are a good example. The attachments are owned by a part which contains the notes they are attached to. This ownership is justified by the fact that the existing attachments cannot affect symbols of more than one part (more than one instrument). There are some cases in which this would make sense (as the dynamic marks), but it is not used in the standard music notation and the attachments cannot be shared between parts. On the other side, the attachments are neither part of notes (they can affect a whole group of them) nor the staves (the affected notes can traverse from one staff of the part to another).

A scheme of connectors' classification is shown in the figure below. The connectors can be logically continued for example to the next system or page. This occurs when a connector joins the notes which are not in the same system (typically for spatial reasons) but are still all in the same part (logically, it is not the same instance).

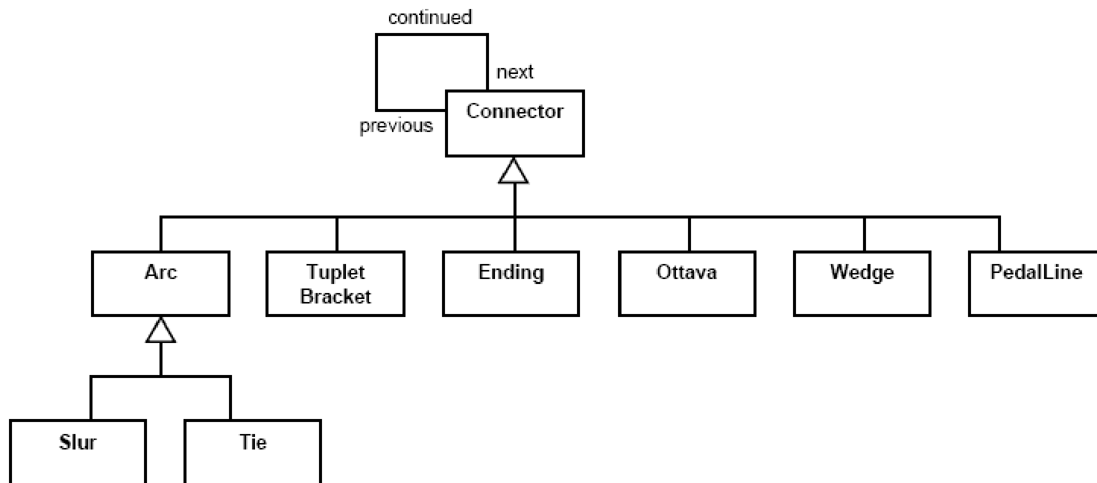

Figure IV - 8 Connectors

The slurs and ties share the common shape, but differ by their placement and function. Therefore, they are modeled as subclasses of a common class – *arc*. The *tuplet bracket* holds both the bracket and the number similarly to *ending* and *ottava*. The *wedge* class is used for representing crescendo, decrescendo and diminuendo signs. *Pedal line* contains a pedal symbol and a line showing how long the pedal is hold. The instantaneous pedal mark is a subclass of mark and is not considered as a connector.

Following is the diagram of marks. Most of the symbols were already described in chapter II. *Pause* stands for fermata and pause signs (as for example breath mark). *Text* frame is a superclass for *lyrics* and *rehearsal mark*. Lyric represents a fragment of song lyrics (one word or syllable). Rehearsal mark represents both rehearsal numbers and rehearsal letters. Since the attachments are not implemented, neither the connectors nor

the marks are in definitive state. More subclasses can be added, some details may be changed, but the overall conception should remain stable.

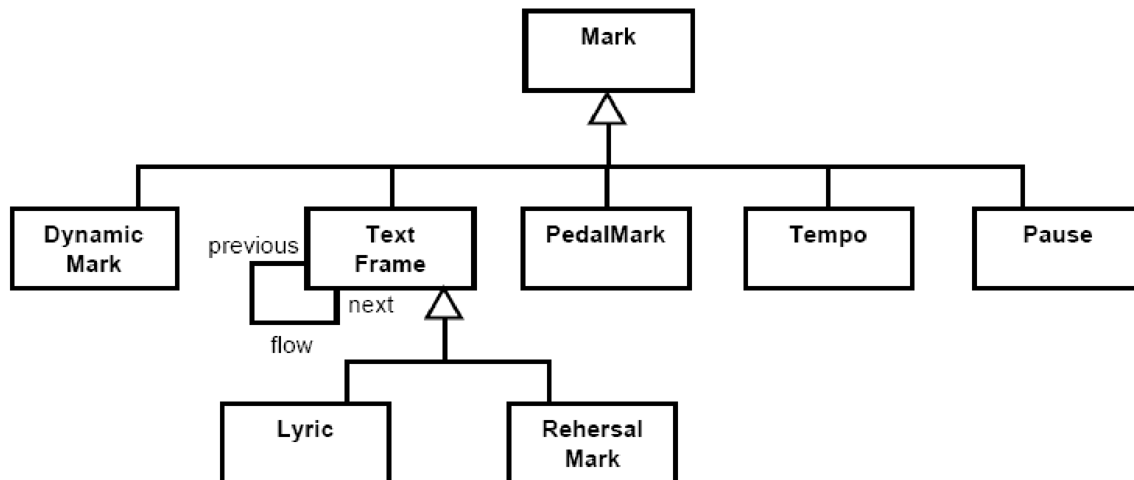

Figure IV - 9 Marks

IV.2 - The score class

After the aggregation hierarchy was described, I will return to its topmost class – the *score*. It is, as I already mentioned, the entry point to the internal representation. All the communication between the editor and the representation should pass only through the score class or the CMNSymbol interface. I will now list the basic services provided by the score class along with their explanation.

