Web Delivering of Music Scores

DE4.1.1
WEDEL Tool Kit

Version: 3.2
Date: 28/12/00
Responsible: DSI

Project Number: IST-1999-10165
Project Title: Web Delivering of Music Score
Deliverable Type: restricted

Deliverable Number: DE4.1.1
Contractual Date of Delivery: 31 December 2000
Actual Date of Delivery: 31 December 2000
Title of Deliverable: WEDEL Tool Kit
Work-Package contributing to the Deliverable: WP4
Nature of the Deliverable: RE
Author(s): DSI (Paolo Nesi, Pierfrancesco Bellini), CESVIT

Abstract:
User manual for the adoption of the WEDEL tool kit. The idea of the WTK is to provide a large set of functionalities including loading, saving, music manipulation, MIDI import/export, playing, executing, transposing, editing, navigating in music, printing, exporting, converting, classification and identification management, etc., under the support for the protection mechanisms in order to allow building more complex applications WEDELMUSIC compliant.

Keyword List:
music, internet delivering, audio format, image format, symbolic format, watermarking, protection, encryption, music distribution, copyrights, mp3, visually impaired people, speech music, Braille music, music analysis, music printing, music format.

Table of Content

1 Scope of the Document

The goal of this document is to be a manual for the adoption of the WEDEL tool kit.
The idea of the WTK is to provide a large set of functionalities including loading, saving, mudic manipulation, MIDI inputr export, playing, executing, transposing, editing, navigfating in music, printing, exporting, converting, classificatio and identification management, etc., under the support for the protection mechanisms in order to allow building more complex applications WEDELMUSIC compliant. These can be:

- Simple plug in of the WEDEL Music Editor such as the Analysis module of the IRCAM, or
- Real music editors such as the VIP Music editor of SVB, ILSP, and ARTEC

In the following section, both object oriented model and class documents are reported. On these basis the developers are guided to build their own application respecting the phylosophy of WEDELMUSIC object oriented model as defined and invented at DSI.

This document is private. Public version of this document will be distributed and could be used only under the signature of an agreement similar to that reported in the next subsection.

1.1 WEDEL Tool Kit Adoption Agreement, WTK adoption agreement

In this section a general description of the agreement that will be required to sing for adopting the WEDEL tool kit, WTK, is discussed. The adoption agreement includes a list of duties and rules that have to respect by the adopter of the WTK. These guidelines will be used for producing the agreement that will have to be signed by the all the users that would like to use the WEDEL tool kit for implementing their applications. The same agreement will have to be signed by the project partners since its early version or when a version capable of loading WEDEL format will be ready.

