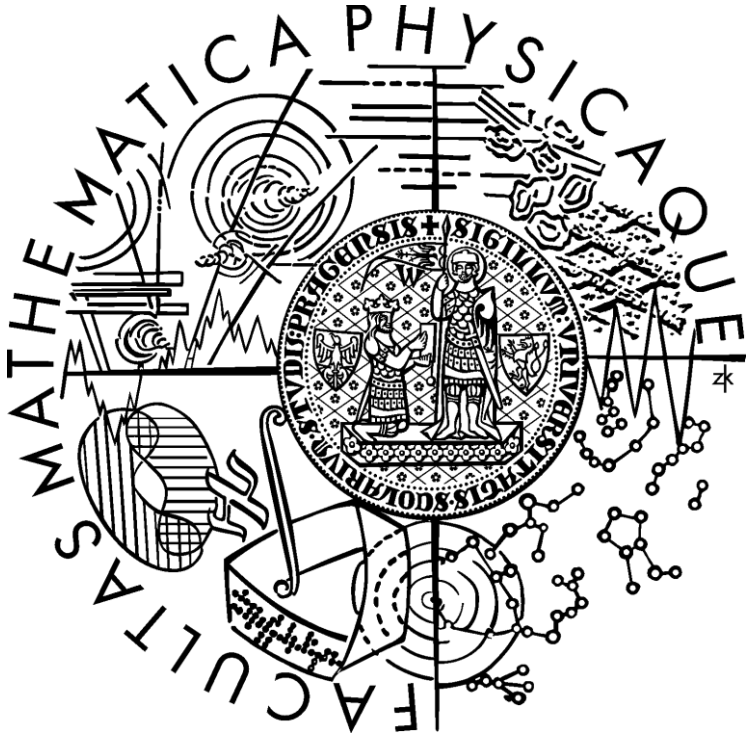
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Lukáš Kopenec

Notový editor
Musical Notation Editor

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. David Bednárek, PhD.

Studijní program: Informatika, Softwarové systémy

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 dík především mému vedoucímu RNDr. Davidu Bednárkovi, PhD. a mým vyučujícím v hudební škole (MgA. Michelle Hradecké, Markétě Džunevě a Monice Pecikiewiczové), kteří mi poskytovali cenné konzultace k navrhované koncepci a k potřebným funkcím editoru. Za druhé pak děkuji především svým rodičům a Ester, díky nimž jsem svou práci mohl vytvářet v příjemné a inspirující atmosféře a jimž vděčím i za celou řadu užitečných postřehů. Zvláštní dík pak patří mému otci za nádherný a originální název editoru.

Prohlašuji, že jsem svou diplomovou práci napsal(a) samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne

Název práce: Notový editor
Autor: Lukáš Kopenec
Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. David Bednárek, PhD.
e-mail vedoucího: David.Bednarek@mff.cuni.cz
Abstrakt: Cílem této práce bylo navrhnout a implementovat WYSIWYG notový editor, který by řešil některé problémy, se kterými se potýkají hudebníci v běžné praxi, a také se pokusil odstranit některé nedostatky existujících produktů. Hlavní motivací jsou zkušenosti autora se situací na hudebních školách, kde učitelé vlastní velké množství tištěných notových materiálů, které neustále kopírují svým studentům a doprovázejícím muzikantům, což vede jednak k vysokým nákladům (papír, toner), ale také k rychlé ztrátě přehledu o vlastnictví, počtu kopií a umístění jednotlivých partitur. Tato práce se snaží poskytnout školám platformu pro levnou distribuci a organizaci vlastněných notových materiálů. Vedlejší motivací je skutečnost, že koncepce současných notových editorů je neměnná v průběhu posledních dvou dekád a nabízí uživateli jen omezené možnosti kontroly celkového obsahu jeho hudebních projektů. Navržený notový editor umožňuje vytváření bohatších a komplexnějších partitur (např. zpěvníků nebo hudebních učebnic) a také možnost včlenění dalšího obsahu (např. nahrávek či doprovodných textů) přímo do vytvářeného projektu.

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

Title: Musical Notation Editor
Author: Lukáš Kopenec
Department of software engineering
Supervisor: RNDr. David Bednárek, PhD.
Supervisor's e-mail address: David.Bednarek@mff.cuni.cz
Abstract: The goal of this work was to design and implement a WYSIWYG musical notation editor that would solve some of the problems that the musicians encounter in their practice and would try to eliminate several imperfections of existing products. The principal motivation is the author’s experience from the musical schools. The teachers own large amounts of printed scores that they need to distribute to their students and to the accompanying musicians which is very costly (paper, toner) and leads to disorganization of the ownership, number and placement of the scores. This work tries to provide the schools with a cheap platform for distribution and organization of their notation materials. A secondary motivation is the fact that the conception of existing notation editors has remained unchanged during the last two decades and offers only a limited control of the overall content of the musical projects. The proposed editor allows the user to create richer and more complex scores (e.g. songbooks or musical schoolbooks). It also provides the possibility to insert additional content (such as mp3 recordings or accompanying texts) directly into the created project.

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

Table of Contents

Chapter I
Introduction

The majority of musicians, may they be composers, performers or music teachers, needs to work frequently with a written music. They need to write down their ideas, distribute the compositions among their students and, last but not least, they need to organize it. My work is principally motivated by my experience at the music school. The teachers own a large amount of printed scores. Each time a student starts to work on some composition, the teacher has to make him a copy. They also need to obtain a score for the corepetitor or the musicians in the accompanying ensemble. Evidently, this is very costly and it leads very soon to disorganization. The teachers are involved by thousands of papers, they own many exemplars of the same composition or lose the single copy of the score they had. Moreover, they do not even know what compositions they own or where to find them.

There are many musical notation products on the market today including some free of cost (although these are generally very poor in functionality) that can be used to write down the music or even to scan it. However, none has ever tried to address the above mentioned problem of distribution and organization of the scores. We can achieve both to some extent using just the existing means. Once the compositions are in the electronic form, we can organize them into folders and use the file-system facility to search among them. The distribution can then be realized by either installing the same software as is used by the school to the students (but this may introduce licensing problems) or by exporting the piece to, for example, a PDF file which can then be given to the student. But the potential of such a solution is still very limited.

And there are other reasons why I finally decided to create my own musical notation editor. If you examine the existing products you will find that their concept has remained unchanged during the last twenty years. Despite advanced music editing options, the control of the overall score layout is very limited. Typically the pages are fully occupied by the musical content and can only display a restricted set of text fields such as the title of the composition, the author etc. very soon with a fixed format. At the best case the user can insert a page of a different type that can contain a richer content such as pictures or custom text. But these pages do not usually support the musical content. Furthermore, the software assumes that you have just one composition per one score. Hence, there is no possibility to create, for example, a songbook project. But what if the user requires more freedom?

The majority of existing editors allows the user to extract individual parts from the full score and store or print them separately. Some of the products are also able to retain the relationship between them and propagate the changes from the full score to the parts and back. But this is all. Now imagine that the user writes a score that contains for example a Gregorian hymn in the original notation and its modern transcription. Logically he would like to write a single version and generate the other one whilst having them relied in the similar way as in the case

of extracted parts. He may also want to present both forms of the hymn on the same page. None of the existing software allows him to do that.

In the list bellow I present the main problems that I would like to address with my work and I briefly explain the way I have chosen to solve them:

- The Creation of Richer Content and Better Control over the Overall Layout
    - The musical project can contain more than one composition.
    - The data of these compositions can be "drawn" on the score pages inside rectangular sections in the same way as pictures and text fields.
    - The editor will allow multiple musical section types, so the same composition data can be represented in multiple ways in the same score (e.g. Gregorian and modern musical notation).
    - The editor will retain relationships between the composition data and all their representations in the project. Hence a change made at one place is immediately propagated to all the other affected parts.
    - The musical project can directly hold additional content like mp3 recordings or texts.
- The Organization of Musical Materials
    - The parts of the musical projects are accompanied by extensive metadata sets and where it is possible (such as in case of composition's scale or name of the parts) the editor collects them automatically.
    - A tool will be provided that can store these metadata fields in a database along with the project for a fast and easy search.
    - Besides this the internal content of the musical project can be organized into folders for an easier orientation.
- The Distribution of Musical Materials
    - The editor is free of cost and thus it can be provided to the students.
    - The studied composition can then be shipped to the student by email or on a removable storage.
    - Moreover, since the project can contain additional data, the teacher can easily include for example a recording of the composition or recommended reading.
    - When the projects will be stored in a database a web portal can be created to easily search and access the scores either internally for the needs of the school or externally to the students.

The created editor will be a WYSIWYG application that will allow user to easily create, edit, store and print scores. The program will target print outputs. It is not intended for manipulation of MIDI or other sound formats. However, this functionality may be easily added later in the future.

In the rest of the text I will describe my editor and explain the decisions I made. The plan follows:

1. The Chapter II describes the concept of the existing musical notation editors, explains its drawbacks and then brings the description of the concept that I have proposed including its comparison to the existing one.
2. The Chapter III describes briefly the architecture of my editor and then brings a detailed description of the most important data structures and algorithms used for music processing.
3. The Chapter IV describes the user interface of my editor.
4. The Chapter V brings a summary and conclusion.

Chapter II
Concept of the Editor and its Comparison to Existing Products

This chapter brings a detailed description of the new concept that I have designed for my application and the discussion of its advantages compared to what is offered by the existing products. I will first sketch the existing concept and point out its drawbacks by illustrating them on two examples of advanced user scenarios. Then I will describe the design I would like to propose in my work and illustrate its power on the same two examples.

II.1 Existing Concept

Practically all the existing notation editors use the same concept of the user interface and music model. Whilst it is sufficient for many uses, it has some severe drawbacks when it comes to advanced scenarios. Today, the most commonly used products are:

- Sibelius version 6; The product is now owned by the Avid company
- Finale 2010 by MakeMusic, inc.
- Encore version 5 originally developed by Passport Designs, inc., now owned by Gwox
- NoteWorthy composer version 2 by NoteWorthy Software, inc.

Note, however, that the above products are all WYSIWYG applications. There are also several text-based music notation processors, especially MusiXTeX by M.A.B. Soloists and LilyPond (open-source) that are widely used, but fall into a completely different category. The editors cited in the above list differ mostly by their user interface and the set of advanced editing features. However, the basic concept is practically the same all the time: The user works with a score that is divided into pages, except for NoteWorthy composer that uses only a linear view. The editor automatically manages the number of pages in the score in order to accommodate the whole musical content according to the settings of number of measures per system and the count of systems on one page. Besides these pages, the user can insert blank pages that do not hold any musical content, but can contain text fields and graphics, this way a title page, for example, can be created.

The user starts with editing the full score. Then he has an option to extract selected part(s) into separate scores. Sibelius and Finale keep these extracted parts relied with the original full score, so changes made at one place are automatically reflected in the other scores, but you can layout them independently. The editors do not allow you to extract only selected measures of the composition, the part is always complete. The music notation system shipped with the editors is the standard western music notation. However, Finale, for example, can be extended with a Medieval plug-in that allows you to create and edit scores in medieval notation. Sibelius can also be extended by third-party plug-ins, but I have not discovered any similar plug-in for it.

II.2 Drawbacks of the Existing Concept

The previous section described briefly the most important points of the existing concept. Among the WYSIWYG editors, there is none that would be based on truly different ideas. In this section I will point out its major drawbacks.

One Score Equals One Composition

In the introduction I have already briefly sketched some of the limitations I perceive in the classical concept. One of the most important is that all the editors define the score as the basic work unit. This implies that the user is restricted to one composition per each project. Even though this definition may seem natural and satisfies many common use-cases, I will illustrate that it may become a serious limitation for advanced user scenarios.

Imagine for example that the user wants to create a piano learning book. Here is a list of what a typical learning book could contain:

1. A textual introduction, explanation of basic principles.
2. Pictures indicating the correct position of the hands.
3. Etudes, Compositions.
4. Pedagogical notes, explanations of the exercises and other useful information.

There will usually be more etudes (since they are short) on one page and they will be mixed with the textual notes, sometimes also with the illustrative pictures. What are the user's possibilities with the existing software?

The first problem is that he cannot have more than one composition per project, so he has to create a separate score for each of the etudes and compositions he will have in his book. As I already mentioned in the introduction, most of the editors divide the pages in the score into two types: musical pages and pages with non-musical content. And the capacity of musical pages to hold a non-musical content is very limited. Pictures can rarely be inserted and text fields have often fixed content.

Therefore, the only reasonable possibility the user has is to create all the scores separately, export them as pictures and paste them in the final book project that will be written in some advanced text editor like Word, TeX or similar.

But what happens if the user identifies a mistake in some composition and needs to fix it? The only way is to find and open the correct score, correct the error, re-export the score and re-paste it to the text. Obviously this is a tedious and error-prone process. Besides, the organization of the project parts is left on the user's responsibility. And what if the user decides to make an electronic version of the book where the pedagogical pictures would be replaced by short video-clips (showing the right mode of play)? His only possibility is to create a web-site which can be difficult for him and requires finding a suitable web-hosting that can introduce additional expenses.

One Score Equals One Representation of the Composition

Another problem with the current concept that I have also already presented in the introduction is that the editors do not allow you to mix different representations of the same composition inside one score. The only possibility the user typically has is to extract individual parts from the full score. Then, depending on the concrete editor, he may either store them separately (and independently) of the original score which can clearly easily lead to inconsistencies. Or, as in the case of Sibelius, for example, store them within the original project where the editor retains the relationships between the extracted parts and the full score and ensures the consistency of the dependent elements. But could we do more?

As I stated in the introduction my principal motivation of creating this kind of editor was my experience with the situation at the music school. Hence I will frequently use the examples of different learning books for supporting my arguments. Imagine now another advanced scenario: A user wants to create an encyclopedia where he wants to describe the evolution of the musical notation system. As an illustration of how the notation has changed he makes a page at which he presents the same composition written in several distinct notation systems. It is probable that he would also like to insert some short texts in-between to explain the major differences. What are his possibilities with the existing software?

As I already mentioned, Sibelius and Finale can be extended by third-party plug-ins and so, theoretically, they allow you to edit scores in any notation system. But even if there were plug-ins for all the existing systems, neither Finale nor Sibelius allows you to mix different systems in the scope of one project. Therefore, once again, the user would be obliged to write the composition separately in all the notation systems he would like to present, export the copies as pictures and insert them to a text editor. The objections against this mode of work were already presented in the previous section. Except that the situation is even worse in this case, because, if an error is found in the composition, the user has to correct it independently in each copy and re-export and re-paste them all.

Creation of Rich Content

The last major drawback of the current concept I would like to cite is its poor capacity of creating a rich content. I have already touched this problem several times in the previous sections so in this section I will organize these points and rewrite them into details.

The most severe limitation is that the pages in the majority of existing editors are split into two types:

1. *Musical Pages* that can only hold musical content (systems of staves) and just a restricted set of other objects.
2. *Text Pages* that can hold text and pictures.

The editor itself controls the layout of the musical pages and the user has very limited possibilities to influence this layout. Typically his only option is setting the margins to stress or extend the area of the musical content, but he cannot split it. Thus, he can place text fields (and sometimes pictures) around the musical area, but usually not inside. He cannot for example insert an illustration of the correct hand position in-between two staves on the page.

Another problem with this type of pages is that very often the user can only insert text from a restricted set of fixed-content text fields (like the score name, author, copyright notice etc.). In many editors it is difficult or even impossible to insert a custom text. But, as I illustrated in the two previous advanced scenarios, in some cases it would be extremely useful.

The formatting options are usually also limited. The text properties are often dictated by a global house style and cannot be modified locally. It is true that a global style is more than reasonable in most of the cases. But again, there are advanced scenarios where more freedom would be desirable.

And as the last point I would like to mention the lack of alignment possibilities. In some cases the user may want to rotate or overlap parts of the content. He might for example want to place some text or music vertically in order to better exploit the page area. Or may he just want to make his work nicer. The options provided by the existing software are very restricted in this sense.

II.3 New Concept

In this section I will describe the new concept of musical notation editors that I propose and I will explain how it addresses the drawbacks of the current one that I described in the previous section.

A Higher Level Work Unit – The Musical Project

My major criticism to the existing concept is the definition of the score as the basic unit of work and the limitations that result from such a definition. To avoid this problem I defined a new work unit at a higher level and called it a musical project.

A project consists of the following three parts:

1. Compositions
2. Scores
3. Resources

However, the whole project is physically stored as a single file, so the user is not concerned with the organization of the project package in the file system or might not delete some of its vital parts by mistake. I will now describe all the three parts.

Compositions

A composition contains musical data. These data are independent of the concrete musical notation system. The composition's content is edited in a user-selected notation system that is presented in a non-paged linear view. The linear view can only contain music; it cannot hold text or pictures. Its sole purpose is to allow user to edit the data in the composition without having to think about the final score's layout. In common western musical notation the linear view looks like a single system of staves that is "infinite" or long enough to hold all the composition's symbols. Besides this a composition contains a metadata sheet that describes it (title, author, scale, instruments, etc.).

Scores

A score represents a final printed publication. Basically it contains just pages and all the pages have the same generic type. There is no distinction between a musical and text page. A page defines its format (i.e. size, margin, background color, etc.) and it may contain an arbitrary number of rectangular sections. The sections on their turn divide into three different types:

1. Image Section
2. Text Section
3. Music Section

An image section may contain an arbitrary picture. A text section contains text that may be either custom or bound to a metadata field. And finally the music section visualizes a selected part of a composition in the project. The user can select a continuous range of measures from the source composition and even the parts that will be presented inside the music section.

The size, layout and rotation of the sections can be customized at any moment. A section also defines the z-order so the user can overlap two or more sections and define their relative positions in a similar way as in a vector-graphics editor.

Even though the user is given a complete freedom in drawing and aligning the sections on a page, it would not be comfortable for him if he was obliged to first edit a composition and then manually draw a music section on each page of the final score. To leverage this task, the editor can generate a default score from a composition that can then be customized.

The editor also supports multiple music section types, which allows the user to represent a composition in different notation systems within the same score or even the same page. But, since all these sections are bound to the same composition data, a change at one representation is immediately propagated to all the other affected places. Hence, technically speaking, the score (or more precisely the music section) represents a view of the composition data. Note, however, that the change propagation only applies to the shared data stored in the composition. The view extends these data by additional notation system specific information as is for example the layout or description of the staves. Changes to these data are local to each section and are not automatically spread.

A score is again accompanied by a metadata sheet that describes its content.

Resources

Finally a project can contain an arbitrary number of resources. A resource may be anything, a binary or a text file, that the user assumes being relevant to his project. For example, when a teacher at a music school prepares a composition for his student, he may attach an mp3 recording of the composition or some text about its author or history to the project. Also, any picture that is drawn to a score inside an image section is automatically inserted among project resources.

Project Organization

The project may contain an arbitrary number of any of the described three parts. Moreover, for an easier orientation, the user may organize the project parts into folders and subfolders; the depth of the folder tree is not limited.

Realization of the Advanced Scenarios

Now, let's return to the two advanced scenarios that were described in the previous chapter and examine their possible realization in an editor that uses the musical project as its basic work unit.

The first was the piano learning book. The implementation is now evident. First, the compositions are inserted into the project, one for each etude in the book. Second, a score for the book is created, either blank or generated by the editor. Now, the user has a full control over the layout. He "draws" the etudes on the page and then he puts text and image sections around them or in-between exactly as he needs.

Now, if he finds an error in a composition, he just corrects it either in the linear view or in the score and the change is immediately propagated to the other affected parts. If he then decides to create an electronic version with illustrative video-clips or recordings, he simply inserts these files into project as resources. In order to help the student to find the correct clip, he can organize the resources into folders with names corresponding to the chapters of his book. All the process is intuitive and practically infallible.

The other scenario was that with the encyclopedia. Here the realization is even simpler. The user creates just one composition using the linear-view and his preferred notation system. Then he "draws" the composition on a page several times, each time using a different type of notation system. If he discovers a mistake, he corrects it at any place and the other occurrences are changed automatically.