Adding and removing notation symbols to / from the score. All the symbols which are present in the score are direct or indirect result of some of the score method call. Only descriptions of the symbols are passed as the parameters because only the direct parents of the symbols are allowed to create or destroy their instances. This is not a requirement coming from the music notation, but mostly from the philosophy of the object-oriented programming. Compared to the state where the symbols are created in the program and then passed to the score or directly to the parent object, this approach is "cleaner" from the design point of view and also the final program code is easier to manage, because every object has its stable and well defined point of creation and destruction. This is also the reason why I tended to have a model where no class is allowed to be owned by more than one object and modified the original Lassfolk's model wherever this requirement was not satisfied.

Drawing of the score. The drawing of the score or its selected parts is also accomplished by a call to the score instance. Since the score is organized per pages, the minimal unit for the drawing method is one page.

User interface support. The score class offers several methods to support the program's user interface.

- First is the management of so called "active symbols" which are the symbols containing a given point (typically the coordinates of mouse pointer). There are two types of them – Active measure and staff which are managed internally and the program cannot access them. It can only demand their calculation. They are used for symbols insertion. When the editor requires an insertion of note or rest, they can be inserted to active staff and measure. Second type is a generic active CMNSymbol and the program can demand a reference to it. As explained at the beginning, all the notation symbols are subclasses of this class. Thus, the editor can use the common symbols interface to manipulate principally the visual aspect of the active symbol. This type of active symbol is also used for context menus support. I will provide more details in IV.3.
- Another user interface facility is the "User-Interface symbol". This is intended specifically for the mouse input. When the user inputs a symbol, specifically a note, it is difficult for him to see precisely where it will be inserted, especially when it is placed on ledger lines. Therefore a note head of the regular size is displayed under the mouse cursor and shows where exactly the note will be placed (in vertical coordinate). When the ledger lines are needed, they are displayed as well.
- The last but not least is the undo / redo support. Every non-constant action on the score returns a special object which can be executed to undo the action effects. The only restriction of use is that classical stack architecture of undo / redo is supposed. It means that every compensating action presumes that the score before its execution is in the same state as it was after the commitment of the action it is compensating. If this is not true, then in most cases, the compensating action will not be able to commit. However, if it commits, it returns a compensating action for itself, so that the undone action could be automatically redone if necessary.

Common notation tasks. The common tasks as key signature change or setting the measures per system attribute can be realized by a single call to the score object.

Score information. The last service is a wide range of informative methods retrieving the actual count of measures, size of pages, barline types of given measures etc.

IV.3 - The common symbols interface and active symbol service

In the previous section I returned to the score class, in this one I will return to the CMNSymbol class and describe in more details its interface. As mentioned in IV.2, beside the score class, the editor has one more communication way to the internal representation and it is the reference to active symbol and the CMNSymbol interface. From its greatest part, this interface is a set of virtual methods allowing the modification of common symbols attributes (position, size, color, visible state). The methods are

virtual, because the change may not be accomplished the same way in case of different subclasses. Many symbols have for example the aggregates and may need to propagate the action to them. Nevertheless, this class also offers three types of user interface support methods:

- Activation / deactivation – This pair of methods serves to facilitate the symbol selection. It can be called anytime, but is typically used when the mouse pointer passes to the object's area. Then the activate method is responsible for changing the object's visual appearance to mark that the symbol is active and the deactivate method returns it to the previous state. The visual appearance may change in different ways. Most often the color change is used, but the object can display a selection frame, anchors or whatever it needs.
- Context menus support – The CMNSymbol defines a virtual method *getUIActions* which returns a list of actions which the user can perform on this specific symbol. These actions are instances of the same class as the undo / redo support objects and so they behave in the same way returning automatically compensating action on their execution. If user selects a menu item then, if the corresponding action does not need any parameter, it is executed via *executeAction* method also defined by the CMNSymbol interface. If some parameters are needed then possibly some dialog is shown and finally the action specification is filled and the action is executed.
- Clone method – This method creates an exact copy of the symbol. The outcome from its creation is the support of the clipboard – especially copy & paste feature. However, the clipboard support is not implemented in this thesis, so this method is prepared for future work.

IV.5 - The undo / redo actions

I have talked several times about an object representing a compensating action or context-menu item, but have not said much about it. It is a simple but powerful class. Its diagram is presented in the figure below.

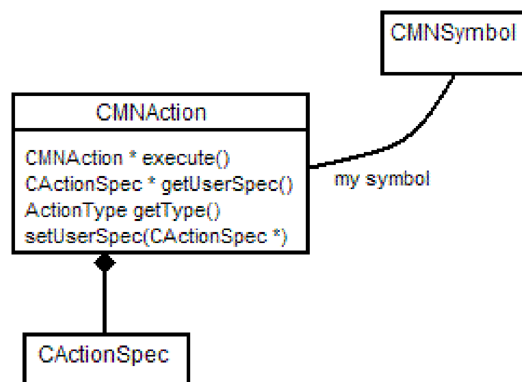

Figure IV - 10 CMNAction class

As you can see it has an association with the CMNSymbol. It is the topmost symbol which was affected by the compensated action. This symbol will execute this action in its *executeAction* method. Despite, as you certainly noticed, the action does not grant access to this association. In contrast, it has its own *execute* method, which will make the call to CMNSymbol and will return its result - the compensating action. The reason for that is mainly the ease of use. The editor can simply have two stacks: one for undo and one for redo actions. When the user requests to undo the last action, the program just pops the action from the undo stack, calls its execution method and put the result to the redo stack.