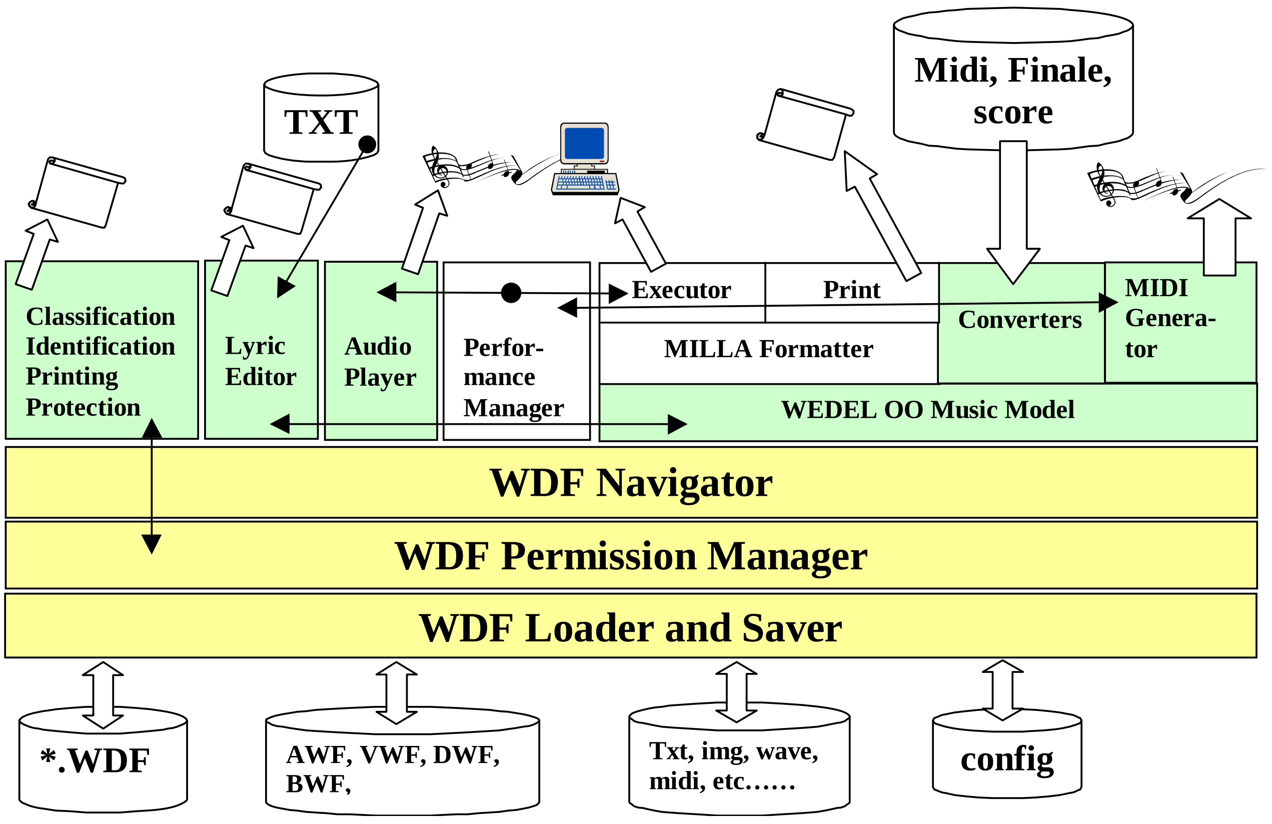
1. the WEDEL Tool kit will be delivered in OBJ or DLL code. The code cannot be unassembled.
2. the WTK will include a serial number. This cannot be modified.
3. The delivered WTK can be used only by personnel employed by the firm that has signed the agreement.
4. The firm cannot be make copies and redistribute the WTK to others.
5. The WTK can be used for implementing applications or plug in. Each application has to be registered to the DSI or WEDELMUSIC consortium. Non registered applications will be not capable of loading music in protected format. During the registration the DSI or WEDELMUSIC organisation will provide a number of ennobling the reading of protected objects.
6. There is no cases in which the WTK can be used for producing files containing description of music features coming from the Object Oriented model of music. The production of ASCII, Binary or Encrypted files by using WTK code is not allowed. The generation of temporary or hidden or in RAMDISK files is not allowed. All the above generation of files are not allowed.
7. The protection mechanisms enforced into the WTK that takes into account the protection aspect of the loaded WTK object have to be respected, no violation is permitted.
8. The builder that intends to use the WTK for implementing a saver into a different format has to communicate the intention to DSI or WEDELMUSIC organisation. This will be possible according to the permission manager and thus according to the rule for respecting prices and abilitation stated by the publishers in the WDFGPC.
9. The builder have to use only the Call Back prepared for them. Each different open call back will be disable for their use.

If one or more of the above rules will be non respected a specific legal action will be performed by WEDELMUSIC consortium or DSI for recovering the economical damage performed by the infraction.

2 The WEDEL Tool Kit

The WDF tool kit is comprised of all the components which are coloured in the next figure. This means that the following components are included into the WDF Tool Kit:

- **WDF Loader and Saver** -- loading and saving of the WDF files.
- **WDF Permission Manager** -- a tools for managing and controlling the permissions allowed
- **WDF Navigator** -- the music editor and integrator. The main entry for building WDF objects and for inspecting them.
- **Classification, Identification, Protection, Printing Manager** -- an object oriented model for collecting and managing all the information related to classification, protection, identification, and printing record listed in the previous paragraphs.
- **Lyric Editor** -- the editor for loading and assigning lyrics lines to parts.
- **Audio Player** --.a player for producing music from several audio formats.
- **WEDELOOMM**: WEDEL Object Oriented Music Model. The OO model of music used at the basis for modelling
- **Converters (symbolic Music)** -- tools for converting well known symbolic music formats such as Finale, SCORE and MIDI into the WEDEL format.
- **MIDI Generator** -- a tool for converting WEDEL music format to MIDI format.