II.4 Summary

I believe that these two examples have clearly demonstrated the power of the proposed concept. I admit that these are very advanced scenarios and the users will rarely create such a complex musical project. However, there are much more common situations that can exploit this concept and that are still difficult or impossible to realize with the old one. One case may be a multi-part composition like a mess or a concert. It is more natural and comfortable for the user that he can store all the parts within a single logical package. Furthermore, he can also save space by putting the beginning of one part on the same page as the end of the previous part and so on. Another possible case is different songbooks. First, the user may benefit from the possibility to place multiple songs on the same page. Second, he may create several distinct scores in the same project each containing a different selection of the songs. This feature can be especially exploited by the music publishers. The custom resources also provide new and interesting possibilities, especially in the pedagogical world.

Furthermore, the introduction of the shared composition data offers a great potential for new features, e.g. "virtual parts". Imagine a composition written for a typical 4-voices chorus (soprano, alto, tenor and bass). For rehearsal purpose, the score often contains a piano part

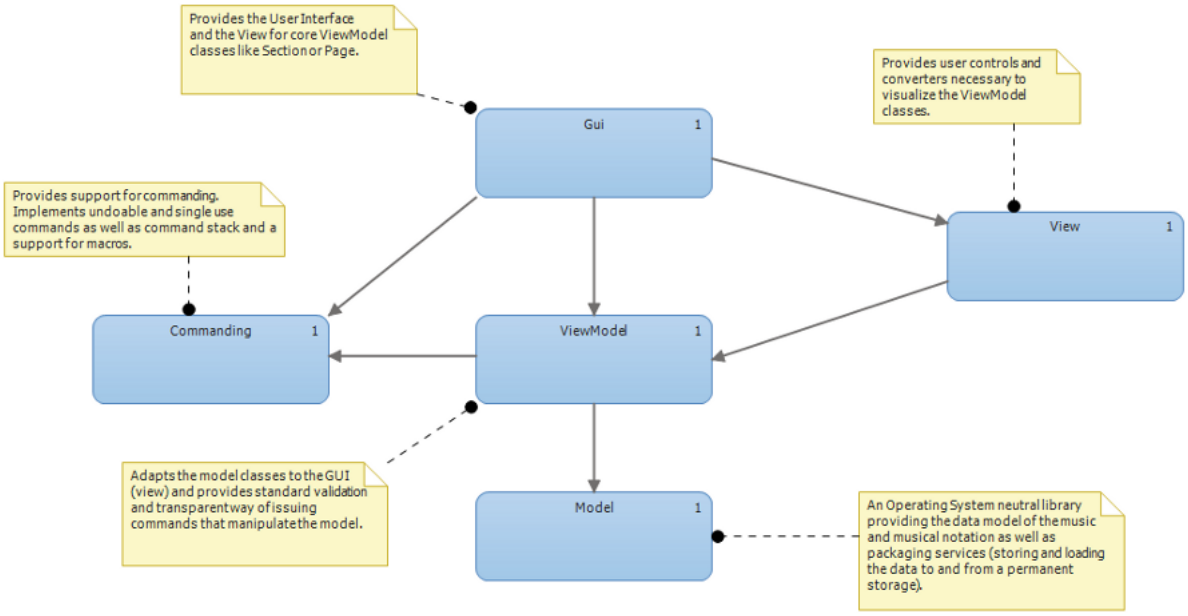
that contains nothing different than the transcription of these four parts into the piano staves. Since the notes are the same there is no need to rewrite them manually and copy & paste them creates once again an independent representation unrelated to the previous one, which can be problematic once a mistake is discovered. Using the proposed model, it is relatively easy to represent a virtual part that groups voices from multiple existing parts.

Chapter III
Data Structures & Algorithms

This chapter will describe the architecture of the editor as well as the most important data structures and algorithms. Where appropriate, it will also discuss alternative solutions and the reasons that finally led me to choose the implemented one.

III.1 Basic Architecture

The editor is based on the Model – View – View-Model pattern as illustrated in the figure below:



As you can see the communication is only allowed downwards from higher layer to a lower one. The layers have no means to directly call a superior layer; the only way to inform higher application levels of an asynchronous change is by publishing events.

The model provides the data model of the music and musical notation as well as packaging services (storing and loading data to and from a permanent storage). The view-model adapts the model classes to the target view and provides a transparent way of issuing commands that manipulate the model. Finally, the view provides user controls and converters necessary to visualize the view-model classes. There are two additional layers: commanding and GUI. The commanding library provides the implementation of core commands functionality; especially the command stacks and their manipulation and commands binding that will be further explained later in this chapter. The GUI implements the graphical user interface that allows user to easily manipulate the data and provides additional functions like zooming or printing. The following section will bring a more detailed description of each individual layer.

Model

This layer consists of the following sub-parts:

- Packaging – Provides the manipulation of a physical storage
- Elements – Implementation of the composition & notation models
- Collections – Set of generic collections used by other parts and view-model layer
- Validation – Provides a unified way of validating the values written to the model

Packaging

The data of a musical project are physically stored in one ZIP archive the internal structure of which is based on the Open Packaging Conventions (OPC) designed by Microsoft. Microsoft now uses OPC as a standard for storage in most of its products (e.g. Open XML in Office or Open XML Paper Specification format – XPS) and it encourages developers to implement it in their own applications as well.

Briefly, an OPC package consists of package parts that correspond to compressed files in the archive and directed relationships that can join two package parts or a package part with an external content. Each relationship has a string attribute called "type" and the OPC API allows you to easily enumerate relationships starting at a given part and filter them by type. The package parts are addressed by Universal Resource Identifiers (URIs).

The Model library extends the basic OPC API implemented in .NET by providing strongly typed package parts, virtual paths and restoration of previously deleted parts. It also encapsulates the relationships mechanism and the control over the creation of new parts. Therefore the higher levels are provided with an easier (less universal) and safer interface – they cannot create an invalid part or relationship. Six package part types are defined. First five correspond to individual project items: score, page, resource, composition, linear composition view and the last one to the project itself. The only binary part is the resource part. The remaining project items use XML for storing their data. When a package is open, it is not read as a whole. Instead, each individual part is loaded the first time it is used.

The physical organization of the parts in the package does not correspond to the hierarchy that is presented to the user. The reason is that the physical package organization is not very flexible and for example each time the user renames an item or directory the corresponding package part(s) have to be deleted from the package and recreated at the new path. There is no possibility to change the URI of a part once it has been created. To solve this issue the Model library uses virtual paths. A virtual path is nothing than a string that holds the path that will be presented to the user. This path is rooted at the logical owner of the project item (i.e. the score in case of a page) and can contain any number of directories. The mapping of the virtual paths to the URIs of their corresponding package parts is stored in path collections that are a part of the root project parts (i.e. the score holds a map of its pages, the project holds a map of its scores, etc.).

Elements

The model elements are the classes that implement the project model (i.e. the composition and musical notation data, page content, etc.). They are particular in the sense that each instance

encapsulates an XML element that back-ups its values. The organization of the elements of a project item corresponds to the XML tree of its serialized representation. In fact what happens when an item is loaded is that its XML data are loaded to a DOM representation and a corresponding model element is created for each XML element in the tree. Values written to the properties of a model element are immediately written through to the DOM, so it holds valid and updated data at any moment.

Such an implementation may seem uncomfortable at a first sight, but it has important advantages. The most important is that the serialization and deserialization is very simple. Since the values are written through to the DOM, the serialization is completely left on the framework that writes the DOM to the output stream. Each model element is then responsible for deserializing its own data which consists of assigning the attributes and elements to their corresponding properties. Since the whole XML tree has been loaded, the navigation through its nodes is easy. The only problem is to determine the type of the model element that should be loaded. In many cases the type is uniquely given simply by the position of the XML element in the document. For example, when the content of the "measures" XML element of a composition part is loaded, it is known that it is a collection of measure elements. In situations when the type cannot be inferred from the element location (e.g. when it may be one of several derived classes) a type identification is serialized.

Moreover, a tree organization is easier to manage than more dimensional representations. Of course, some elements need to reference elements that are not their immediate children. In this case a unique identifier of the target element (GUID) is used to make such a reference persistent. Since GUIDs are long, in order to decrease the size of the package parts, the identifiers are generated on demand the first time another element creates a reference to this element.

All the element classes derive from an abstract base class called *ModelElement*. It encapsulates the common part of XML element creation and provides functionality related to create persistent references, navigating through the elements tree, data validation and writing and reading properties using their name. The latter is very important since it allows the view-model to considerably reduce the number of individual command classes. I will discuss this feature in the section dedicated to the view-model.

Validation

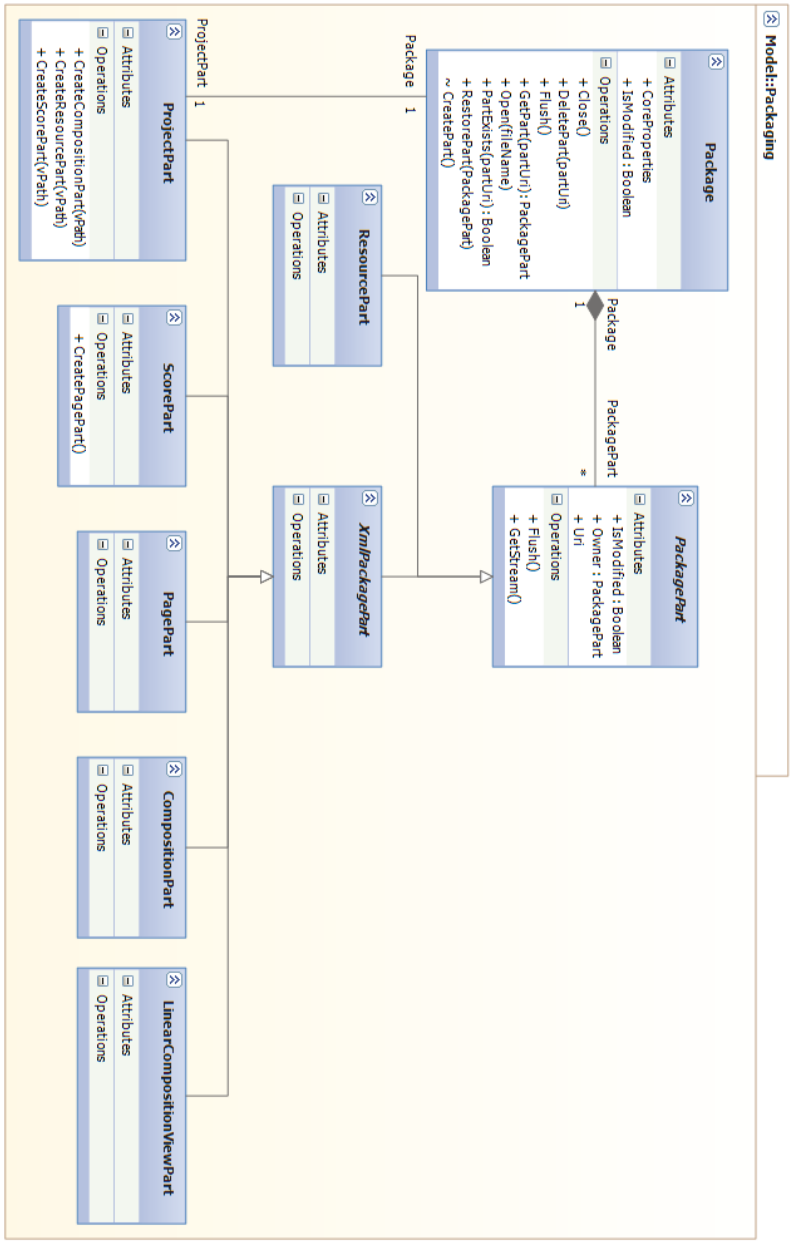
The Model library provides a standard way to validate the values that are to be written to the properties of its classes. The validation is based on rules. Each object may define a set of rules that constraint the value that can be written to its properties. A rule contains a method that verifies a given value and a textual description that explains the constraint it imposes. The library then defines an interface called *IValidator* that defines methods to get all the rules imposed on a property or validate a value using the property name. When the validation fails, the caller is given a set of all the validation rules that have been broken.

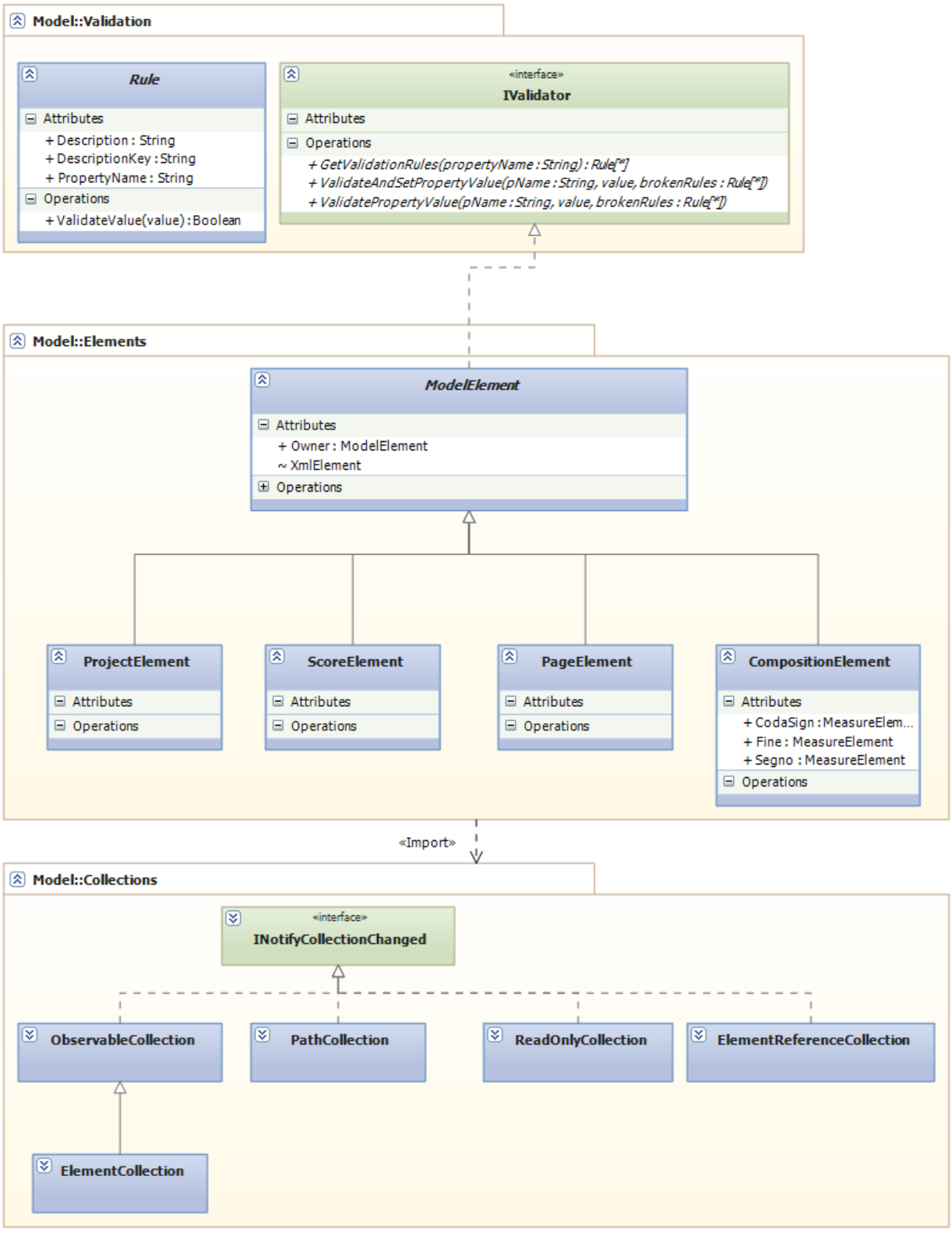
Collections

The Model library implements its own set of generic collections. There are two reasons why it does not use the standard collections provided by the .NET framework:

1. In order for the view-model to work correctly, the collections have to publish an event that is raised whenever the collection's content has changed and contains the description of that change. The .NET framework does define an interface and two collection classes that provide this feature, but none of these is implemented by the Mono project. Since this library was designed as portable, it was necessary to create its own counterparts of the .NET collections.
2. Automatization of the serialization and deserialization of collections of model elements and collections of references to model elements.

The following two UML diagrams capture the main structures of the model layer.




Model::Validation
Rule
Attributes
+ Description : String
+ DescriptionKey : String
+ PropertyName : String
Operations
+ ValidateValue(value) : Boolean
«interface»
IValidator
Attributes
Operations
+ GetValidationRules(propertyName : String) : Rule[*]
+ ValidateAndSetPropertyValue(pName : String, value, brokenRules : Rule[*])
+ ValidatePropertyValue(pName : String, value, brokenRules : Rule[*])
Model::Elements
ModelElement
Attributes
+ Owner : ModelElement
~ XmlElement
Operations
ProjectElement
Attributes
Operations
ScoreElement
Attributes
Operations
PageElement
Attributes
Operations
CompositionElement
Attributes
+ CodaSign : MeasureElem...
+ Fine : MeasureElement
+ Segno : MeasureElement
Operations
«Import»
Model::Collections
«interface»
INotifyCollectionChanged
ObservableCollection
PathCollection
ReadOnlyCollection
ElementReferenceCollection
ElementCollection

View-Model

The purpose of the view-model is first to adapt the model classes to the presentation layer. This includes for example the conversion of model length units to those used in the view, conversion of structures such as brushes or colors etc. And the second purpose is to provide a way to issue undoable commands that will manipulate the model data. The view-model also expands the virtual paths into a corresponding hierarchy and provides the auto-save mechanism to allow recovery of open project after an application or computer crash.

The hierarchy of the view-model classes corresponds roughly to the hierarchy defined by the model. However, there are two major differences:

1. In contrast to the model where each project item had it own package part and hence its own element tree, there is a single tree rooted at the project in the view-model.
2. There may be classes in the view-model that do not have direct counter-parts in the model. This includes especially the notation model where many properties did not need their own persistent representation. For example, in model, it is perfectly sufficient to store information that a note has two augmentation dots. However, in the view-model this scalar property needs to be expanded into a vector of stand-alone augmentation dot objects because the presentation layer needs individual position information for each of them.

Clearly, the view-model already needs to have some notion about the target presentation layer, although it still cannot reference it directly. At least, it has to know the units and structures that are used so that it can convert the model values. In my implementation the view-model has also a basic knowledge of the user interface. This includes only the notion of basic application parts, like the fact that the application will have some project explorer window that will display the project tree. The view-model does not know anything about the disposition of these parts in the user interface nor does it care about their exact implementation. If I keep the example of the project explorer window, to ease its implementation the view-model extends the definition of project parts with a list of their logical children, flags controlling the state of the item (like if it is currently expanded or selected) and the possibility to store an icon or context menu within their objects. Another feature of the view-model that requires the knowledge of the target view is that it converts the model validation interface to the standard validation interface of the presentation layer.

But probably the most important and complex functionality is related to the commanding. The view-model implements numerous commands that manipulate the model data and are able to roll-back their action. The logical level at which they operate varies from setting a single property to the creation (or removal) of a whole hierarchy of elements. The commands are the only way for the superior levels to interact with the model data. However, the caller is not limited to create and execute the commands by hand. Setting a property on a view-model object results in creation of a command that writes the corresponding value to the model. In the first view-model version the command was executed immediately. But then I realized that deferring the execution brings important advantages that are illustrated by the following two use-cases:

- If the user performs a larger edit operation like for example editing the page appearance in a dialog window, he expects all the changes he made to be either committed at once when he presses OK or aborted when he presses Cancel.
- One of the new features that were brought by the Microsoft® Office Fluent™ user interface is the live preview. For example when in the Microsoft Word the user hovers a quick style ribbon gallery item, the active paragraph in the document is changed to use this style, but if the user moves the mouse away without selecting the style, the change is dismissed.

Both features can be easily implemented using a deferred command execution and the redirection of the property value to the command.