Each action is holding its type which serves for its identification in the symbol's execution method and in the context-menu display method. This type is a simple enumeration with values as *ChangeBarlinesType* or *UndeleteSymbol*.

There are actions which do not need any additional parameters and their type is sufficient for their execution. However, most of the actions will need to provide additional specifications before they can be executed. For example, the *ChangeBarlinesType* action will need the numbers of affected measures and the new barline types. For that purpose, the action holds a reference to a *CActionSpec* instance. *CActionSpec* is an abstract class and if an action of given type needs to pass additional parameters to its execution, it is supposed to inherit a new specification class from CActionSpec with all the data items it needs and pass it to the action object. When it is executed, the action specification object is casted (based on the action type) to the correct type. A run-time check is performed for type-safety. Since it would not be very efficient to have a special specification class for each type of the action when they have often the parameters of one or two different types, I have written several generic specification classes which can be used. Moreover, the actions can have a recursive character, when one complex action is a composition of several simpler action executed in a pre-defined order.

There is one more thing that is worth mentioning and that is the delete / undelete realization. As I already explained, the compensating actions are presuming the stack undo / redo architecture. How to restore the original system state after an object was deleted? There are several solutions to this problem, which are discussed in the next chapter. Finally I decided for the following one: When the object is deleted, its instance is actually not destroyed. Its records are removed from its parent's internal structures and the reference to the instance is stored in the action object and on its execution, it is reassigned to its former parent. If the action is not executed then it is responsible for destroying the symbol instance in the time of its own destruction.

Chapter V – Discussions

In the course of the editor development I often came across a situation where a problem had more than one possible solution. Sometimes it was easy to found a comparison measure between them, other times it was a difficult task to choose between. In this chapter I will introduce the most important or interesting of those problems and discuss the alternative solutions and the reasons which led me to choose the implemented one.

V.1 – Notation symbols classification

The greatest part of the classification was taken from the Lassfolk's model. I tried to justify my alterations at the place of their description, but I will sum up a bit their motivation and discuss in more details the most important one – the introduction of a measure class.

One of the principal motivations of my changes was to resolve all the ownership or object relations ambiguities. Firstly, the resulting program code is cleaner, simpler and more easily manageable. Secondly, the author often have not well defined (or even not defined at all) the criteria of selection between the candidates. The restriction of the relations (often connected with an introduction of new classes) led me sometimes to a partial loss of universality. This is the case of creation of the part (resolving the ambiguities of attachments ownership). By its adoption to the model I sacrificed the possibility to notate more parts on one staff. Nevertheless, as I already told, this capacity would not be, in my opinion, often used and it can be, if necessary, simulated with a one-staff part with complex name and more voices.

A few times I have discarded some class and moved its functionality into another. I did this in case of very simple classes (systemic barline, staff line). I understand their existence in the analysis model – the problem should be decomposed to its basic components. But, those classes accomplished so trivial tasks and since they formed the leaves of the aggregation hierarchy, I found them unnecessary in the course of the design process.

The most important change I made was the splitting of the main aggregation line into two branches by the intermediate of the measure class. It is also the most disputable alteration. I had several reasons for doing it: Firstly it is resolving the ambiguities of the barlines ownership, but this is not very important, because other solution was to move the barlines to the system. More important were my understanding of a measure role in the notation and my reflections about the implementation of the editor.

A measure is representing the time coordinate axis of the score. It is a visually distinct object delimited by barlines, having its well-defined properties controlling the symbols placement – as its type and therefore should be modeled in the representation. It is true, that the very thing to dispute is the existence of its visual representation. It depends on the point of view. One can tell that barlines, time signature and so on are

distinct object while the measure is not. It is true that I would not make such an important alteration to the original model if I had only this objection. Nevertheless, when I thought about the implementation of the model I found several others which finally supported my decision. Firstly, as I told I based my user-interface on the one implemented in Encore. There the measures form central objects for score manipulations. Moreover, in the musical practice, the measures have also this important role. In the original model with decentralized barlines and no measure evidence, it would be difficult to even compute how many measures the composition has. The layout and durational symbols insertion algorithms described in next chapter would be also much more difficult without a measure notion. On the other side, separation of aggregation branches requires introduction of some communication between them. This complication, though, does not seem important enough to balance the benefits which a distinct measure class brings. There is one problem which I did not realize when I developed the class. It is that, rarely, it may happen that the staves belonging to one system are notated in different measure types. This situation is not very common, but exists and appears for example in some parts of Mozart's Don Juan or Orff's Carmina Burana. In the scheme I developed there is no mean to represent it. However, I have already thought about a small patch to my model which should fix this problem. I suppose to introduce it in my future works on the editor.