The WDF object oriented model is derived from that of MOODS including the implementation of:

- New symbols;
- The management of all the related elements on the WEDEL object;
- The integration of image score and symbolic;
- The integration of audio files with all components: symbolic music and image score;
- converters from Finale, SCORE, and MIDI in the several versions;
- XML based loaders and savers for the WEDEL Model defined in the previous sections and comprised by several files;

- image score loader, IWF;
- an object oriented model of all the structures of the WEDEL HEADER;
- Font Table editor;
- html links towards external documents and starting from the music notation elements;
- the class hierarchy due to the management of classification, permission, protection and identification;
- the lyric editor and management of lyric in the model;
- the freaboard editor and management of tablaures in the Music model;
- transversal functionalities for managing audio files, producing MIDI, playing Wave files, etc.;
- model support for multistaff music pieces: arpa, piano, organ;
- of the model for The inclusion of tablature and fretboards;
- the support for the management of staff with 1, 2, 3 ,4 ,5, 6 lines;
- justification and line breaking algorithms based on style parameters;
- a strongly improved version of MILLA mechanism for managing visualisation rules independently on the music complexity including: beaming, up/down, etc.;
- a debugger for MILLA specification;
- WEDEL Editor for navigating on WEDEL Objects structure;
- Permission Manager for controlling the activities of WEDEL Editor and Music Editor according to the permission granted into the WEDEL Object;
- Adding several symbols: tablaturs, pedals, some clefs, alterations with parenthesis, several different noteheads, some horizontal symbols (slurs, crescendo, descrescendo, bend, tie, with different lines: dashed, dotted, continuous), da capo, a coda, etc.
- Support for Exchanging, Merging, joining, splitting layers to parts and viceversa
- Support for printing music, both main score and parts
- Support for controlling consistency of measures
- Support for formatting in automatic manner multi layers
- Export in MIDI format
- Player audio, WAVE and MP3
- Of mechanism for extracting excerpts from music scores
- Of mechanism for extracting excerpts from audio files
- Synchronisation mechanism from symbolic execution to the Audio file
- Representation and visualization of music score and the audio file
- Representation and visualization of music in image score format and the audio file
- Adjustment of the music execution rate in real time for music score
- Adjustment of the music execution rate in real time for music score in image score format.
- Moving, hiding, importing part
- Transposition algorithms
- Mechanisms for including lyric
- Reorganisation of the GUI of MOODS:
  - reducing hierarchy deep of menus for graphic symbols,
  - inclusion of classical menus for direct activation of functionalities
- adoption of the scroll bar for visualising page
- the mechanism for managing the printing view
- ….

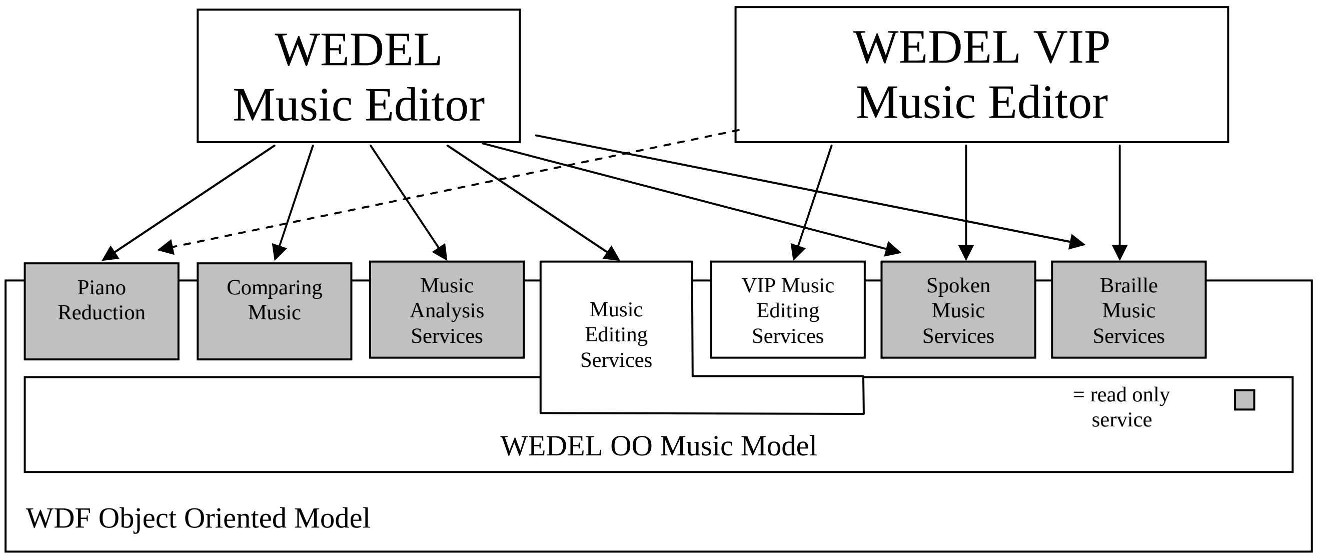
3 General Idea on the WEDEL Tool Kit, WTK

The idea is to provide (by DSI) to the involved partners the OO Music Model as a static library with the associated header files. In each class of the WDEDEL-OOMM have been added a virtual protected method for each type of service that have to be implemented by the partners.