Each view-model class that corresponds to some model element has an internal map of unsaved changes, i.e. the properties that were modified but the underlying command was not yet executed. When such a property is read, the value stored in the command is returned instead of the model value. The class also defines two methods, one for committing the pending changes and the other for aborting them. When the changes are committed, a single macro command is created that executes all the commands for the individual properties. Hence, if the user later presses Ctrl + Z all the changes made during that transaction are rolled back at once. The live preview can be implemented simply by calling the abort method when the user moves the mouse cursor away.

View

The view is responsible for visualizing the view-model objects and for passing them the actions sent from the user interface. It handles the mouse and keyboard input and translates it into view-model properties set and commands creation. Since the view is implemented in the Windows Presentation Foundation (WPF), its code is very simple because it exploits the concept of data binding and templates.

III.2 Composition Model

As explained in the chapter about the editor concept, the music data stored in the compositions are independent from a concrete music notation system. In order to achieve this, the data model can be created either by comparing several notation systems and extracting their common parts or by trying to describe what music actually is regardless of how it is represented. I have adopted the second option and in this section I will describe my composition data model and the thoughts that led me to create it in this way.

Initial Reflexion

In its very basis the music is nothing but a set of tones and silences ordered in the time. More tones can be played at the same time and each tone has 4 basic characteristics: Pitch, Timbre, Duration and Loudness. Silences have only one characteristic and it is the duration. These properties must absolutely be captured by the data model.

The order of the tones and silences (I will use a common term durational symbols for them in the rest of the text) is not completely linear, because there may be repetitions. Either simple, when a continuous passage is just played twice or more complicated that include jumps (like different repetition endings or segno, fine and coda). However, the sequence of durational symbols is typically naturally divided into smaller rhythmic groups with regular duration that are called measures or bars. The duration (or meter) of the measures can vary.

It is true that not all notation systems have the notion of measures, but this division is also a matter of music perception and human nature. Practically all compositions show this inner regularity and there is always a way to partition the symbols into some system of bars.

The repetitions (both simple and complex) can only occur on the measure boundary. No repetition can start in the middle of a bar and none of the notation systems that uses measures allows you to do that. It would also make no sense and the listener would perceive it as unnatural and disturbing. After all, since the meter of the bars can change, this condition does not limit the composer in anything, but it simplifies the design of the data model.

A model that would capture all the properties described in the preceding paragraphs would be already sufficient for simple compositions. But there are still many interpretation related data that are vital for the player and that cannot be represented in this core model. This information includes articulation, instrument specific features like pedal usage for piano, alternative passages (ossias) etc. These data form a sort of attached information that extends or complete the core model.

I have briefly depicted the thoughts that stand at the beginning of my composition model. But before presenting it, I will explain why I have developed my own data model instead of using some existing format. There exists several music interchange formats, but the majority of them are designed for common western music notation. Practically the only standard format that is not directly bound to a concrete notation system is Musical Instruments Digital Interface (MIDI). However, even though this format is very good for performance applications like sequencers, it is not complete enough to capture the data described above.

Data Model

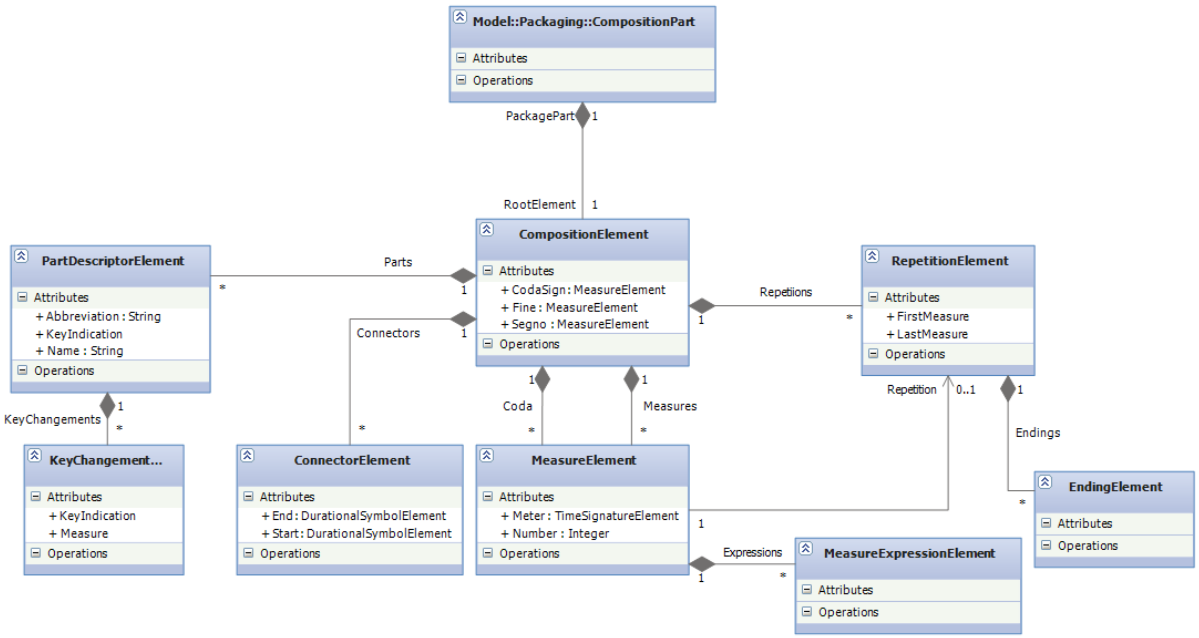
This section brings the detailed description of the data model of a composition. Most of the model is presented in the form of UML class diagrams with explanations where necessary.

Top-Level Structure

In the figure below is the UML diagram of the top-level model data structures. As you can notice, the data are stored in a composition package part in the project package. The root element is the composition that contains two collections of measures: The main measures collection that is always present and the coda collection that is used when the composition contains a coda (tail). The composition also holds a collection of repetition objects that describe simple repetitions and repetitions with endings. For capturing jumps, the composition contains three more attributes (*CodaSign*, *Fine* and *Segno*). When a measure is assigned to one of these attributes, it is marked as the target of a corresponding jump. The measure can be of course assigned to more than one jump targets. The start of the jump is given by a measure expression owned by the corresponding measure.

The composition also owns a collection of connectors. Connectors are attachment symbols that connect two durational symbols. But, since they can possibly connect two symbols of different measures or event different parts, they have been implemented here at the top-level.

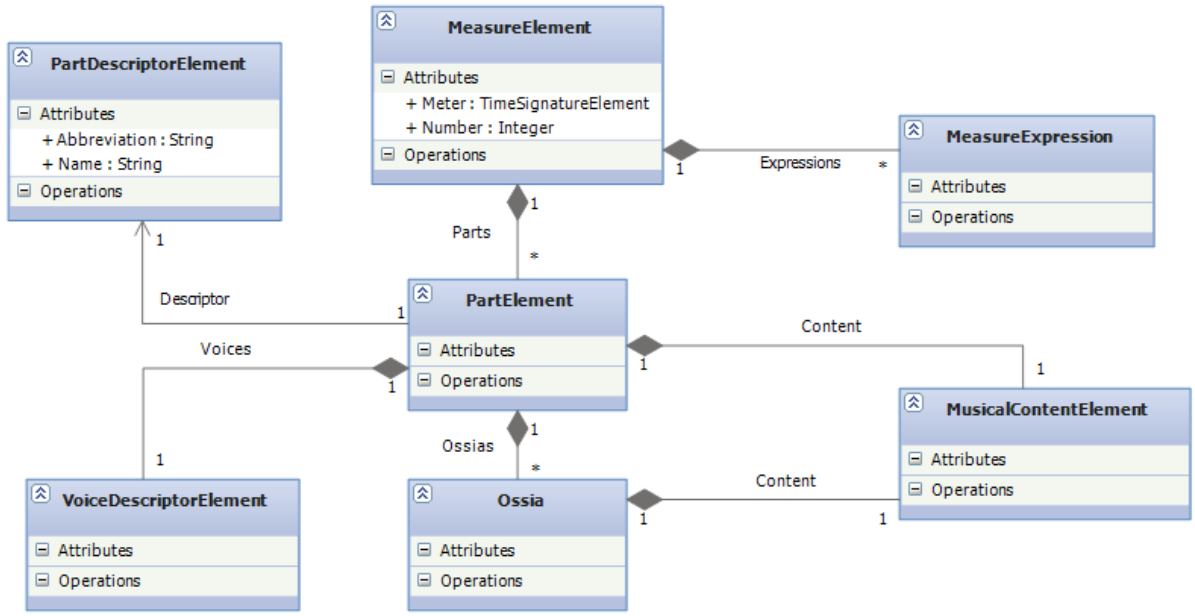
The last object that is directly owned by a composition element is the collection of descriptors of parts of the instruments that perform this composition. One part typically corresponds to one instrument, but it may represent a group as well. Besides the part name it holds an indication of its key (scale) and optionally of the key changes. All the remaining musical data are a part of the measure objects that will be described next.



Measure

Bellow is the diagram capturing the structure of measure content. The measure attributes are its meter that defines the rhythm and the number that is just a 1-based index of the measure in the composition and serves for an easier navigation in the printed score. It owns a collection of measure expressions. A measure expression is a symbol or text related to the interpretation of the measure or the composition's passage starting at this measure. One example are the jump instructions like *da capo al segno*, *dal segno al coda* etc. Other example are notes like *mysterioso* (mysteriously), *rubato* (with varying speed) etc.

The second element owned directly by a measure is the part (collection of parts). A part contains all the durational symbols that belong to this measure and are played by the instrument described by the referenced descriptor. The reason why the model has that structure is that the XML represents data in hierarchy whilst all the music notation systems have rather a two-dimensional structure (one axis is the time and the other the different instruments that play different tones). Hence we need to make a cut and represent either the measures within the parts or the parts within the measures. Because the model departs from viewing the music as an ordered sequence of tones, I have chosen the second option. Actually the part could be simply an attribute of the tone since it does nothing that represents its timbre and logically we might want to store all the tone's properties at one place. However, this representation makes it simpler for higher levels that implement notation systems to retrieve the symbols that belong to the same part.



The part element holds three different element types. The first is the collection of voice descriptors. A voice descriptor describes one voice in the part. Its main purpose is to allow representation of different meters (measure types) inside one part. In the majority of compositions the measure meter is common for all the parts and all the voices. However, there are compositions where the staves of the same part are notated in different time signatures. One example of these is the Carl Orff's Carmina Burana. The data model of the composition does not have a notion of staves (since it is a notation system dependent feature), so the

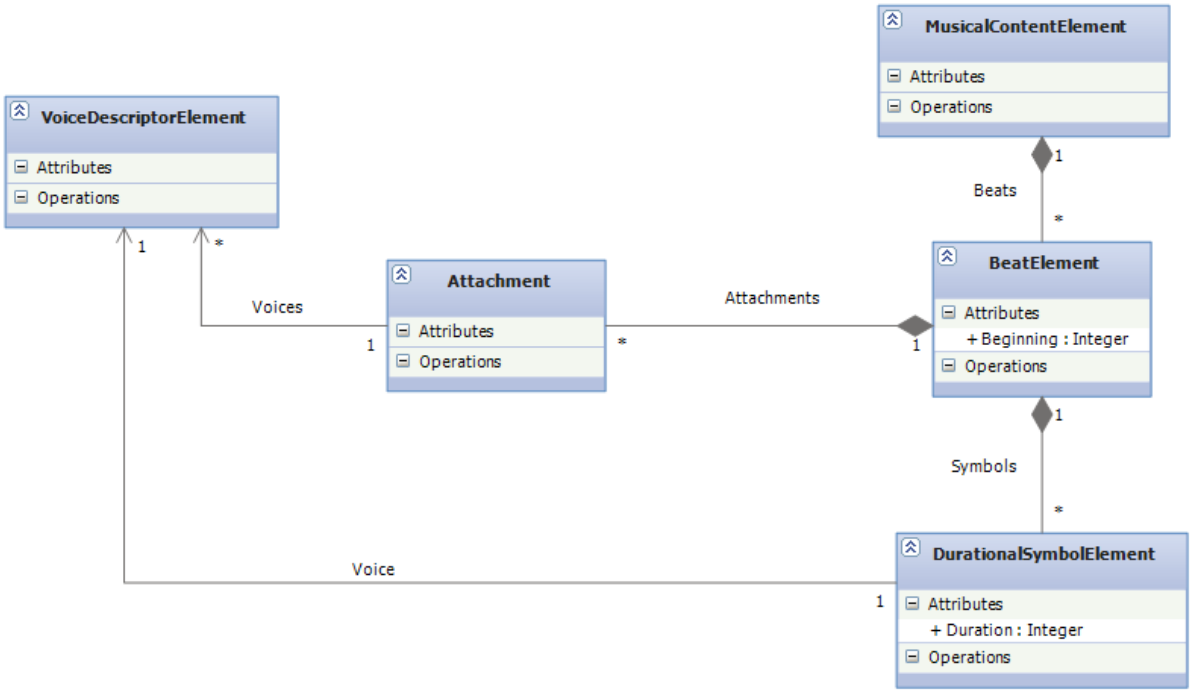
granularity is at a single voice level. The voice descriptor is also used to collect some statistics about the described voice that can be used by superior layers (e.g. the shortest duration in the voice or the number of symbols).

To represent alternative passages the part holds a collection of ossias. In common western music notation ossia is represented as a small staff placed above their corresponding passage. Since both the part and ossia contain musical data with the same structure, the own musical content (time-ordered durational symbols and their attachments) has been moved to a separate element that can be owned by either of them.

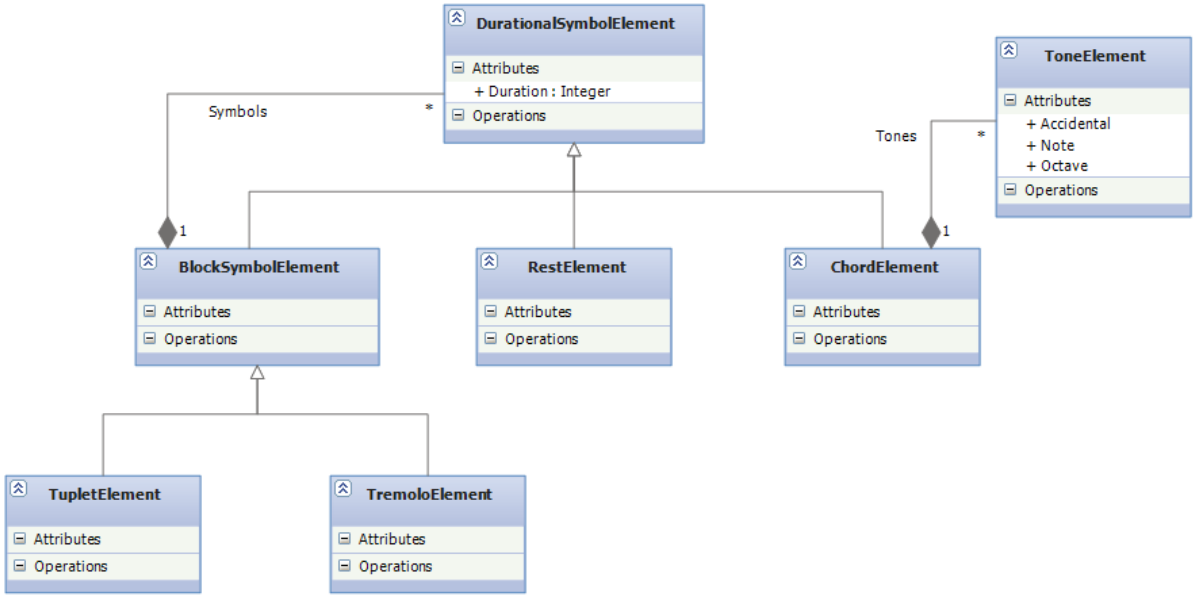
Musical Content

The next described component is the musical content of one measure of one part (ossia). The basis is an ordered collection of beats. A beat correspond to a musical beat. It has a beginning expressed as a multiple of 128-th beats. For example the first musical beat is encoded as 32. I have chosen this definition to keep the value integral. This also implies that the shortest duration that can be expressed in this model is one 128-th beat. However, since most of the books about notation practice define a 64-th note as the shortest one, this value is more than sufficient.

Each beat contains two types of symbols: durational symbols and attachments. The former represents symbols that have duration (i.e. notes and rests), the latter stand for symbols that add information and are, in some form, attached to the beat. Both of them reference the voice descriptor object that was described in the previous section. An attachment can affect multiple voices. A durational symbol can only belong to one voice and a beat can only hold one durational symbol of each voice.



Durational Symbols



The above picture captures the hierarchy of durational symbols. The most important are *Rest* and *Chord*. A rest represents a silence and a chord a collection of tones. An alternative would be to implement a *Tone* directly as a durational symbol. I have finally decided for the first variant because a common practice, when writing a homophonic composition, is to put all the tones that belong to the same chord into one voice. It semantically connects them and it is more readable for the user as well.

I have also dedicated special classes for tremolos and tuplets. These symbols are composed from multiple durational symbols but their duration is shorter than the sum of durations of which they are made. A tremolo is used to notate a rapid repetition of a single or multiple tones. A tuplet (also irrational rhythm or extra-metric groupings) is any rhythm that involves dividing the beat into a different number of equal subdivisions from that usually permitted by the time-signature (e.g., triplets, duplets, etc.). In my model they are both represented as a single symbol with its own duration that can however contain other durational symbols inside itself.

A tone is represented as a simple structure made of three values:

1. Note that expresses the pitch class of the tone (relative pitch).
2. Octave that defines the absolute pitch of the tone (octave + note = absolute pitch).
3. Accidental that lowers or raises the tone by up to one tone.

Two notes with fundamental frequencies in a ratio of any power of two (e.g. half, twice, or four times) are perceived as very similar. Because of that, all notes with these kinds of relations can be grouped under the same pitch class. The pitch class in this representation is expressed as an integer ranging from 1 to 7, thus giving a scale of 7 tones (C, D, E… B).

An octave is the interval between one musical pitch and another with half or double its frequency. In my model it is represented as an integer where zero stands for the middle octave (A ~ 440Hz).

The accidental expresses alterations to the basic 7 tones scale. It is a real number ranging from -1.0 to 1.0 where 0.0 stands for no change, 0.5 would change the note by one semitone and 0.25 by one quarter tone.

Using this representation, any tone can be easily expressed whilst keeping a neat difference between flat and sharp tones in the tuning systems where they are not equivalent.

Custom Data

The composition model is independent of a concrete music notation system. The notation systems are implemented in the music sections that form the content of pages of scores in the project. Each music section forms a stand-alone element tree and the communication with other sections is limited to a set of so called border requests. This also means that the music sections have no shared global storage. However, sometimes it is necessary (or at least desirable) to share some information among all the music sections of the same type. Where to put these data?

For this purpose the composition model defines two points where the sections can store arbitrary data. These data are not interpreted by the model but are accessible to any bound music section. The first place is the part descriptor. It can be used for storage of additional part data. Fox example, the Common Western Music Notation system uses this extension point for the description of the staves to which the part is notated. The other extension point is on the durational symbol. It can be used to store notation-specific information about the symbol, like the staff on which it is placed.

Summary

The principal design goals of the composition model were universality and simplicity. I think that both were accomplished. The model can represent a wide-range of musical compositions whilst it is formed by less than 25 classes. Since its structure captures all the principal categories of musical symbols, it can be easily extended by simply deriving a new class from one of the existing types. Another extensibility power is the concept of custom data described in the previous section.

Before finishing this chapter, let's briefly sum-up how the main musical characteristics discussed in the Initial Reflexion.