V.2 – Action objects

The action objects probably holds the primacy of the greatest count of alternative solutions I developed and selected from. It is due to the fact that as I worked on them I have been progressively discovering their potential and consequently I have changed their field of activity several times. Some of the solutions I invented would not work because of implementation language limitations or just for an unconsidered detail in their design. I will not describe all the propositions I made, but I will focus on most important ones.

Originally, the action objects were intended specifically for context menus support. The *CMNSymbol* interface declared a pair of virtual methods *getContextMenu* and *dispatch*, which were supposed to be implemented in the symbols that would use the menus. The first method returned a specialized object representing the menu. Its items were considered to be of two different types:

- *Non-dialog* having constant or no parameters as for example *hide* command
- *Dialog* requiring additional parameters from the user. These items should own the appropriate dialog class as their protected attribute which they would instantiate and manage themselves when they were executed.

After all the parameters are collected, the item should call the *dispatch* method providing the name of method which should be called on the active symbol and its parameters.

It was the parameters passing which defended me to implement this model. Since the methods have different count and types of parameters it was difficult to propagate them to the *dispatch* method. After this I tried another approach. I discarded the *dispatch* method and provided the menu item with a functor which should call a concrete method

on the active symbol. Because the functor would be a specialized object, he would know exactly the signature of the called method and the problems with parameters passing would disappear.

Nevertheless, I had many difficulties with implementing this solution and after a consultation with my supervisor we agreed, than no elegant way out of this problem exists and I developed the following design:

I have resuscitated the *dispatch* method, but renamed it to *executeMenuItem*. The menu item object contained only an identifier that could be resolved in *executeMenuItem* and when the user selected the item, it was passed as a whole to this method which executed it. The identifier was firstly realized by a resource identifier, so that the menu could be still automatically generated without further interventions from the program side.

Nevertheless, I started to realize the potential of this design and since I needed to implement the undo / redo support somehow, I began to think about merging the support for those two user-interface facilities. The realization was rather straight-forward. The menu items were extended to the description of actions and the result was nearly the design described in IV.5. However, several deviations still existed. First thing that has changed was the action identifier. As I already declared I wanted to develop the internal representation as a layer independent from the program that uses it. The communication between them should be realized by an intermediate of the representation's "API" and the program should depend on the representation not versa. However, the resource identifier used in action objects created a strong dependence of the internal representation on the editor. The presence of those resources could be stated as a requirement of use, but I found this an unclean solution. Therefore, the identifier was changed to an abstract identifier which is now represented by a global enumeration, but I think about possible improvements to the future.

Second difference was, that the compensating actions were returned directly by the methods of CMNSymbol interface. Finally, I redesigned them and made the *executeAction* the single method of CMNSymbol which returns the compensating action. First reason is, that the CMNSymbol methods can be called internally without intervention of the user and they will not be compensated. Those internal calls preserves logical state of the score and so normal compensating actions can still operate. Other reason is that some actions as *moveBy* are called repeatedly (when the user drags the object by the mouse) by only the final effect is to be compensated. Last reason is that most of the user activity comes from context menu and so through the *executeAction* method.

I have also needed to decide how to resolve the references to affected objects. Pointers are fastest, but become unusable if the object they referenced was deleted. In the time of compensating action execution, the score guarantees the same logical state as it had after the compensated action executed. But the validity of pointers is not a priori

warranted. I hesitated between two solutions: One was to assign an UID with every CMNSymbol instance and use it as a reference. This would however require to guarantee the uniqueness of the symbol in the score scope and particularly some indexing system or other mean to find the symbols quickly by their UID. I implemented the second solution, which on symbol removal does not free its memory and thus does not invalidate the pointers. The symbol is removed from internal structures of its parent and the pointer is stored in the action object. Because the memory is not freed, I tried to avoid its excessive consummation by limiting the undo and redo stack depths. Their level is set to 1024, which seems to me to be a reasonable number. Actions in deeper level are deleted along with their symbols, if it was a delete action. It is easy to find out, that the invalidation of pointers after the delete action was freed is not harmful, since the delete action was certainly the topmost action on the stack. After the symbol is removed from the score, no user-interface action can be taken on it.

This was the last paragraph concerning the action objects. Perhaps, I could have provided less details, but the action objects are an important part of the representation. Probably not at first glance, but without a built-in support for the compensating actions, it would be difficult to keep the representation and program separated. The editor would have to understand the classes and their internal structures and its code would grove considerably.

Chapter VI – Algorithms used in the editor

In this chapter I will present the important algorithms used in the editor and discuss alternative solutions.

VI.1 – Durational symbols insertion

When a durational symbol is being inserted into a measure, several integrity checks have to be performed. These checks are simpler for a rest than for a note. A whole sub-section is dedicated to the insertion of a note symbol. I will first describe the tests performed on a rest since they are common to both symbols.

First important fact to realize is that symbols belonging to different part and / or different voices do not influence each other and therefore all the process occurs in the context of a single voice of a single part. The symbol is typically inserted to the active staff and active measure of the score (described in IV.2) and their position is given as a point [beat, halfspace]. This point is computed from the mouse (or more generally a given point) coordinates – the halfspace exactly and the beat initially approximately. The result will be refined during the insertion process. A special value can be passed for the beat, indicating that the symbol should be inserted to the first free beat.