The four tone properties:

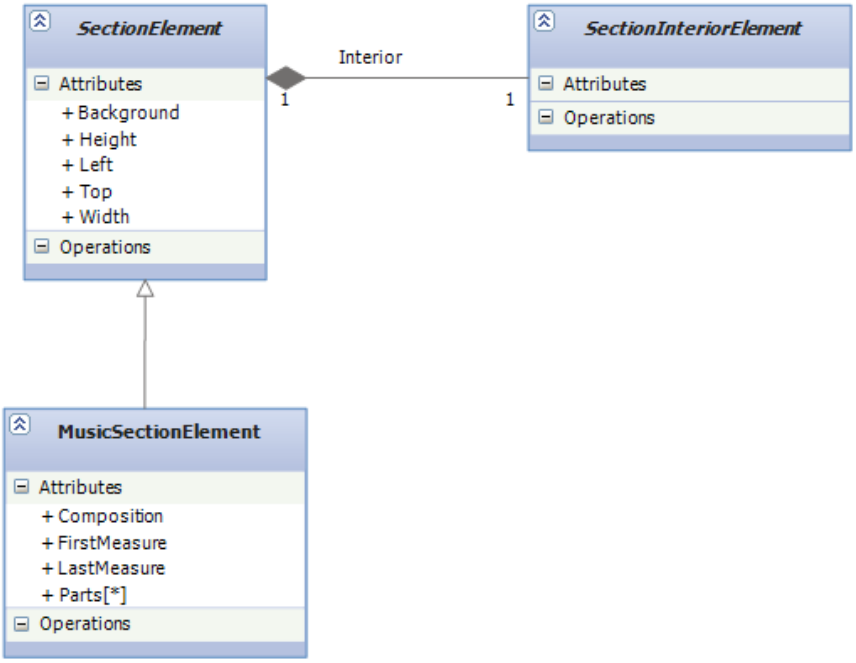
1. Pitch – The Tone class (Note, Octave, Accidental)
2. Timbre – PartElement ownership, reference to the Part Descriptor
3. Duration – Durational symbol ownership
4. Loudness – Dynamic Mark attachment

The time ordering is given by the measure and beat ownership.

III.3 Music Section & Notation Model

The data stored in a composition do not contain any visualization information. There is no notion of position, size or layout. The presentation of the composition's content is a responsibility of music sections. A music section does not necessarily visualize the entire composition. It can specify the set of parts and a continuous range of measures it will present.

The music section class itself is actually just a place-holder. It contains data describing the appearance of the section (like its position, size or background color) and the presented composition part. A concrete notation system is then implanted to its interior by deriving from the abstract *SectionInteriorElement* class. The interior of the section is created by a factory method that is passed into the section constructor as its parameter. This structure is captured by the diagram below.

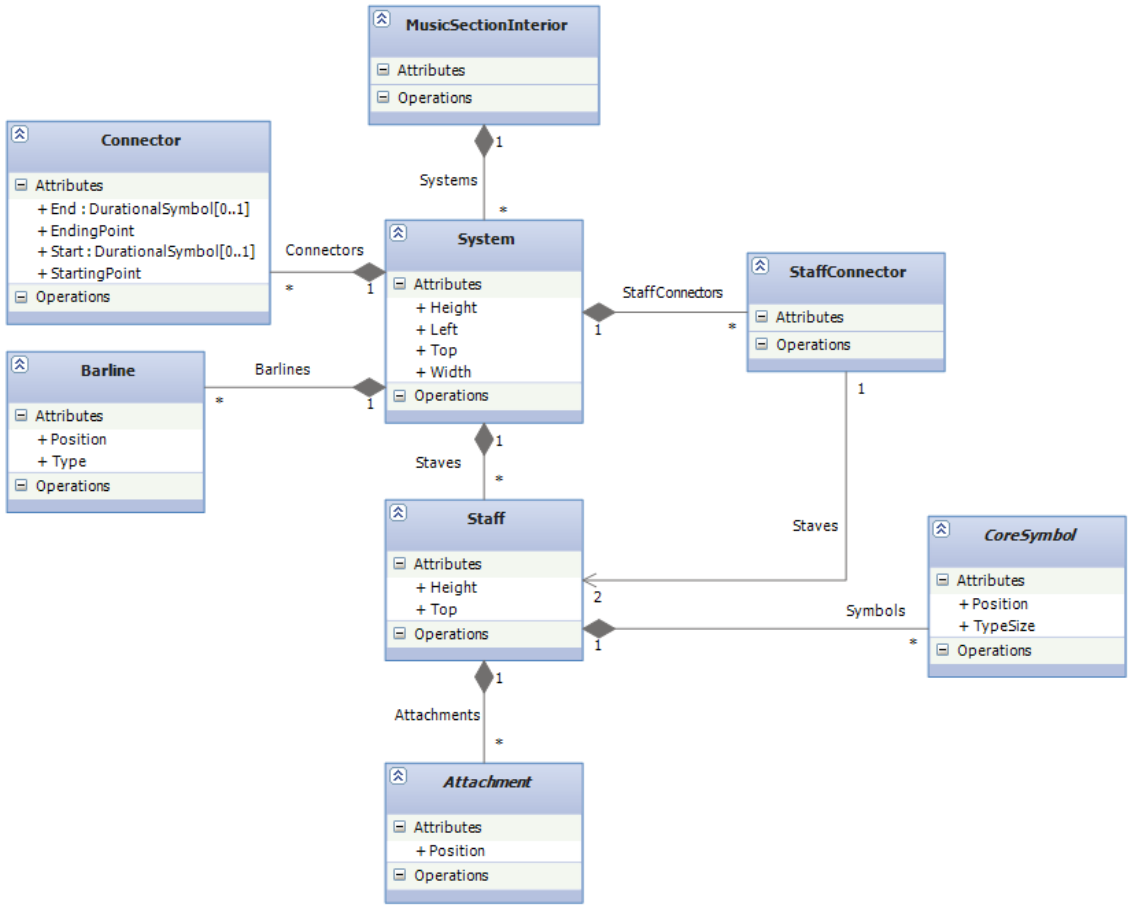


Common Western Music Notation Model

The remaining part of this section is a description of my implementation of the common western music notation system which is the default notation system used by the editor. The text will cover the structure of the notation model as well as the way it binds to the composition data.

Top-Level Structure

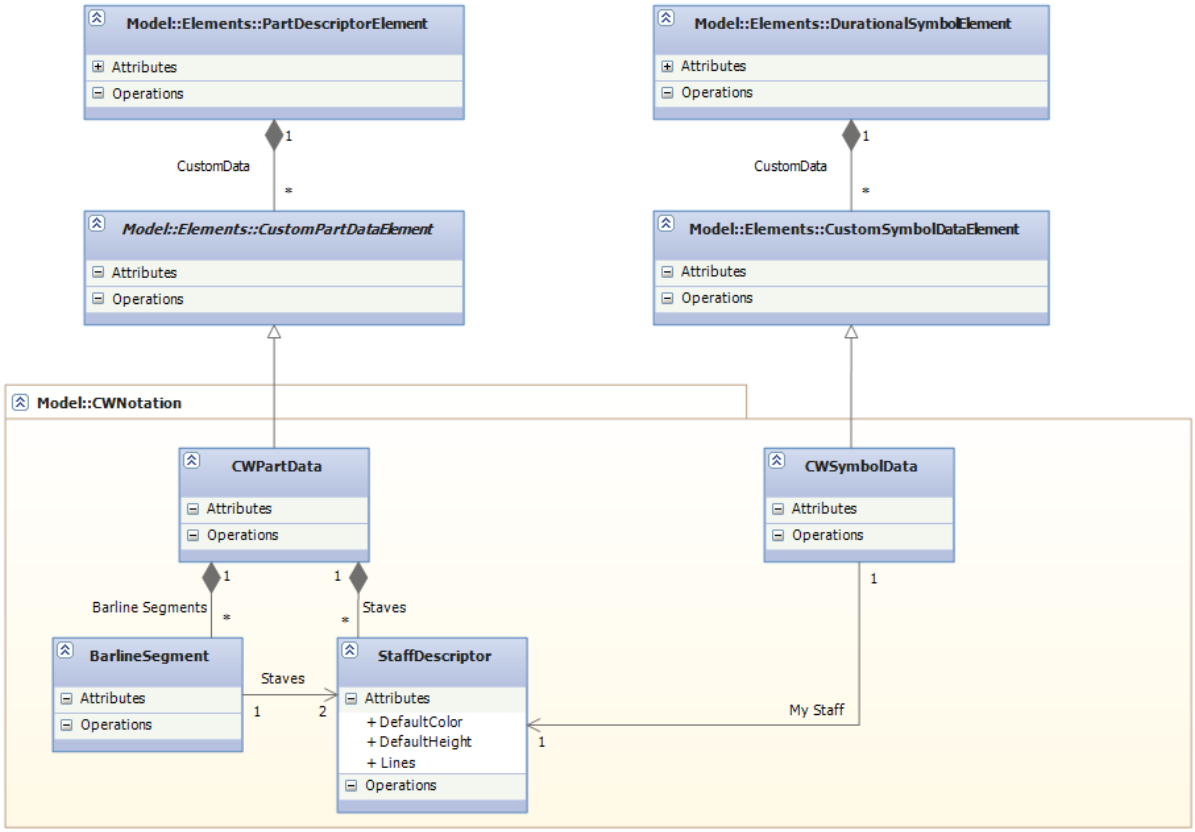
At the root of the model stands the *MusicSectionInterior* class that derives, as explained in the previous section, from the *SectionInteriorElement*. It is a sort of bridge between the notation model and the outer (score) environment. Its purpose from the score point of view has been already described. Viewed from the other (notation model) side, it is a container for systems (of staves) and it handles the outgoing communication that crosses the section boundary. The structure is illustrated in the next figure.



The first notation object in the hierarchy is the system. A system is a way to connect staves to indicate that they are played in parallel. It can own four different types of elements of which the most important are staves. The other symbol types are staff connectors that are used to visually connect related staves, barlines that separate individual measures and connectors which are direct counterparts of connector elements from the composition model. Note that most of the classes have position attributes which is an important difference to the composition representation.

Another noteworthy change is the axis along which the composition is divided. In contrast to the composition model that used a time-wise division (i.e. parts within measures), this notation representation was created using the part axis. However, since the time-wise organization is already captured in the composition data, there is no need to re-represent this information and the model uses simple linear collections of symbols. This fact allowed me to radically decrease the depth of the aggregation hierarchy.

All the remaining musical content is owned by staves. As you can read from the figure, these symbols are divided into two abstract classes: core symbols and attachments. Both of them will be described in details later in this text. The description of staves is stored in a custom data element inside the part descriptors in the composition model. The scheme is illustrated in the following diagram.



The collection of staves is created accordingly to the collection of staff descriptors of parts presented in the owner music section. The extended part descriptor contains one more object that defines the segments in which barlines are drawn on the staves of this part. The notation model uses also the durational symbol extension point to store the descriptor of the staff at which the durational symbol should be presented.

Barlines

As I already wrote, a barline is a visual separator of two measures. However, several types of barlines exist each having a different effect on interpretation. The simplest is a single barline that has only the basic separator function. Then we use a double barline that does not have a direct impact on the interpretation but typically indicates an important change (like different scale or tempo) between the two measures, Repetition barline appears in pairs, opening and closing line, and defines a simple repetition of the enclosed passage. The last type is the final barline that denotes the last measure of the composition.

Except for the difference between a single and a double barline, the type can be easily determined from the properties of the separated measures. The barline class holds references to the measure element on the left and on the right. One of them may be null when the barline is on the system border. The type is changed automatically when their properties change. If he needs to, the user can manually toggle between a single and a double barline.

Staff

A staff contains two linear collections: A collection of core symbols that is ordered by the position of the symbol from the leftmost to the rightmost one an unordered collection of

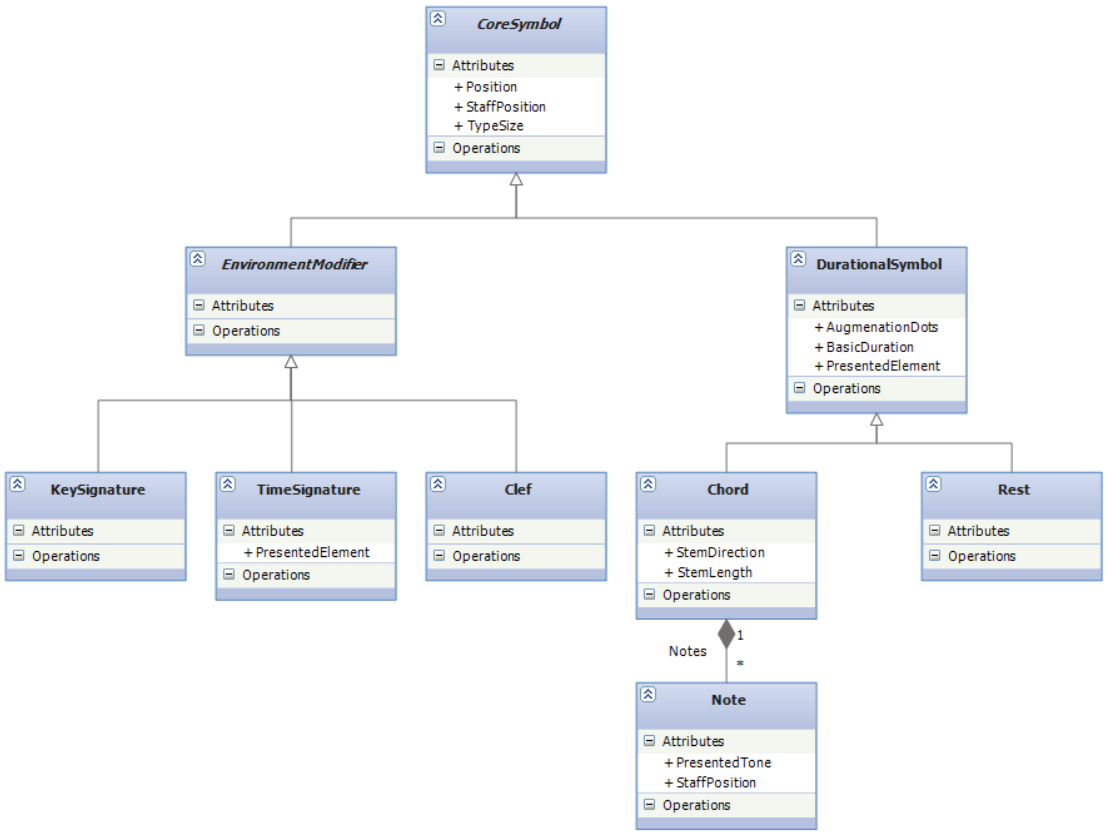
attachments. Each staff contains a reference to its descriptor that allows the application to easily relate two staves of the same part or two staff instances from different systems that logically represent the same staff. A staff automatically creates representations for all the symbols in the composition data that belong to the measures presented by the owner system and have custom symbol element pointing at the descriptor of this staff.

Core Symbols

Core symbols are symbols that have a dominant role in forming a line of music, both compositionally and in forming the graphic layout. Their hierarchy is illustrated below. As you can see, they divide into two groups: durational symbols that correspond to the durational symbols from the composition model and environment modifiers. The latter set or modify the staff environment. The environment consists of four parts, clef, key signature, time signature and temporal (measure-wide) accidentals, and defines the unique conversion between a tone in the composition data and a note in the musical section. Each symbol of this class modifies environment from the place where it was encountered to the next symbol of the same type.

The time and key signatures are created according to the composition data (meter and key indications). The clef does not have any counterpart in the composition model.

You may have noticed that, in contrast to the model of the composition, only two types of durational symbols are defined here. The reason for creating additional symbol types in the composition representation was that tremolos and tuplets have different duration than is the sum of the durations of the symbols they were made from. Since this information is already stored in the composition, I could have simplified the model at this level.

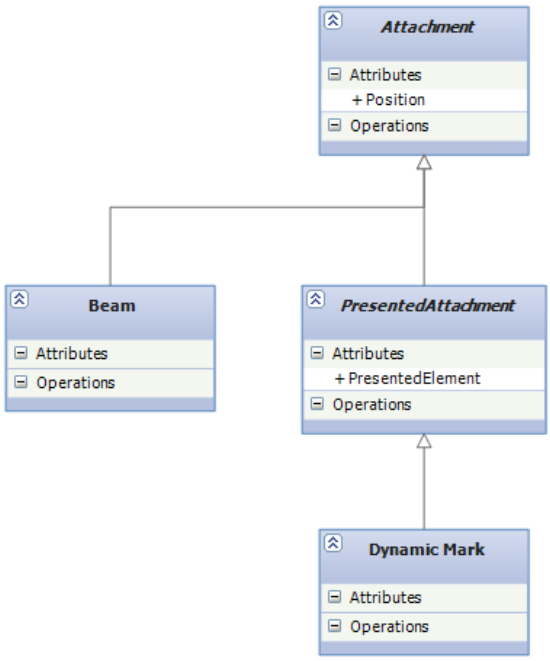


Another observation to be made is that the duration of the durational symbols has been divided into two parts: basic duration and the number of augmentation dots. The basic duration defines the shape of the symbol and the augmentation dots increase the total duration up to the value defined by the presented durational symbol element.

Each durational symbol references the durational symbol element from the composition data that it represents. Each note symbol then references the tone element to which it corresponds. Any change in the presented element is immediately reflected at this layer. Likewise any manipulation of a core symbol that affects the composition data (like changing the duration of a chord) is translated into operations on the composition model. As for the environment modifiers, they do not correspond to any composition element. Hence, they are example of information added by the notation system to the original composition data.

Attachments

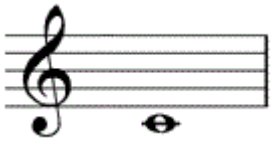
Like in the composition model, the attachment is a generalization for symbols that add information to and are, in some form, attached to core symbols. I will show here only the basic classes and two concrete attachment symbols.



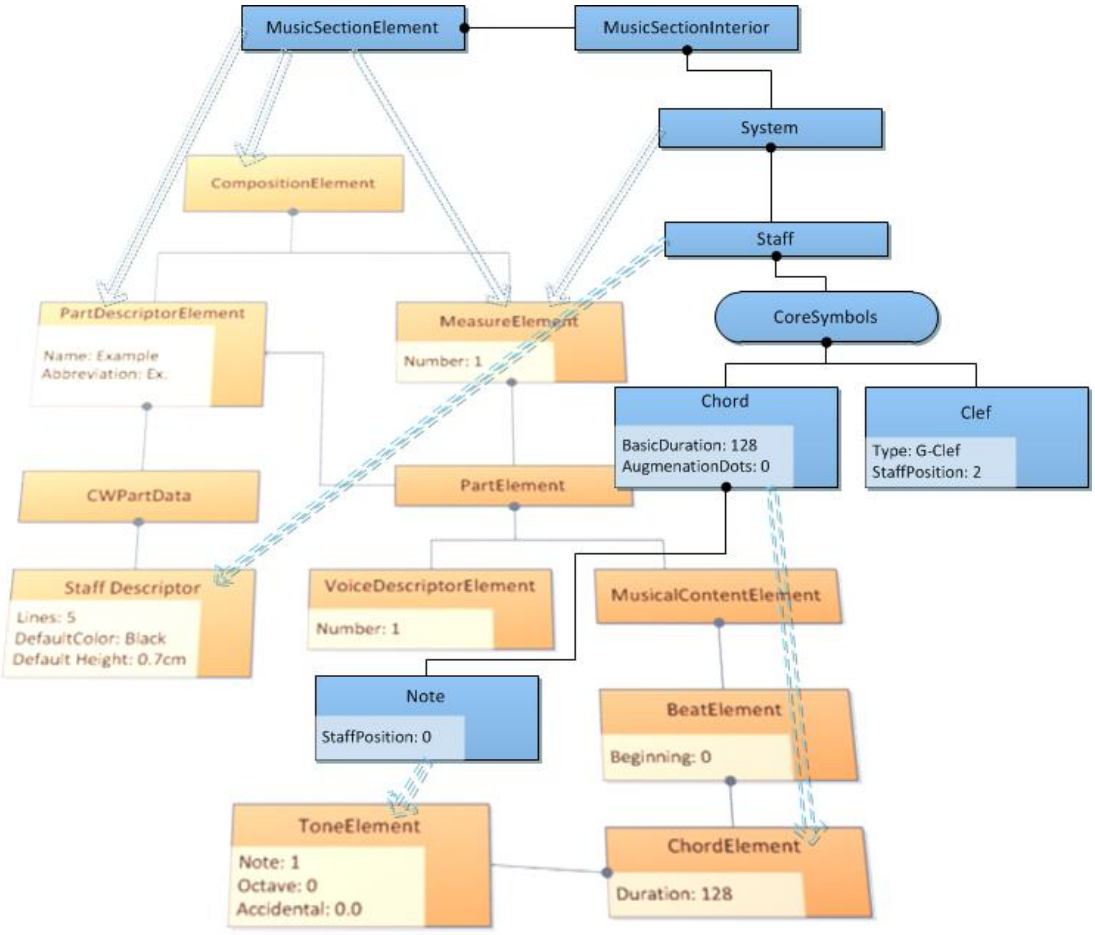
As you can see, there are two types of attachment classes: unqualified attachment that is the root of the inheritance hierarchy and *presented attachment* that is a base type for symbols that represent attachments from the composition model. The diagram contains examples of both. The dynamics is global and notation system independent information, hence it is modeled in the composition data. The dynamic mark is a corresponding symbol used in common western notation to express it. On the other side a beam does not add any musical information. It is used only to increase readability of the score by accentuating the rhythmic groups. Therefore, it has no representation in the composition model. Examples of other attachments are pedalization, tempo and agogics or lyrics.

III.4 Examples of Basic Score Representation

Before closing the part about composition and notation models I will show here two examples of how basic scores written in common western music notation would be represented at the composition and the music section levels. The 1st score is in the figure below. It has a single staff that contains a treble clef and a whole middle C note.



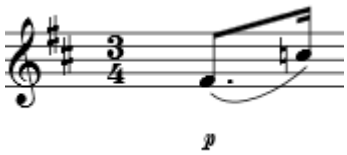
Below you can see the diagram capturing the representation of this score in the composition data and the music section. Both diagrams are superposed and references between them are marked with dashed arrows for direct representatives and dotted arrows for other reference types. The orange classes belong to the composition model and those from the music section level are blue.



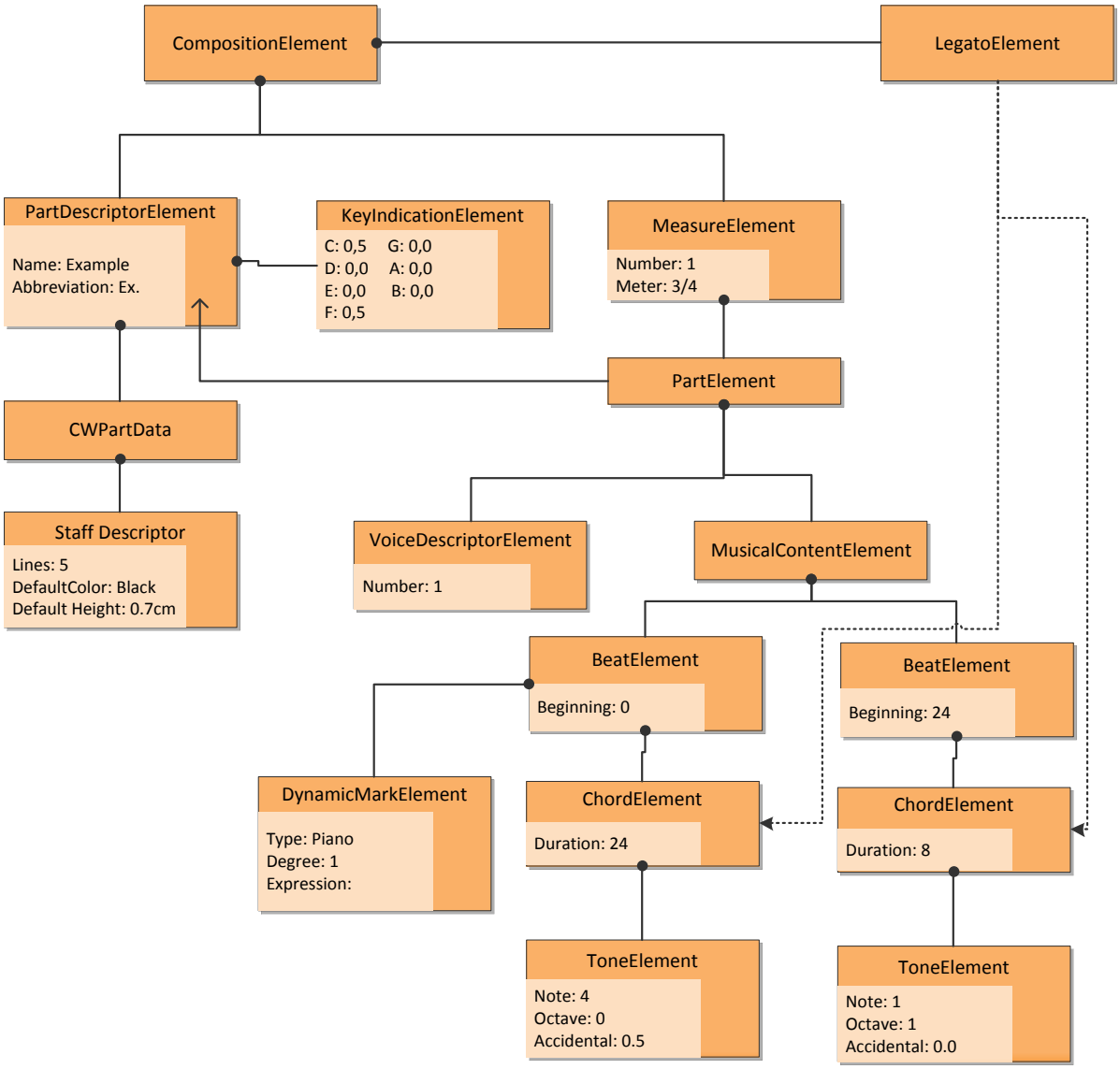
The structure may seem overcomplicated at a first glance, but actually there are just a few classes and also the number of direct references (other than direct ownerships) is very small. The music section references the composition that it visualizes, the descriptors of the presented parts and the first and last of the presented continuous measure range. The system references the measures that it presents. The staff holds a reference to its descriptor and

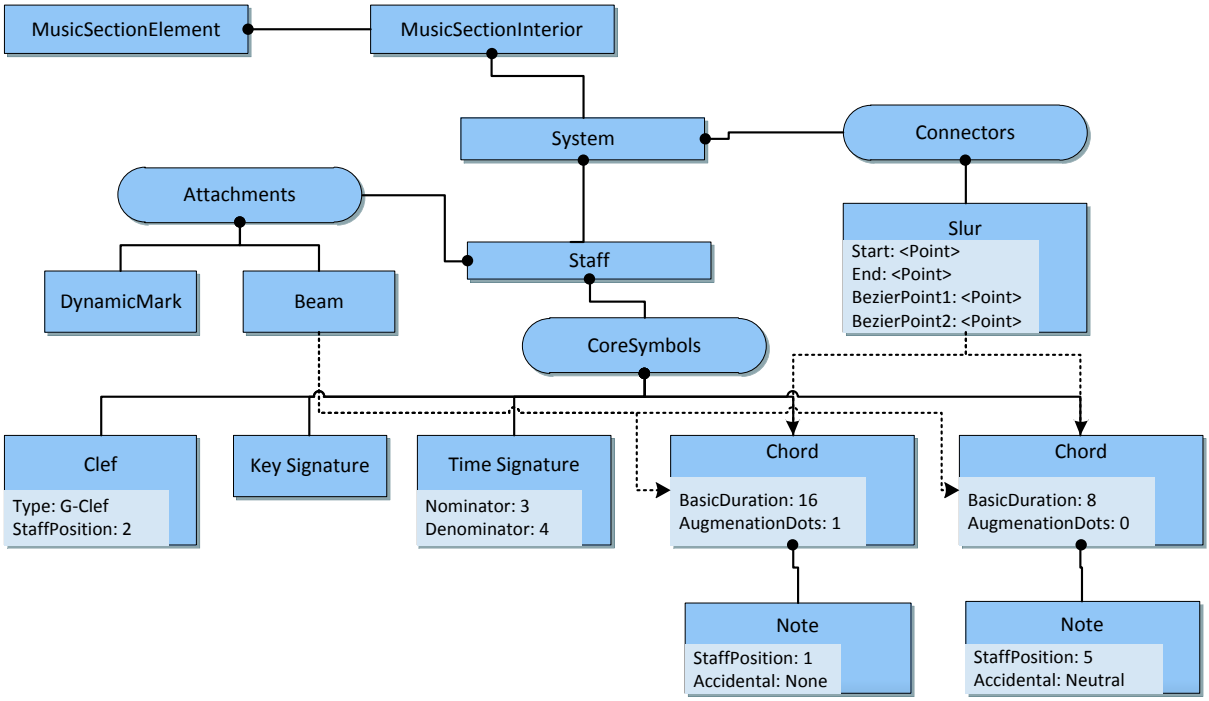
finally the chord has a pointer to the presented chord element and the note to the presented tone. As you can notice, the notation model does not repeat information represented at the composition level. It just adds the necessary data to visualize it.

The 2nd score is more complex. It consists of a treble clef, a D major key signature, ¾ time signature, two beamed notes connected by a slur (legato) and a dynamic mark (piano).



In this second example I will present the composition and music section levels separately, since otherwise the figure would be difficult to read.





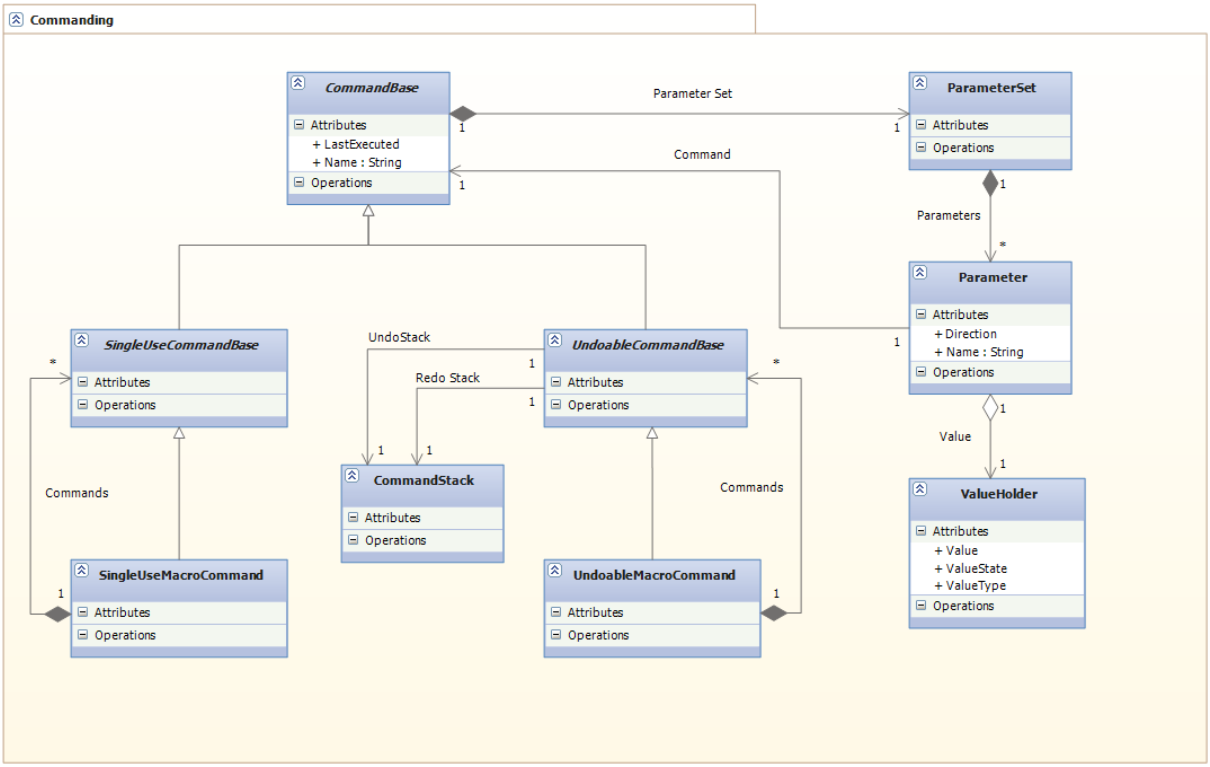
The basic structure is identical to the previous example. The first important difference is the Key Indication Element owned by the Part Descriptor. This element, as its name suggests, indicates the key of the part to the higher levels. It is a list of 7 items, each corresponding to one note of the heptatonic system. The value of each item ranges from -1.0 to 1.0 and corresponds to the Accidental field of the Tone Element. The Part Descriptor publishes a "Key Changed" event that the superior levels may subscribe in order to receive notifications about new key and its range of validity (first and last measures). Another new element is the Legato Element which represents the slur connecting the two notes (chord elements). The last new class is the Dynamic Mark Element that represents the piano dynamics in the score. Note that it is not associated with any note, but directly with the beat to which it belongs. The reason is that typically, the dynamics is the same for all voices. However, the Dynamic Mark element has also a "Voices" attribute (omitted here) that can be used to indicate that only specified voices are affected. The remaining difference is the Meter attribute of the Measure Element.

At the music section level you can observe basically the same new classes as in the composition data. I have omitted the position attributes except for the slur. Hence you can notice that it is drawn as a cubic Bézier curve and that the coordinates of all the control points are stored within this level. There is one new class that does not have any counterpart in the composition and it is the Beam. The beam is a purely graphical element that eases the orientation of the player, but does not bring any new musical information. Therefore, it only exists in the music section. Note also the way the duration and accidentals are encoded at this level. The duration of the first note (eighth note dotted) that was represented as a single number in the composition data has been split into two values: Basic Duration (8-th note) and Augmentation Dots (1 dot). The displayed accidentals are given by the current key.

III.5 Commanding Library

One part of my editor that I would also like to mention in this text is the commanding support library. It provides core functionality for creating undoable commands including one interesting feature. It is the ability to bind one or more parameters of one command to parameters of the same type on one or more different commands. In this way one can easily crate complex trees of command that are automatically executed as a single command.

This functionality allows the view-model to define a reduced set of basic commands that can be composed to create an advanced macro command without a necessity of defining a new command class. The structure of the library is captured by the figure below.



As you can see, it contains only a few classes. Two of them are frequently used in the view-model and the user interface: *Undoable Command Base* and *Command Stack*. The latter implements a simple stack that raises an event when its content has changed. This notification, that allows the user interface to react on changes of the stacks by enabling and disabling corresponding buttons etc., was actually the only reason to define a dedicated class instead of using the standard collection offered by the framework. The *Undoable Command Base* class is a base class from which all the undoable commands in the application should derive. It controls the execution of the command and its subcommands as well as the management of the associated command stacks. The execution of a command can only be invoked by calling the *Execute* method. The command itself decides (depending on its internal state) whether it will do, undo or redo its operation. This internal state can be one of the following three values:

- *Not Executed* – The command has not yet been executed. The execution will do the command operation.

- *Executed* – The command has already been executed. The execution will undo the command operation.
- *Undone* – The command has been executed and then undone. The execution will redo the command operation.

The transition between these states is obvious (*Not Executed* -> *Executed* <-> *Undone*). As I already mentioned, the class manages the command stack itself. It could have been left on the caller's responsibility, but this way is more convenient for two reasons. First, it simplifies the code on the calling side and eliminates the risk that the caller forgot to modify the command stacks. Second, it allows the caller to create and initialize the command at one place and execute it at another where, however, he may not have access to the correct command stacks. One example of this usage is the process of the creation of a new section (music, text or image) on the page:

1. The user activates a user-interface command by clicking on the button for creating a new section.
2. The user-interface determines the type of the section to be created and creates and initializes a corresponding new view-model command.
3. Now the user has to draw the area where the section will be inserted. He may also change his mind and decide to create another section or not to create any. For this reason, the command created in the previous step was not yet executed. It was just stored in the application state.
4. The user has drawn the area where we want to put the new section. The page control examines the application state and retrieves the command for creating the section. It does not know what section type is created but it knows that the command has a parameter called "CreatedSection" that will be filled by a reference to the resulting section element. It uses the parameter binding functionality and adds commands for setting the position and size of the new section (initialized by the area drawn by the user) to the original command, inserts these two into one macro command and executes it. It does not have to care of the actual command stacks that will be used.

If a command has subcommands (its parameters are bound to other commands), only the root command can be put on the command stack, the subcommands must not. For this purpose the undoable command base uses a three-stage command execution:

1. The command state is inspected and the mode of operation is decided (do, undo, redo). After a successful execution of the remaining two stages, this stage also manipulates the command stacks appropriately.
2. The subcommands are executed on this stage. Depending on the mode of operation they are executed either before the own operation of this command (do, redo) or after it (undo). Because the subcommands are executed by a direct call to their 2^{nd} stage method, they are not put on the command stack.
3. The own operation of this command. This stage is the only one that is implemented by derived classes. The remaining code is performed by the undoable command base class.

Parameter Binding

If you take a look at the diagram of the commanding library once again, you will notice that the command parameters are implemented as dedicated classes stored in a so-called parameter set. This class is responsible for searching the parameter instances by name and contains the code that realizes the parameter binding. Each parameter is identified by a name and has a direction that defines whether it is an input, output or an input-output parameter. In order for two (or more) parameters to share a common value during binding, the parameter has a reference to an (possibly shared) instance of a value holder. Besides the value itself, the holder stores the value type and state. The value can be either *Computed* or *Not Known*. When it is not known, the parameter has either not been set yet or it is bound to a parameter of another command that has not been executed. The concrete case can be identified by the state of the *Command* property of the *Parameter* object.

The parameter binding is a directed relationship. It always sets one parameter as a provider and the other as a consumer. Two parameters can be bound only if they have a compatible type (i.e. the value of the provider can be assigned to the value expected by the consumer). For future work I am thinking about adding a value converter into the binding mechanism that would further extend its possibilities.

III.6 Layout Algorithms

The standard music notation is a highly complex graphical system and as such it defines a large amount of complex rules for the layout of the score and its symbols. They constraint various levels from how a single symbol should be placed on a staff line through alignment of notes inside a chord up to the overall look & feel of the score. This section will bring a brief description of algorithms that I developed to implement these directives. I will not present all of them, just the most important ones. All the rules were taken from the Notography learning book by Ivo Zelingr (1). In order to better understand the algorithms, I will cite the corresponding rules before each one.

Note: In the following section, I will often indicate the sizes or spacing between symbols. In notography the basic size unit is one staff space (i.e. the distance between two consecutive lines of the note staff).

Chord Layout

Note (or Chord in my notation model) symbols are the most important part of the drawing. They fill-up a function of orientation in playing for the player. That is why they have to be drawn very expressively. The chord layout includes positioning of five symbol types: notes that form the chord, their accidentals, ledger lines, augmentation dots and finally the chord's stem. First I sum up the rules applying to this situation:

Stem

The rules for stem layout are the simplest ones. They control two parameters: length and orientation.

- The stem is three and a half spaces long. If there are more notes in the chord or the stem has flags (for durations equal to or shorter than the eighth note), we extend the stem by a suitable length.
- When the notes are drawn outside of the note staff (on ledger lines), the stem is extended to the third line of the staff.

The orientation of the stem is controlled by the position of the note that is furthest from the middle staff line. If it is above the stem will point down, otherwise it will be oriented upwards.

Alignment of Note Heads

The notes of the chord have to be drawn exactly underneath themselves. They have to make a vertical rank. An exception has to be made however, when the interval between two (or more) notes is just one second. In such a case there is only one half of a staff space between their heads and so they cannot be placed in a vertical rank because they would meld. In this situation we have to put them on opposite sides of the stem. We still have though two options to place them: Should we put the upper note on the right and lower on the left or the other way round? In order to avoid confusion there is the following rule for the notes placement:

We start to draw the chord from its base note that will define the whole appearance of the chord. The base note is the note that controls the orientation of the stem as explained above. We put it on the right side if the stem if it is pointing down or on the left side if it is oriented upwards. From the base note we continue to draw the chord putting the other notes in the line and every time a note does not get in, we place it on the opposite side.