In the beginning, the approximate beat is tested against the measure type. If for some reason, it is out of range, the symbol is rejected with an appropriate error code. Secondly, the algorithm has to verify if there is still enough free space in the voice to accept the symbol of requested duration. The free space is determined by a sum of durations of already present symbols subtracted from the measure type. If the symbol is too long, it is again rejected. There is one exception from this simple rule valid for whole and breve notes. The whole note or rest can be inserted also to a measure with type shorter than 4 beats in condition that it is a single symbol in the voice. The breve note is an old type of note which is not often used today. It has the duration of two whole notes. No constraint is put on its insertion, although it is used mostly in a 4/2 measure. Finally the exact beat is computed: The algorithm finds the first symbol placed on a beat inferior to the requested one and attempts to put the new symbol immediately after it. That is the *Final Beat = Inferior's beat + Inferior's duration*. It may happen that the final beat is already occupied by another symbol. In this case, the symbol is rejected.

VI.1.1 The note insertion

The described process is common for the rest and note. However, the insertion of a note requires additional checks and computations, especially because of a possibility to form a chord with other notes. In the chord, all the notes begin at the same beat and are of equal duration. This complicates the last step of the already described algorithm in the following way:

If the final beat is already occupied then primarily check if the present symbol is a note or a rest. If it is a rest, reject the note. If it is a note, compare its duration to the new note's one. If they are different, reject the note. If the durations are equal, test whether the

present chord (we can assume for this step that a single note forms a simple chord) does not already contain a note on the same halfspace as the new note's one. If yes, reject the note. If all the tests passed successfully, insert the note.

There are a few other steps required to create a correct note symbol and they are all performed by the note instance itself. If the note is alone on its beat and does not form a part of a chord then the single remaining problem is the stem. The rule for its direction is simple, if the note is below the third staff line, its stem points up, otherwise down. However, if more voices are written on the same staff then the default voice of the staff has stems up and others stems down. The stem size is defined as 3 halfspaces if the note is in the regular staff space. If the note is on ledger lines then the end of its stem is on the 3rd staff line regardless of the note position.

In the chord the situation is more complicated. The calculation of all the chord attribute is a 3-phase algorithm:

1. Calculation of stem attributes and retrieval of the "base note".
2. Calculation of head attributes
3. Repositioning of the augmentation dots

Ad 1: The rules used in 2 and 3 operate with a so-called "base-note". It is the note from which the chord is theoretically drawn. If the stem points up then the bottom note is defined as the base note et versa. The size and direction of the stem is calculated as follows:

If all the notes lay on ledger lines and the topmost and bottom note are on the same side of the staff (it means that both are placed either below the staff or above the staff) then the stem ends (as in case of single note) on the 3rd staff line. Otherwise, it is drawn from the base note to the other extreme note plus 3 halfspaces. The direction of the stem is determined from a comparison of number of notes below the 3rd staff line and others. The resulting direction is the same as would have a note in this part of staff where is the greater part of the chord notes.

Ad 2: If in the chord there are two notes in the distance of a second then their heads have to be drawn on opposite sides of the stem, so that both have enough space. The positions are calculated from the base note. The base note is drawn in the normal position and then we walk through the chord drawing each note that would not have space in the opposite side. The ledger lines in this case should be drawn longer than usual, in the way that both note heads are underlaid. Only the lines from the place where this situation occurred in the direction to the staff have modified size. I have solved this situation very simply: Each note possesses its own chain of ledger lines. In a chord they are drawn superposed one above other. If the described situation occurs, then the ledger lines superpose only partially and thus look as a single longer line.

Ad 3: The last step is the positioning of augmentation dots. Firstly, they have to be clearly separated (1/2 halfspace) from their notes and stacked one above the other.

Hence, if some note head is put on the other side of stem in the phase 2, all the dots have to be moved appropriately. This is done by a simple pass through the chord chain. In the same time a second rule is controlled – if a note is positioned on a line, its dot is positioned in the nearest space above. This introduces one problem which is not yet correctly resolved in my algorithm: If the notes are in a distance of one second then their dots collide. A very vague description of the solution is given in [7] on page 96, but it is not a description of the algorithm. I have found, that even commercial editors do not solve this situation correctly. It seems to be surprisingly difficult, but I hope to find the solution in the course of future works on the editor.

VI.1.2 Discussion

From the length of the description of this, apparently relatively simple, task you can take a feel of the complexity of the rules governing the notation. The algorithm is a very straight-forward implementation of the rules described in [7] on pages 89 – 98. One interesting thing to discuss is however the approach taken to draw the stem and ledgers in the chord.

They are both solved in different ways. The main dissimilarity is that the stem instance is shared between notes while the ledgers are not. The main reason is the ease of management. It is true, that the superposition of more ledger lines is a bit ineffective. Nevertheless, all the constraints put to their drawing are satisfied naturally without any additional intervention. If the instances were shared, I would save some memory and processor time in the time of drawing (not much), but would introduce a large overhead in the time of note insertion and especially of its removal. Only the standalone ledgers would have to be removed and potentially they would have to be shortened.

In contrast it is very easy to manipulate a shared stem instance. I defined the base note of the chord to be its parent. In the time of insertion, the note is attached to the existing stem instance instead of creating its own. If it happens, that the new note becomes the base note, then the stem is redirected to it. When a note is removed, it is simply detached from the stem. All the stem manipulations concern just one stem instance. If it was not shared, the insertion and removal would be even more simple. However, the 2nd phase of the note insertion algorithm would have to modify all the instances instead of one.