Alignment of Accidentals

The accidental lowers or raises the value of the tone represented by the note. They make one common symbol with the note, so it is necessary to draw them clearly and locate them exactly towards their respective notes. This rule is especially important when there are more accidentals in a chord.

We draw the accidentals starting from the top and proceeding downwards trying to put them as close to their notes as possible. If an accidental does not get in the line (because it would meld with the above one) we have to put it further to the left. However, whenever it is possible, we try to exploit the form of colliding accidentals to put it as close as possible. An exception is made when there are notes in a one second interval that are placed on ledger lines. The accidental should never interfere with a ledger line (so it must be placed far enough) because such a layout does not look well and it decreases the readability of the score.

Ledger Lines

The ledger lines give us possibility to draw notes correctly outside the note staff. We paint them above or below the staff in regular distance like the standard staff lines. A ledger line overreaches note of one half space on both sides. It is important that the ledger lines in chord are of the same length and make a continuous rank. When two notes are placed on opposite stem sides, their ledger line and all the following ledger lines in the direction of the staff are drawn wider in order to make basis for all the notes.

Augmentation Dots

An augmentation dot is drawn about one half space past the note which it belongs to. When a note has more than one augmentation dot, they have a quarter of the space between them. The augmentation dot again makes a single symbol with its note, so it must be positioned in a way that this relationship is clear. The dot is always placed in the same staff space as its note. If the note is on a line the dot is drawn in nearest space above.

The augmentation dots must be drawn to each note in the chord. Their placement adheres to the rules above. In all situations the dots must be placed in a perfect vertical line. Nevertheless, we have to handle once again the case of notes in a second interval. When there are such notes, the complete dots line is moved one half of space past the note on the right side. Since the number of dots must correspond to the number of notes, it may happen that we must draw the dots in a wider range than the chord.

Chord Layout Algorithm

Now, when you have seen all the placement rules applying on the chord, I can describe the algorithm that implements them in my editor.

The algorithm for stem layout is a straightforward transcription of the rules. I have only fixed the length that is added to the stem when it has flags or when the chord has more notes. The same for positioning of note heads. However the algorithms for positioning of accidentals and augmentation dots are more interesting.

Layout Accidentals:
Input: Collection of notes ordered from top to bottom
Variables:

- *actualNote, noteAbove, noteBelow* – The note whose accidental is currently being positioned, the nearest note above and the nearest note below.
- *lastAccidental, lastAccidental2* – The last two positioned accidentals

Steps:

1. The accidental is placed one third of space to the left of its note. If it would collide with the note above or below, it is placed one third of space to the left of the colliding note. If the reference note is on ledger a ledger, the accidental is moved to the left far enough in order not to interfere with the line.
2. A collision test with the last accidental is performed. If the accidentals collide we compute the minimum necessary offset of the positioned accidental (depends on the shape of both accidentals), move it to the left and proceed with step 3. Otherwise we skip to step 4.
3. A collision test with the *lastAccidental2* is performed. If it is positive, we compute once again the minimum necessary offset of the positioned accidental and move it to the left. Skip to step 6.

4. A collision test with the *lastAccidental2* is performed. If there is a collision the positioned accidental is moved to the left (by the minimum necessary offset). Otherwise we skip to step 6.
5. A collision test with the last accidental is performed once again. Now, if it is positive, we compare the offset computed for these two accidentals with the offset from the previous step. If this offset (from step 5) is smaller or equal we move the current accidental as usual. But, if it is greater we add it to the position of the last accidental. Thus we will ensure the desired triangular shape of the accidentals group.
6. The variables are updated and the algorithm restarts for the next note.

As you can see the algorithm body is not too complex. The most interesting parts are the 5th step that can eventually reposition the last accidental and the computation of the offset based on the accidentals shape. However, I think that it is worth saying that most of the current notation editors including the commonly used ones like Encore, Sibelius or NoteWorthy Composer do not implement it right. They do not keep the requested pyramidal form of the accidentals group nor do they exploit the shape of the accidentals. Although such strict rules may seem superfluous at a first glance, the wrong positioning of symbols considerably decreases the readability of the score. The same situation is with the augmentation dots in chords. When the dots must be drawn in a wider range than the chord, usually they extend the dots only downwards and often their number does not correspond to the number of the notes.

Layout Dots:
Input: Collection of notes ordered from bottom to top
Variables: Temporary array of dots positions, position of the last processed dot
Steps:

1. The dot being currently processed is positioned on the same staff position as its note.
2. If this position is a line, the dot is moved to the space above this line.
3. If the dot collides with the last dot, it is moved one space above.
4. Repeat the previous three steps until all dots are positioned.
5. While the distance of the topmost dot from its note is greater than the distance of the bottommost dot from its note, decrease the staff position of all dots by 2 (one space below).

The algorithm that positions the ledger lines is once again a straightforward transcription of the rules. However, it is maybe worth saying that is implemented at the view-model layer instead of the model one. The reason is that the model does not need a direct representation of ledger lines. They do not add any musical information; they just serve to increase readability of the score. The only place where they need to be considered in the model are the preceding layout algorithms and, since the rules for their placement are known, there is no need to create a persistent class for them. Hence, the ledger line is an example of a view-model class that does not have a counterpart in the model.

As you could see, none of the five algorithms is extremely difficult. Therefore, I find really disappointing that the authors of the existing editors did not implement all the notographic rules they should.

Horizontal Symbols Layout

Probably the most difficult algorithm in my thesis is the one used for horizontal layout of core staff symbols. The number of different situations and rules that applies to them is very large and so the code that implements them is quite complex. The symbols must first span the whole width of the staff; therefore their spacing must be adaptive. Second, the distance between two consecutive durational symbols depends on the duration of the first one. Again, this distance is not fixed there is only a prescribed ratio of the distances of symbols with different durations. Third, the symbols on the same staff and beat must be positioned in such a way that all are easily readable. And finally the symbols on corresponding positions on all staves must be aligned.

I will not cover here all the layout rules, since the text would get too long. I will briefly present the most important ones. In case of interest you can find their complete list in (1) or some of them in (2).

Time Spacing

The layout of the symbols on a staff depends on their duration. A whole note will have more space than a half note, a half note will have more space than a quarter note etc. Theoretically the ratio of these distances should be 2, because the duration of tone is also twice as long. However, if we used this rule for drawing the notes, the symbols would be too distant one from the other and the staff would be half empty. That is why we increase this distance by a ratio of just approximately 4/3. In practice we choose one duration as a reference (usually the shortest one) and define the space that will be used for it (I will call it a base size in the remaining text). Then we pass through the staff from the beginning to the end and position the symbols appropriately. In notography the base distance is typically a matter of guess. When there is a remaining empty space at the end of the staff, the base size is increased a bit. If the symbols go past the end of the staff, this size is decreased and the whole procedure restarts. Sometimes it may happen that the computed space is not large enough to contain the whole symbol (e.g. when it is a chord with many accidentals). When this happens, we add the necessary space. In practice the space we designed for individual durations has to be kept mainly optically. It means that we should consider the actual shape of the symbol, since some of them optically open or close the space. Anyway, this is the matter of graphics, not music and the editor does not implement it for now.

When we layout a system with multiple staves, we layout the whole system as it was a single line. Hence, we must work with all the notes and all the staves at the same time and choose the shortest durations for each passage. These symbols are then positioned using the above rules about time spacing and the remaining notes on the same beats are aligned vertically.

Vertical Alignment

The symbols on the same beat must be aligned vertically. However, since there may be symbols of multiple voices or even of different type on one staff, the rules are more complex. First rule is the order in which symbols of different types are positioned. First go the environment modifiers: The clef before, the key signature after. Of course these symbols may not be present. Then we position the durational symbols in the order given by the highest staff

position. When two chords are in collision, their mutual position is given by the combination of three parameters:

1. The distance between their notes (3 values are considered: 0 spaces, 1 half space and1 space or more).
2. The mutual stem direction (again 3 values: same, opposite, crossed – when the upper chord has its stem down and the lower up).
3. The durations of both chords (2 values: equal, different).

The combination of these three parameters gives us 18 different situations to solve. One more case exists and that is when at least one of the colliding chords has more than one note. When this happens, the second chord is placed on the right side of the first one.

I will not describe additional rules. They handle cases of collisions between auxiliary symbols like accidentals, dots or ledger lines in polyphonic compositions, irregular rhythms etc. Nevertheless, I think that the cited directives illustrate clearly that the symbols layout is a complex issue. I will now explain the principal ideas of the algorithm that is used in my editor.

Horizontal Layout Algorithm

The algorithm proceeds in three phases. In the first phase it determines the shortest duration in each measure for the calculation of the base size. In the second phase it passes through all the core symbols in the processed system and pre-layouts each beat locally. In the same time it collects the total occupied space and necessary durational symbols spacing. Finally the third phase consists in definitive layout of the beats and time spacing. The following text will describe each phase in more details.

Phase I

As it was said, during this phase the algorithm examines each measure and finds the shortest duration it contains. Using this value it then fills a hash table mapping duration to its corresponding unitary space size. The shortest duration is mapped to 1 unit of space and the distances for other durations are computed using the ratio described in the previous section. The hash tables for each measure are stored in a dictionary and returned as a result of this phase.

It is worth mentioning that this procedure does not have to iterate through all the symbols of the system. The information about the shortest in each measure and part is collected automatically when the collection of durational symbols changes and is stored in the composition data. Hence, the Phase I is an O(m * p) operation where m is the number of measures and p the number of the parts presented on the processed system.

The remaining two phases use a term of beat cluster in their description. A beat cluster is a collection of all the core symbols of a staff that belong to the same beat. In the editor it is implemented as a stand-alone class that offers vertical layout and iterator functionality to the layout algorithm.

Phase II

The purpose of the second phase is to pre-layout beat clusters and to collect data for computing the base size. Two values are computed for each measure: the total occupied space, i.e. the maximum of sums of widths of the beat clusters, and the total unitary time spacing. i.e. the maximum of sums of distances between beat clusters.

The second phase begins by creating a beat cluster placed on the first beat for each staff. The algorithm than repeats the following steps until all the clusters have passed the end of their staves (i.e. all the core symbols have been processed):

1. From all the active beat clusters, the algorithm selects a working set (i.e. the clusters that point at the minimal beat).
2. The vertical layout of beat clusters in the working set is performed.
3. The width of the cluster is added to the total occupied space for the current measure and the unitary space for the shortest symbol in the cluster is added to the total unitary time spacing.
4. The clusters are moved to next beat. If they passed the end of the staff, they are deleted.

The complexity of this phase is equal to the number of core symbols in the system. The algorithm core needs a time O(p) on each beat and O(b) to traverse all the beats in the system where p is the number of presented parts and b the number of beats. The layout of each beat cluster is linear in the number of symbols.

Phase III

The last phase finalizes the layout of the symbols using the data collected by the previous phase. The first step is to compute a base size for each measure. The following formula is used:

$$BaseSize_m = \frac{TimeSpaceUnits_m * (AvailableSpace - \sum_{measure} OccupiedSpace)}{(\sum_{measure} TimeSpaceUnits)^2}$$

A transformation function that multiplies the unitary space by the computed base size is than invoked on each hash table from the first phase. Finally this phase also creates the beat clusters for each staff. However their definition is extended by a position attribute that holds the position of the leftmost symbol in the cluster relative to the beginning of the staff. Then the algorithm repeats the following steps until all the beat clusters have passed the end of their staves:

1. The algorithm selects a working set of clusters like in the previous phase.
2. It determines the position at which the clusters will be aligned by computing the maximum of their position attribute.
3. It also computes the maximum width of the accidentals group and the maximum of the offset of the first durational symbol (non-zero in case the cluster contains environment modifiers).

4. The position of the symbols in each cluster in the working set is updated by adding the current position of the cluster and considering the values computed in the preceding step.
5. The clusters are moved to the next beat and their position is updated by adding the maximum time space.
6. If in the previous step the clusters have passed to a next measure, the corresponding barline is placed at the position computed in the previous step and the position of clusters is increased by one staff space. If a cluster has past the end of the staff, it is deleted.

The complexity of this phase is the same as in the case of phase II + O(m) for the computation of base size in each measure. The overall time complexity is therefore

$$O(m * p) + 2 * O(cs) + O(m) \sim O(cs)$$ where cs stands for the number of core symbols

Although the algorithm has to iterate twice through all the core symbols in the system, its performance is very good. I have not made any measurements of the consumed time, but from the user's point of view, it operates with no delay.

The editor actually uses two versions of this algorithm. The described one is used for layout of the music sections where the user decides the available space. The second variant skips the second phase (thus omitting the adaptive spacing) and uses a fixed base size value.

Beat Cluster Layout

I will just briefly describe how the vertical layout of a beat cluster is implemented. The first positioned symbol is the clef followed by the key signature (if there are any). There are special rules implemented for placing the first clef and key signature of the staff. The most complex part is the positioning of chords. The cluster contains the implementation of all the 19 situations described earlier in this chapter. It also performs additional modifications like prolongation of a chord's stem so that it matches the end of another stem when appropriate. The nice part is that the accidentals are placed in a similar way as in the case of a single chord. Hence the editor can use a single universal method for both.

III.7 Summary & Discussions

This chapter described the most important structures in both data models: the composition and the music section. The division of information into two distinct layers allowed me, first to present one musical data in possibly many different notation systems and second to reduce the complexity of both models. The model of the common western music notation is weakly based on the Kai Lassfolk's work (3). However, there are important differences in the representation of symbols and in the realization of their mutual references. For example he uses a large number of direct references, e.g. a chord is implemented as a linear linked list where each note has pointers to the note below and above. My opinion is that direct references introduce superfluous issues with maintainability of the code and whenever an object has to be moved to a different context or removed completely. Hence, their number should be limited and, where it is possible, they should be replaced by composition (exclusive ownership). In order to achieve this, I have sometimes introduced objects that do not have

direct counterparts in the music notation or I have extended their definition. One example is the Chord class that owns from one to many notes. It has removed the necessity of direct pointers between notes, but in music the definition of a chord is a set of at least three tones. On the other hand, one of the basic properties of the Lassfolk's model is that it includes classes only for objects that have a visual appearance.

I think that, even though both models (mine and Lassfolk's) are intended for musical notation software, each was designed with different goals in mind. In my opinion, his work suffers a bit from mixing different levels of abstraction. His model is heavily graphics oriented. He has separate classes for a staff line, augmentation dot, note head or stem. But at the same time he needs to capture the logical music information and so he has to introduce many attributes and direct references. The result is a bit too complex and not suitable for a direct implementation. However, it brings a very careful and detailed analysis of the common western music notation and it really helped me to better understand the roles of the distinct musical symbols and their relationships (although I am a relatively experienced musician).

Nevertheless, I already found a design issue in my model too, although, it is due more to its implementation than the structure itself. The problem is that I wanted to give my model library a strong protection against inconsistencies and prevent the higher layers to introduce any invalid data inside. This tendency has resulted into the interface with read-only collections that update automatically when change to musical data occurs and where the creation of objects is only allowed through the usage of dedicated methods. Whilst the model layer works well and its usage is very comfortable, the problem comes with undoable commands. I will show one scenario that illustrates the most serious issue:

1. The user invokes an undoable command that inserts a new note.
2. The command inserts a new tone to the composition data which results in an automatic creation of a note object in all the music sections that present this composition and a view-model adapter for the note in all the view-model adapters of these music sections.
3. The command retrieves a reference to the created view-model note in the active page and returns.
4. The user undoes this command.
5. The command removes the previously created tone which results in an automatic removal of the corresponding model and view-model notes.
6. The user redoes the command.
7. When now the command reinserts the tone in the composition, new note objects are created in the model and view-model. And the command holds an invalid reference.

The problem gets even worse when the user has executed commands that work with the created note after the execution of the above one and then undoes and redoes them all. This problem is not unsolvable, but it complicates the implementation of commands that have to post process the view-model collections and objects when they are redone. I am currently searching the best way out of this issue. One of the possibilities would be to move the logic to the view-model level and transform the model into a sort of high level storage model.

However, I do not like the idea of leaving the model vulnerable to inconsistencies too much. So this problem is still a matter of research.

There are also several disputable design decisions that I have made, so I will list them here and clarify my point of view.

1. It is true that, besides its advantages, storing the whole project in a single physical file brings also two important disadvantages:
    a. Even if the user has changed a single note, the whole package needs to be resaved.
    b. If the user stores large binary files inside the project (like big video files), it may complicate him the backup or versioning.

    However, I believe that, since the computers are today able to process data at very high speeds, the advantages prevail. From the application's point of view, it is much easier to manipulate a single file and to keep it consistent than to perform the similar tasks on a complete hierarchy on the disk. It is also less probable that the user mistakenly deletes some important part of the project, because, in order to do it, he would be obliged to extract and recreate the archive using ZIP compatible software. From the user's point of view, a single file is easier to distribute or transport to another computer. Besides this, the OPC format is commonly used in Microsoft products and more and more often in products of different companies, even though both cited problems apply to these packages as well.
2. One can also argue about the selected representation of durations (or generally time) as the powers of two. One objection against my model was that it seems strange to represent tuplets in a significantly different way than eighth notes.

    In my opinion, I represent the tuplets as close to their original notes as possible. A triola of eight notes is in my model represented as a single element with duration of one quarter note (32 128-th beats) that contains three eight note elements inside. It is easy for the layouter as well as for the music section model to recognize that it is a tuplet and needs to be treated differently, but that it is composed from eighth notes. When I designed my model, I was thinking about several representations of this musical element and from them two other would be usable:
    a. Represent a tuplet as separate durational symbols (in the above case as three individual eighth notes), but indicate them, by means of mutual references or a shared object, that they are actually shorter and form a part of a more complex symbol.
    b. Represent a tuplet as separate durational symbols with the real duration (i.e., in the above case, with a duration in between the eighth and sixteenth note).

    In both cases, it would be far more difficult for the superior layers to identify the tuplet and process it. Moreover, it would require putting the symbols on non-integral beats. Because of these drawbacks I think that the chosen implementation is the most practical one. Furthermore, the time in music really naturally divides into powers of two and the tuplets form the only irregularity.

3. Another disputable part of my model is the representation of pitch. During the design process I was originally studying different tuning systems with a goal to find their generalization and use it as the pitch data structure. However, after some time I realized that there is no need to search how to encode the real tone's frequency or to care about the actual tuning system used by the instrument. The important thing for the composition model is to be able to encode the "name" of the tone. Even though in some tuning systems the note "E" sounds differently when it is played in C major or in D major, it is still called "E". The actual frequency is the matter of the interpreter and the instrument he uses. Hence, the pitch design goal changed to partition an octave in a way of being able to represent any tone in its range and in the same time keep the data structure simple. Therefore I have finally chosen the classical heptatonic scale with addition of the accidental value that can lower or raise the basic value of the tone by up to 1 whole tone. The resulting representation does not have a direct counterpart in physics, but it is simple to understand and to process.

Besides the described points I did not find any important issue and the model layer is very flexible. Implement a new notation symbol is easy and, because of the implemented storage model, it becomes immediately serializable with just a small effort. Moreover, since the properties of model elements are accessible by name, most of the commands at the view-model layer are implemented by the same class, so typically only the insertion and removal commands need to be created.

Chapter IV
User Interface

This chapter brings the description of the concept of the user interface of my editor. It is implemented in the Windows Presentation Foundation environment which allowed me to relatively easily create a modern and easy-to-use graphical user interface. This technology offers powerful features like declarative UI elements creation and their separation from run-time logic, data binding or templates.

IV.1 Basic Concept

Several years ago, Microsoft has come up with a new user interface that it called Ribbon or Fluent User Interface (4). It replaces the menus and toolbars that were used during the last twenty years. Microsoft invested large sums into the research and testing of the usability of their new interface and I share the opinion that it increases the discoverability of features and functions and that it enables quicker learning of the program as a whole. For these reasons I decided to use this interface for my editor.