VI.2 – Measures insertion / removal

The dislocation of measures in the composition is controlled by a *measures per system* attribute. This attribute is set by the user and determines, how many measures a system can contain. It can have less measures, if there is not enough measures to satisfy it, but not more. If new measures are inserted to the system and their total number is greater than the measures per system, the last measures overflow. They are not inserted directly to the next system, but to an auxiliary data structure and the score instance is responsible for inserting them to a correct location.

VI.2.1 Measures insertion

When the user inserts new measures, he sets 3 parameters: Their count, type and position where they will be inserted. The system containing the demanded location is then requested to create all the new measures and insert them. Typically, some measures will overflow including some of the new ones. The insertion procedure then walks through the score and inserts the overflowed measures to the beginning of the following systems while taking what overflows from them. If there is no other system then it is created. The systems are created by the page and their count is controlled by *systems per page* attribute similar to measures per system. If the page cannot contain the new system, a new one is created. This procedure continues, until the measures stop overflowing.

Since the durational symbols and barlines are owned by the measure, they are moved with it. Problem is with symbols in the second aggregation branch. A special data structure was created for this purpose. When the system accepts or removes its measures it passes their numbers to the parts it owns. If the parts contain symbols associated with removed measures, they release them from their internal structures and move them to this data structure. If, in contrary, the measures are being inserted, the parts examine the data structure for symbols associated with them. This data structure is common for all the measures manipulations and is described in VI.2.4.

VI.2.2 Measures removal

The user can remove a continuous range of measures by a single operation. The most general situation looks as follows: Some measures are removed from the end of System1 on Page1. The Page1 contains some other systems following the System1 then several pages follow and finally on Page2 some measures are removed from the beginning of System2 preceded by several other systems. The pages between Page1 and Page2 can be removed as a whole as well as the systems following System1 and preceding System2. After that we have Page1 ending with an incomplete system (having less measures than its *measures per system* attribute states) and Page2 starting with an incomplete system.

The rearranging algorithm starts from System1 stealing measures from the following systems until System1 is satisfied. Then it moves to next system and repeats the procedure. As a result we have a newly arranged score with as many satisfied systems as possible. Last system can contain less measures than its maximum.

VI.2.3 Changing measures per system attribute

When the measures per attribute is changed, all the three types of systems can appear: satisfied, incomplete and overfilled. To establish a correct state, the algorithm use a combination of procedures described in previous two sections. It starts at the first system and searches for an overfilled or incomplete system. If it finds an overfilled one, it takes its overflowing measures and uses the insertion algorithm described in VI.2.1. If the system is incomplete it uses the procedure of VI.2.2. When the invoked procedure finishes, next incorrect system is searched. The procedure ends when the last system is reached.

VI.2.4 Moving associated objects

The algorithms described above are rather simple. The only real complication in this section is the moving of objects associated with the measures – in this core-editor I talk about staff aggregates – clefs, key signatures and ledger lines. Their association to the measure is realized through the number of measure. The three described operations introduce each a different situation: When a measures per system attribute is changed, no measure appears or disappears, they are simply rearranged and their numbers remain the same. However, when measures are created or removed, their numbers change and so the objects associated with affected measures have firstly to be "moved" – their number of measure is increased (in case of insertion) or decreased (if measures were removed).

When all the objects are associated with correct measures, but possibly owned by systems which do not contain them, the rearranging procedure is called. It is a simple two-pass algorithm. In the first pass, it removes incorrectly placed objects of all the affected systems and puts them to the auxiliary data structure. In the second pass, it inserts them to their correct positions. The insertion and removal is controlled by the parts themselves. The procedure passes them the data structure and they take the symbols that they need.

The data structure is a relatively simple hierarchical object:

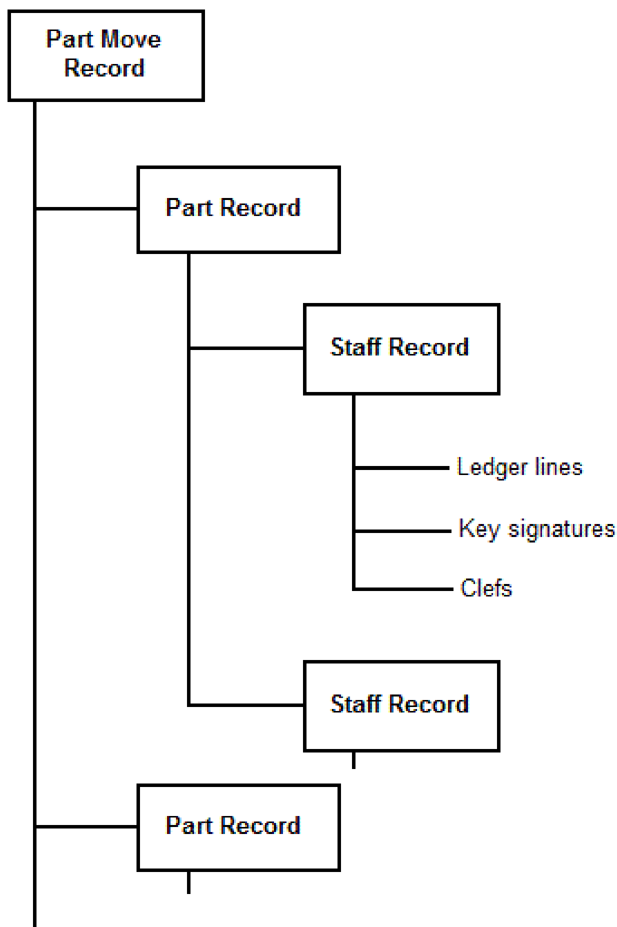

Figure VI - 1 Part move record

The system accepts reference to the *Part Move Record* and passes the appropriate *Part Record* to each of its parts which, in their turn, passes the appropriate *Staff Record* to each of their staves. The *ledger lines*, *key signatures* and *clefs* are simple lists. One reason for their separation to three distinct structures is lucidity. More important is however the fact, that key signatures and clefs require special handling: Each staff must display a constant key and clef in its first measure. Therefore, they cannot be moved arbitrarily.

The constant modifiers are held separated from other environment modifiers present on the staff. But on the staff creation their shadow copy is created placed in the same measure, but invisible and moved as any other symbols. Since the algorithm processes the composition from beginning to end, it is possible to write down the settings of modifiers valid for last measures of the staff that was just processed. When the algorithm passes to the corresponding staff on the next system, it works as follows:

- The stored last modifiers are compared to the constant ones of actually processed staff. If they differ, the constant modifiers are changed to the stored values.
- The Staff Record lists are passed through and objects associated with present measures are inserted. Additional operations are being made:
    - The staff is searched for other modifier valid for the target measure
    - If it is found and they are logically equivalent, the newly inserted modifier is shadowed
    - If it is in addition associated with the same measure and beat, it is removed
- Finally, if a measure is found in the first measure of the part, the constant modifier is changed according to it.

The last clef and key records make also a part of the Staff Record object.

VI.2.5 Discussion

I experimented with different implementations of the association of environment modifiers to their measure. The original one was realized by a pointer. The pointers have the same basic advantage as absolute measure numbers – when the measures are moved, the associated symbols move automatically. However, when the measures are renumbered, the pointers introduce a large overhead, because the new measure instance has to be explicitly found and the pointer redirected.

Therefore I changed the association to the measure number, but the first try used the numbers relative to the first measure of the part. I used this because I prepared the parts to start and end on arbitrary measures (see section Future works in Chapter VII) and so when the parts were moved, all their objects would automatically move with it. Nevertheless, this type of association required either more complicated Part Move Record structure with the data of base offset of the symbols or permanent recalculations of the relative measure numbers to absolute and back.

The association through the absolute measure number requires the renumbering of objects when the part is moved, but the implementation is much simpler.

VI.4 – Symbols layout / drawing

VI.4.1 Drawing

The drawing starts by a call to score's draw method while providing the number of the page which will be drawn and the point where its top-left corner will be placed. This allows the editor to draw more than one page to its client area. The rectangle where the page is drawn is stored in order to find the one under given point when program requests so. The score then calls the draw method of the page. This method is declared by the CMNSymbol interface and has the following signature:

*void CMNSymbol::draw(CDC * pDC,const CPoint& ptCoordOrg,const CPoint& ptCoordExt,const DrawSpec * pSpec);*

The symbols draw directly to a Windows device context, so the *pDC* parameter points to its instance. The usage of device contexts is the principal obstruction to the editor portability. If sometimes I try to port it to linux or other OS I will have lot of work with its replacement.

The *ptCoordOrg* represents the position of the origin of coordinates of the symbol's coordinate system and *ptCoordOrg* contains scales on the axes. Last parameter serves to provide additional specifications to the symbol if they are necessary. It has its own set of attributes, but if necessary, symbols can inherit a new specification class from this one and cast it when they are drawn.

The draw method typically acts differently if the symbol is displayed on the monitor then when it is printed. The output can be determined from *CDC::IsPrinting* method. The common difference is that when the symbol is hidden then it is drawn grayed to the monitor but is not printed at all. A special condition is evaluated when the durational symbols are drawn. Since they belong to different voices, the specification is used to determine which voice is actually edited by the user and symbols from other voices are drawn grayed. I point out, that this condition is not evaluated when the score is printed.

VI.4.2 Symbols layout

The layout algorithm is based on the principles described on pages 67 – 80 of [7]. It is working locally in one measure in the following phases:

I. The "dead space" on the beginning and the end of the measure is calculated. I use this term to denote a space where no durational symbol will be placed. This space is used firstly to separate the symbols from the barlines, so that they do not melt. Secondly, it is reserved for modifiers placed at the start of measure which have their fixed position defined in [7].

II. The "base sections" are located. The symbols are positioned using the "base scale" which is the distance of symbols in the "base sections". This scale is extended for symbols with longer duration or when an additional space is

needed for some symbol (e.g. if the note has accidental). The "base sections" are longest groups of notes having the same duration shorter than the duration of any other groups in the same section of the composition. An example is in the figure below.

III. For each section the total scale is computed. This scale has for its unit the "base scale" of the corresponding "base section".

IV. The base scales are computed. The real measure size reduced by the dead space is divided proportionally to the sections according to the count of their symbols.