I also wanted to give the user freedom in customizing the layout of the application to better suit it needs. Hence the editor offers a similar docking layout system as for example Microsoft Visual Studio. Actually the design of the user interface has been inspired by new Microsoft products, especially Microsoft Office and Microsoft Visual Studio.

The application area is composed from three parts:

- The Ribbon that contains commands
- The Project Explorer Window that visualizes the tree of the open project
- The Tabbed Document Pane where the user edits the open documents (pages, linear composition views, metadata sheets).

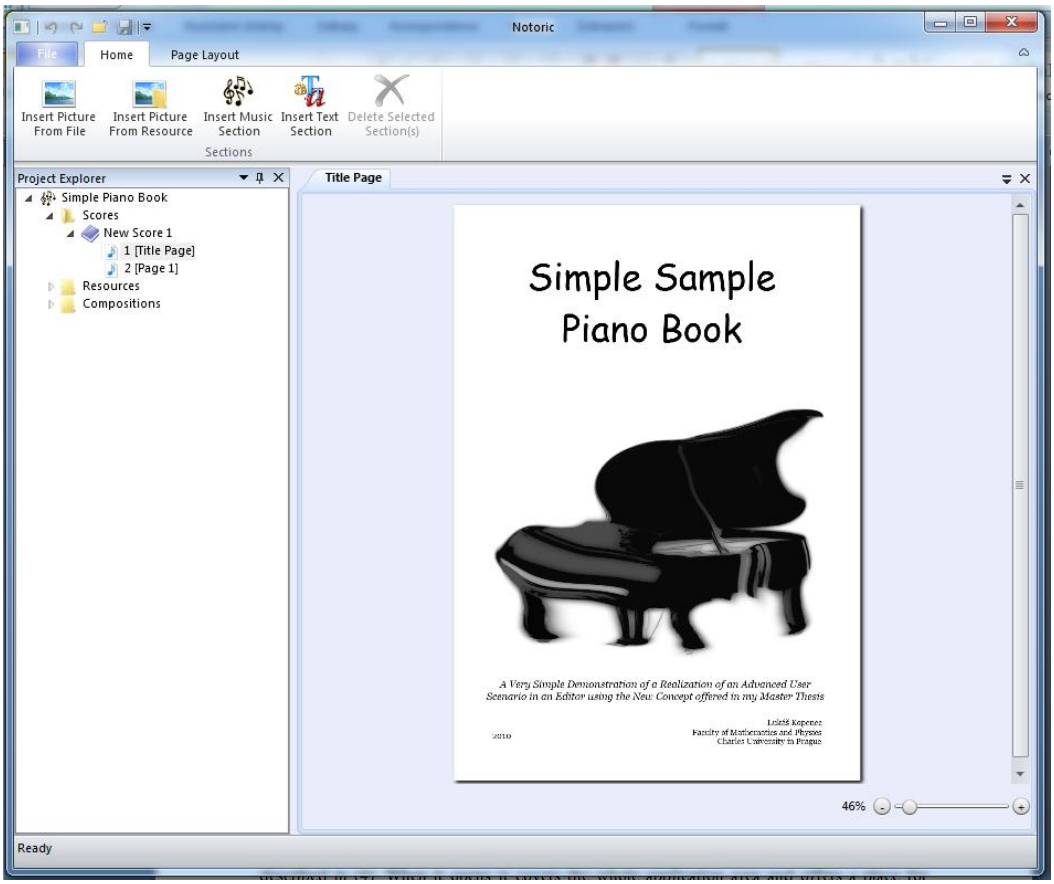
There is one remaining component that is not visible until the user activates it and it is so-called “backstage”. The backstage is a Microsoft 2010 variant of the application menu described in (4). When it opens it covers the whole application area and offers a place for application commands and other information that does not fit onto ribbon. An example is a list of recently opened files or print preview.

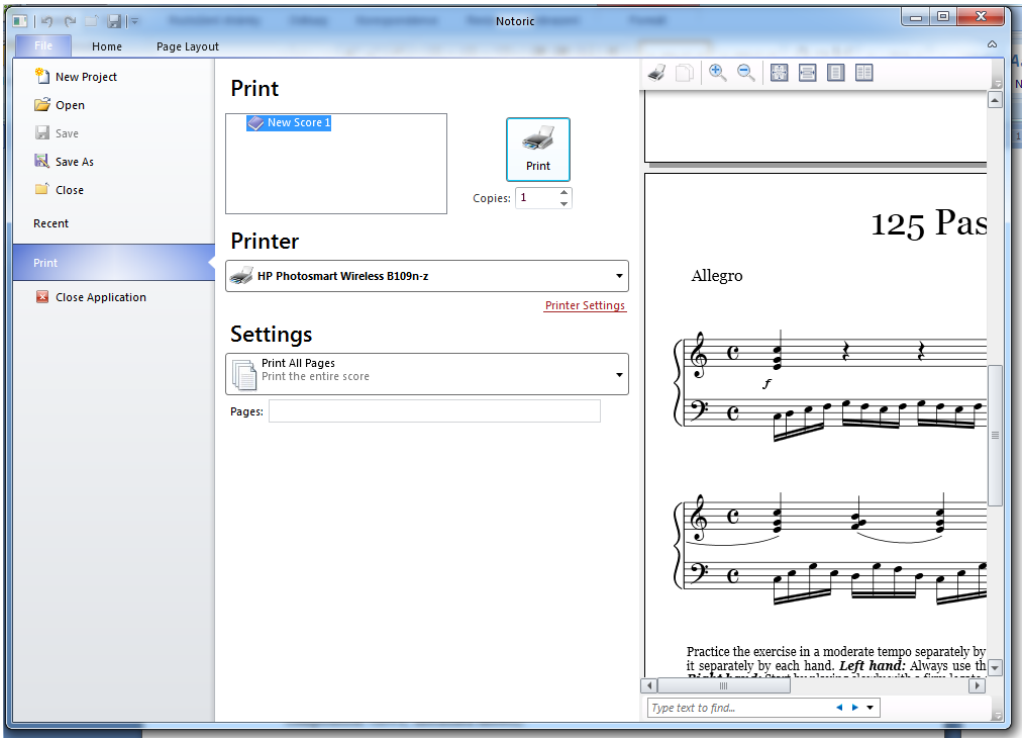
Following the Microsoft recommendations, the user should be able to perform most of the tasks without opening any dialog boxes. Besides the commands in ribbon he can also use the context menus. However, again following the Microsoft recommendations, there should not be more than one way to complete a specified task.

Given by the nature of the editor’s concept, there are two levels of mouse selection: The user can either select the whole sections or he can select part of the content of a section. The rule is defined for this situation that more than one section can be selected at a time, but at any

moment at most one can have active content selection. So, whenever the user makes a selection inside a section, it cancels the previous content selection.

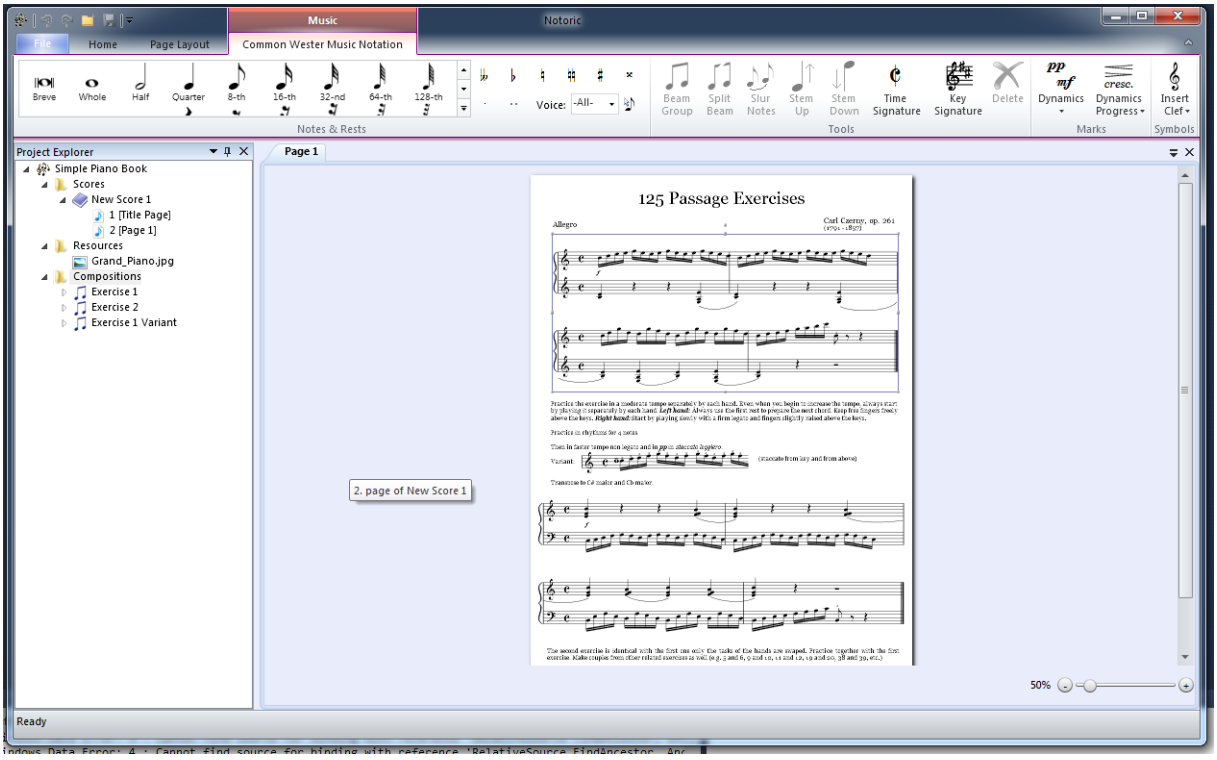
Below are two screenshots of the running application. The first illustrates the editor in its normal state. The second shows the open backstage.





IV.2 Ribbon

The ribbon holds most of the commands available in the application (the remaining are in the backstage and context menus). For my editor I use an open-source implementation of the Microsoft Fluent UI available at http://fluent.codeplex.com/. As explained in (4), the commands are grouped by their function and organized into group boxes and tabs. The ribbon also defines a concept of contextual tabs. These tabs are hidden until the user activates their context (usually by selecting an object of a concrete type) and they hide again when their context is deactivated. I use these tabs for the commands that manipulate content of individual sections as illustrated on a screenshot below:



Besides the ribbon, the window contains also a quick access toolbar (in the top-left corner) where the user can place the commands he uses most often. If he needs more space for his documents, he may minimize the ribbon by clicking the small arrow in the top-right corner. Another feature of the ribbon is a dialog box launcher. It is a small button in the bottom-right corner of a group box that opens a dialog box with infrequently used commands and settings. I use it for example for the page appearance group box.

The "File" button in the top-left corner opens the backstage. As you could notice in the figure, it contains the application commands (open, save, etc.) and two larger tabs. One for printing where the user can select the printed score and pages as well as basic printer settings and where he can look at the print preview. The other contains the list of recently opened projects.

IV.3 Project Explorer Window

The Project Explorer Window visualizes the tree of the currently opened project. It is a dockable window, the user can drag it with the mouse and dock elsewhere in the application area or he may hide it if he needs more space. Besides visualizing the project tree it also

provides context menus with commands related to adding, deleting, opening or renaming individual project parts.

IV.4 Document Pane

The document pane is the place where the user will spend most of his working time. It contains all the currently opened project parts. As in the case of the Project Explorer Window, the document pane can be dragged by the mouse and docked somewhere else if the user prefers a different application organization. To make the orientation in the open documents easier, it displays a descriptive tool tip when the user move the mouse cursor over a tab. For example a tab that contains a page will show a tool tip in the format "{Page Number}. page of {Score Name}". The user can also change the order of the tabs by dragging them with the mouse if he wants to.

IV.5 Undo / Redo / Save – Consistency Constraints

Although the user can open individual project parts, he cannot save them individually. Each time he invokes the Save command, the whole project is saved. The reason is that many changes in one document may have impact on other project parts. The best example is insertion of a note into a music section on a page. This action affects not only the open page but also the presented composition (which is a different project part) and also any pages that present the same composition passage. If the user could save individual documents, it might easily happen that he would make such a global change in one page, save it, but then he would choose to close the project without saving changes. The result would be an inconsistent project package that would fail to open.

Similar problem exists with the undo and redo commands. Each document has its own command stacks and there is one more pair of stacks on the project for commands that go beyond the scope of any document (like adding a new score). Local command stacks allow the application to work as the user expects: When he makes a change into one page and then switch to another he assumes that if he now presses the Ctrl + Z, he undoes an action that he had previously made in this document, not to the page he has just left. However, once again there is this problem with actions that have global impact. Imagine that he inserts a note in one page. Then he opens another that presents the same composition passage and makes some changes that affect the note he has created. Then he switches back to the previous page and undoes the insertion command. What should now happen to the command stack of the other page?

In order to avoid this problem, the application defines a time-based constraint on the command stacks of open documents: The undo / redo operation on the command stack of the active document can only proceed if no open document contains a newer command on the corresponding stack. Otherwise the undo / redo button is disabled. For this purpose each command has a time stamp that contains the time of its last execution. The application also clears the command stacks of the documents that are closed.

IV.6 Creation of a Score

As the last I will provide a brief scenario of how the user creates a new score from scratch.

1. He creates a new project using the “New” command from the backstage
2. He adds a composition to his new project.
3. He opens the linear view of his new composition. This activates the “Music” contextual ribbon group and he can start editing the composition data.
    a. For insertion of a new note or rest he uses the in-ribbon gallery and toggle buttons on its right side for adding an accidental or augmentation dots.
    b. He may also select the voice to which he wants to insert the symbols. If he does not, the symbols are inserted to the default voice for the staff.
    c. He may change the clef type or key signature by using their context menus.
    d. He may select a rectangular region of core symbols (chords, rests, clefs and key signatures) and manipulate this selection using the ribbon commands or context menus. He may also copy or cut it to the clipboard and paste it somewhere else.
4. Once he has finished editing the composition data he has two options for creating the final score:
    a. He may generate a default score from the composition.
        i. He invokes this operation by opening the composition’s context menu.
        ii. He specifies the parameters for the generator. In this version he may only specify the number of measures per system and of systems per page. In the future versions, however, he will be able to select the notation system that should be used and other parameters like score theme etc.
        iii. The score gets generated and he may now post process it if he wants to.
    b. He may create a blank score and draw the music sections by himself.
        i. He creates a blank score using the context menu in the project tree.
        ii. He adds pages to the score using the context menu of the score.
        iii. He creates new sections using the ribbon commands at the “Sections” group of the “Home” tab and then drawing the area where he wants his new section to be placed.
5. Once he has finished editing the score he can print it.
    a. He opens the backstage using the File button.
    b. He selects the print tab.
    c. He may optionally change the page range that he wants to print or modify the printer settings.
    d. He may check the result in the print preview pane on the right side.
    e. He presses the “Print” button.

IV.7 Discussion

The usage of Windows Presentation Foundation for the user interface allowed me to create a nice and powerful user interface with much less effort than I would expect or would be able to get with any other available technology. However, it has one major drawback. The Windows Presentation Foundation is not (and is not planned to be) implemented on Mono, which is an open-source implementation of the .NET framework that is used on Linux and Mac OS. Consequently, the application as is cannot be run on different operating system than

Windows. Nevertheless, the model and commanding libraries are portable and if I make a few modifications to the view-model it will be portable too. Therefore, the only application parts that need to be written separately for Mono are the view and user interface. The implementation of the view in the older Windows Forms environment will not be too comfortable, but with a good planning and organization it should not be extremely difficult. The implementation of the user interface should be easier, although some of the described functionality may not be available. Anyway, I think that the benefits of WPF usage are such that it was worth it.

Chapter V
Conclusion

The individual aspects of the program and its concept were described and discussed in their respective chapters. Therefore, in this last chapter I will focus on the planned pursuit of my thesis, to its contributions and I will briefly present other works that relate to this field.

The purpose of this work was to propose a new concept of musical notation editors, a new musical data model and a bit different approach to the computer music notation processing. I have also created an application that exploits the core principles brought by my thesis. Unfortunately, the program is still more in a stage of a prototype than of practically usable software. The implementation of a complete new editor would require considerably larger amount of time or a bigger team of programmers. Nevertheless, I think that it has proven the viability of the new concept and opened new possibilities for future work. Moreover, as you can see from the attachment A, the program is already sufficient to create relatively well-looking scores and so its most important limitation is now the lack of implemented symbols.

Of course, the last word is to be said by the users. If they find the result unsatisfactory or difficult to use, they will not use it. However, from the reactions of the musicians to whom I have presented my ideas and work, I think that once the editor will be completed, it will find its community.

As for the related work, after a short boom between 1985 and 1993, the interest of the academic world at this domain has considerably lowered and very few articles were published on this subject. A brief survey and evaluation of music representation systems from that period can be found in (5). Unfortunately none could be used for the purpose of my program. The most serious and relatively recent effort is the dissertation of Kai Lassfolk (3) published in 2004. It is also the only work concentrated on practical representation of music notation from the last ten years that I have found. There was another group of articles that focused on a grammar-based music composition (like for example (6)), but these are not suitable for representation of scores.

In the commercial sphere, the most important format is, I think, the MusicXML (7) developed by Recordare as a standard for interchange of music notation between different musical notation products and for online scores publishing. The MusicXML is now widely used and XSLT scripts are available that convert a MusicXML file into, for example, LilyPond format or MIDI. For this reason, I originally planned to use it for the common western music notation model. However, I finally found that there was no elegant way to connect it to the composition data and so I developed my own format. Nevertheless, I plan to implement a converter between my common western music representation and MusicXML.

To finish the section about the related work, I would like to briefly discuss the format of musical metadata. It is surprising that after nearly three decades of massive digitalization of resources, there is still no commonly used standard of classifying the musical scores. Many

libraries have developed their own formats and apparently no serious effort was made to bring order into the situation. There exists one general purpose metadata standard designed by Dublin Core Metadata Initiative (8) that is being slowly adopted. However, since it is general purpose, it is not suitable as is for classification of sheet music. Therefore, different formal specifications exist describing the mapping of the specific domain attributes to the Dublin Core fields. I have finally succeeded to find such a specification for the sheet music designed by the Sheet Music Consortium (9). It was the only format I found that tended to create a standard format. However, according to the date of the last update, the state of their website and the small number of available records, I highly doubt that they have succeeded.

Nevertheless, since the design of a new music metadata format was out of the scope of my thesis and I also did not want to simply add a new one to the existing set, I use this one in my editor. I made only one small change - I split the metadata fields into two parts: the score fields and the composition fields. The composition fields include the metadata that are possibly shared between different scores (like the composers or ensemble composition). The score metadata focuses on the publisher of the score and on the description of the publication. The metadata fields from all the presented compositions can then be easily (and automatically) added to the score part to form a single Sheet Music Consortium record.

The last section of this chapter describes the directions of the future development of my work. They can be grouped into 4 principal groups:

1. Completion of the editor
2. Transformation of the editor into a plug-in extensible platform
3. Integration with an Optical Music Recognition (OMR) software
4. Versioning of musical data
5. Sheet Music organization

The first point is clear. I would like to finish the implementation of the program, add the missing notation symbols, progressively implement the majority (if not all) the layout rules described in (1) and also add advanced formatting features like styling of the score content.

As for the second point, it was one of the design goals from the very beginning of the project. However, when I started the implementation I found that development of a reliable and robust plug-in host is a very complex task and since it was out of the scope of my thesis I decided to leave this functionality for the future. Nevertheless, I kept this intention still in my mind during the whole development process, so most of the elements are designed with respect to this modification. The principal extension point of the application will be the interior of the sections (especially of the music section), e.g. different notation systems, advanced image displaying, etc. The other important point will be the importers / and exporters from / to different formats. But the application will be designed in a way to allow access to all the project data and so to allow implementation of arbitrary feature that somebody might miss.

As I already stated several times, I created this editor partly to help the music schools to organize and access their sheet music. In order to use the features of this editor, the printed scores that form the majority of the materials in the school possession have to be digitalized. Doing it manually would be a tedious, error-prone and time and energy consuming process. Therefore, I would like to integrate my editor either with an existing OMR tool or to develop a new one for its needs.