V. The measure is passed from left to right and the positions of symbols are calculated.

Figure VI - 2 Section (in the dashed rectangle)

The most difficult part of this algorithm is the isolation of base sections. It works in the following steps:

1. Sections are located in each part and voice. By the term "section" I denote a continuous group of notes (in one part and voice) having all the same duration.
2. From these sections one base section is selected following two criteria in this order (but the selection is made in a single pass not two):
    a. Duration of notes – The section with minimal duration is selected

b. Length of the section – The longest section is selected among those with minimal duration
3. Next sections are searched beginning at the beat that follows the last note of the base section.

This algorithm is a bit simplified, but sufficient for the purpose of this core-editor. In the future works it will be improved.

VI.5 – Active symbol detection

The CMNSymbol interface provides two virtual methods to support a detection of an active symbol: *FindActiveSymbol* and *testPointInside*. The second method is a simple predicate that takes a point in its system of coordinates and returns *true* if the point is inside the queried symbol. The first method takes a point in the symbol's system of coordinates and then finds and returns the active symbol which can be the queried one, or some of its aggregates or none. The algorithm is simple:

1. It calls the *testPointInside* to determine if the point is inside this object. If not, the function returns a null pointer and terminates. Otherwise it continues to step 2
2. It recalculates the point to the system of coordinates managed by its aggregates and passes it to their *FindActiveSymbol* method. If some returns a valid pointer then the function terminates and returns it as its result. Otherwise, it returns pointer to itself.

Originally, the symbol called the *activate* method on itself, later the call was delegated to the score and finally, it is left to the calling program. Thus, the editor can decide when to activate and deactivate the symbol for the user convenience.

Chapter VII – Conclusion

The software I have made offers less functionality than I expected when I decided to work on this subject. A development of a music notation editor is a surprisingly complex and difficult task. It is resulting from the complexity of music and music notation themselves. Moreover the layout algorithms were complicated by the fact, that as a graphical system, the music notation has also its esthetic part and many paragraphs of the teaching books of notography start or end with: “this is a subject of the esthetical sense”. However, I think that my editor fits the specifications I have stated in the introduction and offers an interesting approach to this problematic. Unfortunately, for time reasons, I have not implemented the clipboard edit facility. But most of the necessary software support is already implemented and since the editor was from the beginning designed for high extensibility, its addition will be a simple task.

As for the benefits of my thesis – for my own, it taught me a lot of things. It is the first project of really large scale I have been working on. Its development brought me the discipline necessary for management of such piece of program code. I have improved the documentation and worked on my programming style. I have made a lot of effort to create a “clean” object-oriented representation of the music notation and I think I have succeeded from the greatest part.

As for its objective benefits – I have worked on the Kai Lassfolk’s study and I think I have developed its useful and interesting ideas and brought to practice his analysis. My object model is not a blind copy of the one he published in his work, but offers different point of views to some of its parts. Since my editor is only a core, it is not yet sufficient for the notation purpose. However, musicians and programmers can both find useful information either in my thesis or in its comparison to [4].

When compared to existing market products, I think my editor brought a very good user-interface. I could hardly compare the range of functions, but the user interface is a highly important part of each software and must not be underestimated. For example the Sibelius notation editor offers many powerful functions including the optical recognition of written scores. Nevertheless, its user-interface is so complicated that I had real difficulties to input a non-trivial composition. My user-interface is based on the interface implemented in Encore, which is the best and most intuitive (while remaining effective) I have ever seen in a notation editor. I have replaced the old visual elements by the modern ones and made some improvements and I believe that the final interface is even more powerful than the original one.

For future works I plan to do the following:

- Complete and improve the user-interface: Clipboard support, better dislocation of commands and a bit fresher visual style.

- I want, of course, to complete the implementation of all the notation symbols presented in IV and some other features which are not accessible for now:
    - The parts should be allowed to begin at an arbitrary measure and have arbitrary size. This would allow the user to add some part(s) only to a section of the score. This is useful when for example the soloist sings only in separate pieces of composition and is quiet the rest of the time.
    - Finish the implementation of zooming of the score.
    - Provide additional note fonts
- I plan to implement exports and imports to / from other formats used for musical notation, to name some: MusicXML, MusixTex, LilyPond. And also an import and export from / to MIDI format.
- I think about further exploiting the potential of action objects and replace the representation's API by the commands similar to these which are used in Microsoft Visual Studio 2005.
- I even meditate about rewriting the code either to .NET or more interesting possibility would be to put the representation above an abstract engine and make the program portable to linux or other operating system. I had experienced a lot of difficulties caused by MFC and sometimes I really regretted to use it.

To conclude, I think I have created a core of a live able WYSIWYG notation editor. An object-oriented model developed on the proposition of Kai Lassfolk was chosen for its internal representation. Some changes I have made would merit revision, and on the contrary, there are parts where I have followed too blindly the original model. However the representation in total is good and flexible. I am glad to select this theme and I hope to continue this work in my master thesis.

References

[1] Capella 2002, product documentation, www.notecka.cz, www.capella.de
[2] Encore 4.5.5, product documentation, www.gvox.com/encore.php
[3] Finale 2007, product documentation, www.finalemusic.com
[4] Lassfolk, Kai: Music Notation as objects, ISBN 952-10-2205-1, 2004
[5] Michels, Ulrich: Encyklopedický atlas hudby, NLN s. r. o., 2000
[6] Sibelius 4, product documentation, www.sibelius.com
[7] Zelingr, Ivo: Notografie: Učebnice notografického záznamu, Supraphon, Praha, 1986