I would also like to introduce versions of the musical project and a sort of source control and compare tool for them. Since the storage format is XML, one possible way to do it should be to use the existing tools for text data and implement the comparison tool by transformation and deserialization of the textual diff. However, this feature is in a very early design stage, so I have not analyzed it deeply yet.

The last point includes the usage of the project metadata and was already briefly sketched in the introduction. I would like to create a tool that will extract the metadata from the project, index them and store in a database with a link to their corresponding package. In this way, they can be easily searched and organized. The tool should be also able to generate a web portal allowing access to the stored scores. The final stage should be the connection of this database with the source control from the previous point. Such a platform would be very useful for music schools or publishers and it could also find its way to electronic libraries.

To sum up, I think that I have proposed a viable concept of music notation editors and that an application that will exploit this principles should easily find its way into to the community of musicians and / or music publishers. It is true that the advantages of my definition of a musical project become really significant when it comes to advanced scenarios like the learning books, song books or encyclopedias which, certainly, are not a kind of project that an ordinary user would create every day. However, since the editor was designed particularly for the needs of a music school or music publishing, I believe that these features will not be useless.

References

1. **Zelingr, Ivo.** *Notografie: Učebnice notografického záznamu.* Praha : Supraphon, 1986.

2. **Weaner, Maxwell, et al.** *Standard Music Notation Practice.* s.l. : Music Publisher's Association of the United States, Inc., 1966, 1993.

3. **Lassfolk, Kai.** *Music Notation as objects.* 2004. ISBN 952-10-2205-1.

4. **Microsoft Corp.** Ribbons. *MSDN Library.* [Online] http://msdn.microsoft.com/en-us/library/cc872782.aspx.

5. **Wiggins, Geraint, et al.** A Framework for the Evaluation of Music Representation Systems. *Computer Music Journal.* 1993, Vol. 3.

6. *Grammar-Based Music Composition.* **McCormack, John.** 1996, Complexity International.

7. **Recordare.** MusicXML Definition version 2.0. *Recordare.* [Online] 2010. http://www.recordare.com/xml.html.

8. **The Dublin Core® Metadata Initiative.** [Online] 2010. http://dublincore.org.

9. **The Sheet Music Consortium.** The Open Archives Initiative Sheetmusic Harvester and Service Provider. *UCLA Digital Library.* [Online] 2006. http://digital.library.ucla.edu/sheetmusic/OAIProject.html.

Appendix A
Editor Demonstration

A Simple Piano Learning Book

Two first pages of a hypothetical piano learning book created in my editor.

Simple Sample Piano Book

A Very Simple Demonstration of a Realization of an Advanced User Scenario in an Editor using the New Concept offered in my Master Thesis

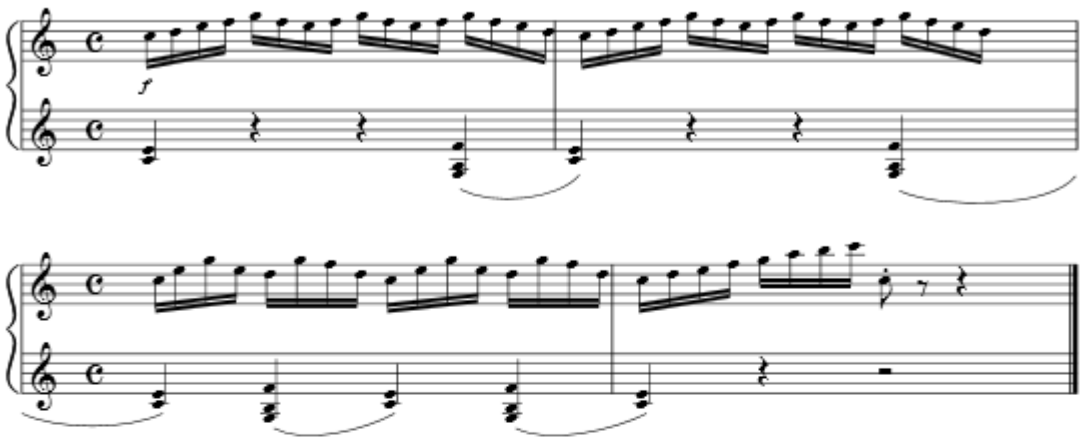
Lukáš Kopenec
Faculty of Mathematics and Physics
Charles University in Prague

2010

125 Passage Exercises

Allegro

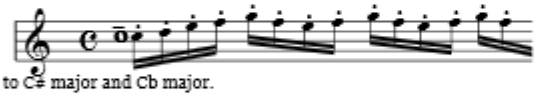
Carl Czerny, op. 261
(1791 - 1857)



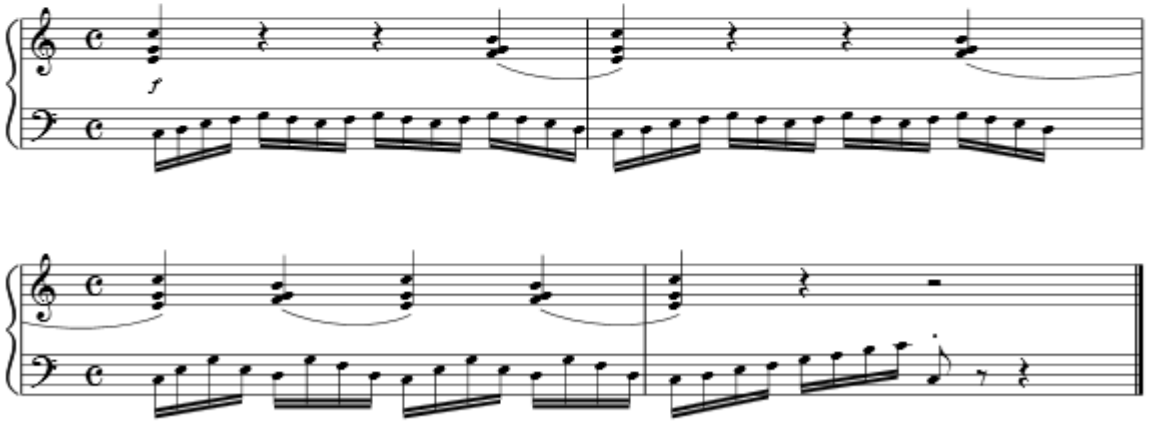
Practice the exercise in a moderate tempo separately by each hand. Even when you begin to increase the tempo, always start by playing it separately by each hand. ***Left hand:*** Always use the first rest to prepare the next chord. Keep free fingers freely above the keys. ***Right hand:*** Start by playing slowly with a firm legato and fingers slightly raised above the keys.

Practice in rhythms for 4 notes

Then in faster tempo non legato and in ***pp*** in *staccato leggiero*.

Variant:  (staccato from key and from above)

Transpose to C♯ major and Cb major.



The second exercise is identical with the first one only the tasks of the hands are swaped. Practice together with the first exercise. Make couples from other related exercises as well (e.g. 5 and 6, 9 and 10, 11 and 12, 19 and 20, 38 and 39, etc.)

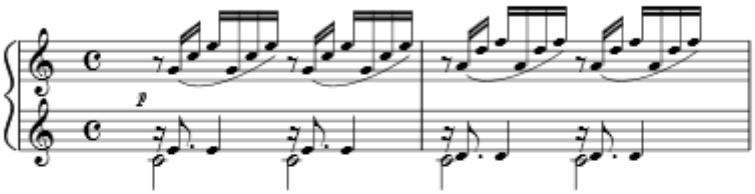
A Sample Collection of Famous Piano Compositions

A simple collection of compositions (just the beginnings of them) created in my editor. Notice the compositions previews in the table of contents, they are also created as music sections and are, therefore, automatically updated if a changed to the musical data occurs.

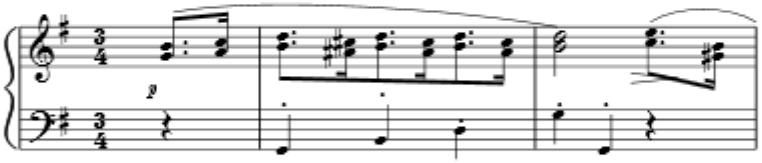
Both projects can be found on the attached CD.

INHALT

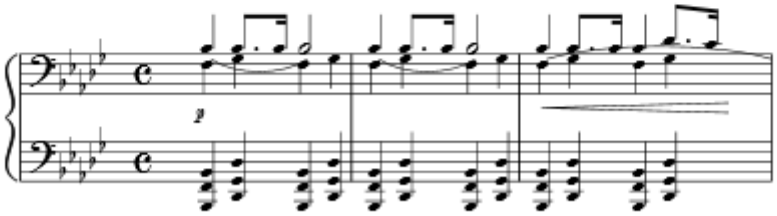
Bach: Präludium aus Wohltemperiertes Klavier I/1



Beethoven: Menuett G dur



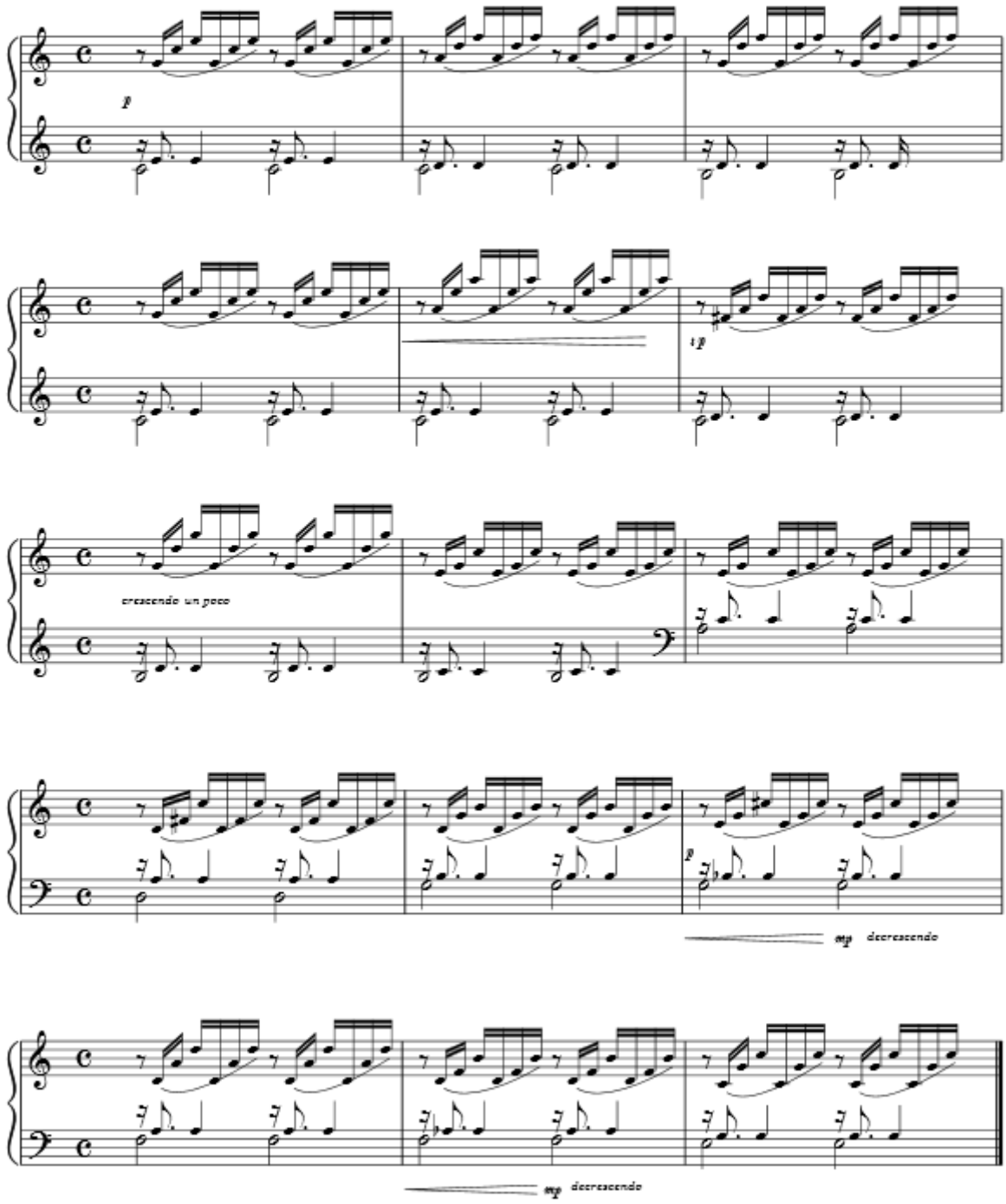
Chopin: Trauermarsch



Präludium aus Wohltemperiertes Klavier I/1

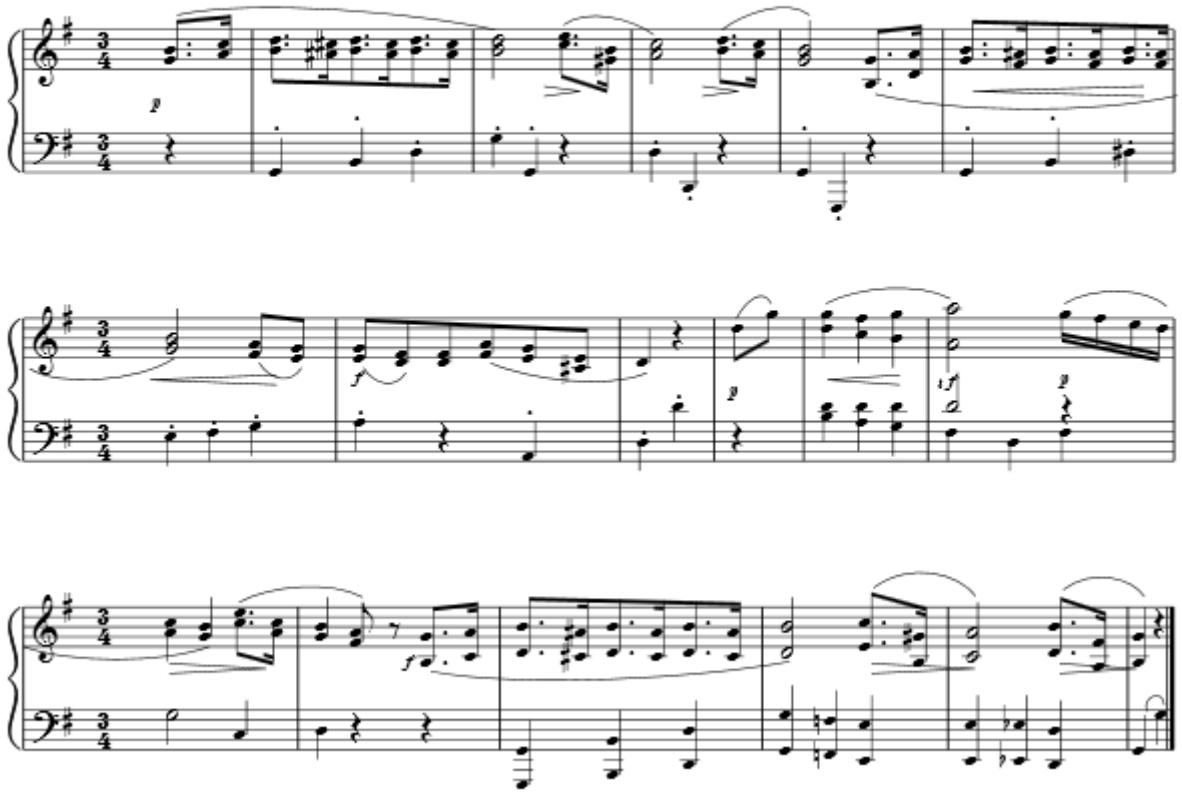
Johann Sebastian Bach
(1685 - 1750)

Moderato



Menuett G dur

Ludwig van Beethoven
(1770 - 1827)

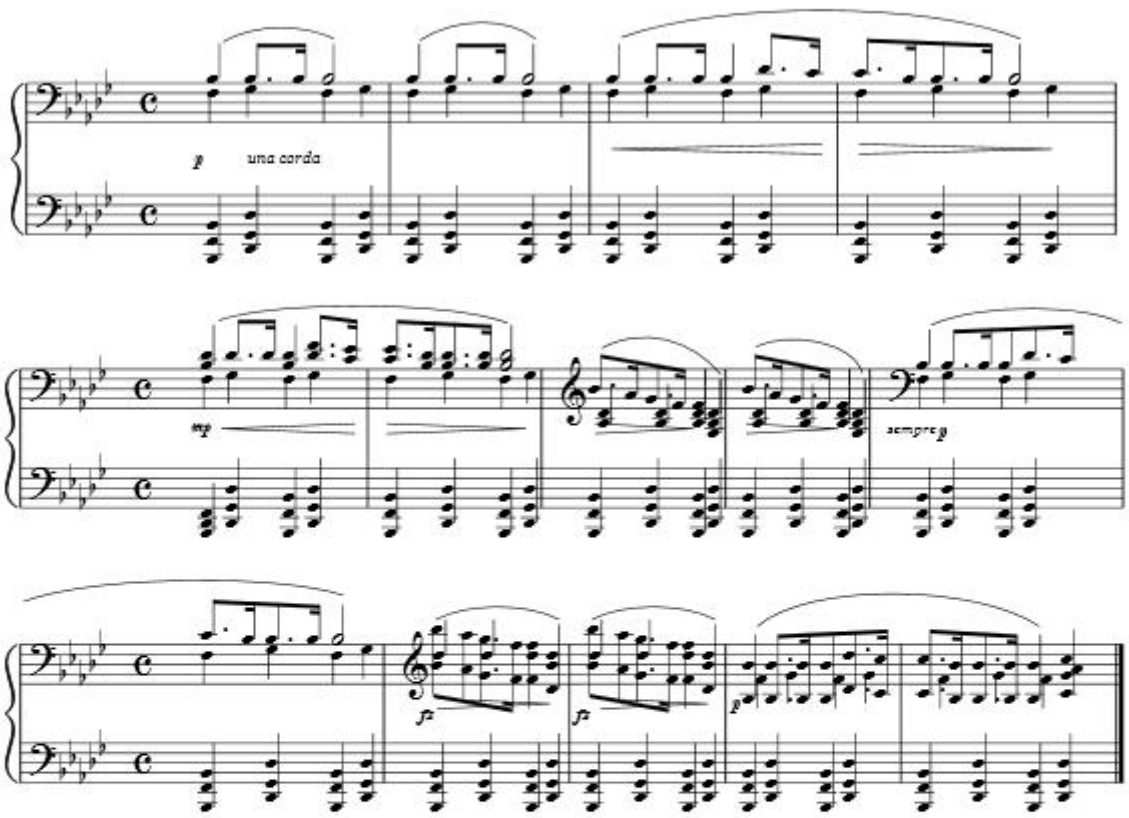


Trauermarsch

Sonata No. 2 B moll

Frédéric Chopin
(1810 - 1849)

Lento



Appendix B
Implemented Notation Symbols

The List of Common Western Music Notation Symbols implemented in the editor:

- Notes – standard notes of all used durations, augmentation dots
- Rests – rests of all used durations, augmentation dots
- Beam
- Articulation – standard types
- Clefs – standard clef types (C, F, G), staff position of the clef can be adjusted by the user giving the possibility to create a range of clefs, examples:
    - Treble Clef – G clef on the 2nd staff line
    - Bass Clef – F clef on the 4th staff line
    - Viola Clef – C clef on the 3rd staff line
    - Tenore Clef – C clef on the 4th staff line
- Accidentals, Key Signature – all commonly used types and scales
- Time Signature – all standard meters, user can specify the beat units and number of beats
- Dynamics, Dynamics Progress marks – all standard types, standard expressions
- Slur (legato)
- Piano Brace

The commonly used notation symbols that were not yet implemented:

- Tempo Expressions
- Ornaments, Cue & Grace Notes
- Chord Marks
- Tie
- Fermata
- Pedal marks
- Lyrics