The addition of several functionalities for supporting drivers for:

- Analysing music (MusicAnalysis());
- Reducing orchestra music to piano, (PianoReduction());
- Comparing music, (ComparingMusic());
- Printing Braille, (PrintingBraille());
- Describing music with speech music, (SpokenMusic());
- VIP Music Editing (VIPEditing());

These last aspects ahve been provided by DSI to other partners as Call Backs in the object oriented hierarchy of the WEDEL Object Oriented Music Model.



For example for Music Analysis have been added:

```
int Partitura::MusicAnalysis(MusicAnalysisData*)
int Battuta::MusicAnalysis(MusicAnalysisData*)
…
etc.
```

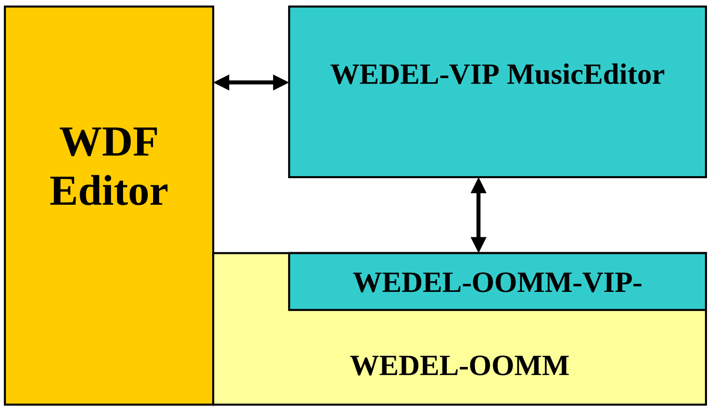
The partner have to implement the methods to provide the service using the public and internal services of the classes, the partner could also use an own defined class (e.g. MusicAnalysisData) where input information is provided (e.g. exact specification of the service required, file/device where to put the results, configuration parameters, etc.) where additional intermediate data can be stored and where aggregated results can be saved to be passed to the upper layer (typically user interface).

Two kinds of WTK have been identified:

1. the WTK for VIPs (this is a real tool kit in which the distribution will allow the implementation of totally new music editors).
2. the WTK for music analysis (this is the typical Plug In module, the WEDEL Music Editor remains the main music entry of the system and the added functionalities are attached via a menu and can interact with the WEDEL Object Oriented Music Model).

Please not that the Piano reduction can be used also from the tools developed for VIP people.

3.1 The WTK for VIPs



This tool kit is composed of:

- WDF-Editor, classes for user-interface (main window, dialogs) and classes for data representation of information stored in a WEDEL file.
- WEDEL-OOMM, the music classes with service methods.

The WDF-Editor may open the Standard Music Editor or an Extra Music Editor (in this case the editor for VIPs). The WDF-Editor calls the function *char *GetExtraMusicEditorName()* to retrive the name of the music editor, if this function returns NULL it means that there is not an extra music editor. This name is displayed in the contextual menu for opening a score and it is displayed in the selection of the default music editor.

The WDF-Editor starts the VIP Music Editor calling the function:

```
    int OpenExtraMusicEditor(WEDELFile* pWDF,Partitura* pMainScore, int partNumber)
```

where

- `WEDELFile *pWDF`,
  is the pointer to the loaded/new WEDEL File, from this pointer the information related to the components of the WEDEL file can be retrieved and modified.
- `Partitura *pMainScore`,
  is the pointer to the loaded main score or a new one, the VIP services can be called on this object to navigate and modify the object.
- `int partNumber`,
  is the part to be opened, 0 for the main score, 1 for the first part, 2 for the second etc.

The return value indicates if the Extra Music Editor is properly started (value 1) or not (value 0).
The objects referred by `pWDF` and `pMainScore` will be accessible until the WDF-Editor calls the `CloseExtraMusicEditor` function.

The WDF-editor closes the Extra Music Editor calling the function:

```
    int CloseExtraMusicEditor()
```

the return value indicates if the editor is properly closed (value 1) or not (value 0).

To maintain syncronized the list of the symbolic parts and the parts viewed in the music editor the following function can be used:

- `void WDFPartAdded(int partNumber, char* name)`
  this function is called from the WDF-Editor when a new part is added, partNumber is the number of the part and name its name.
- `void WDFPartDeleted(int partNumber)`
  this function is called from the WDF-Editor when a part is removed.
- `void MusicEditorPartAdded(int partNumber, char* name)`
  this function have to be called from the MusicEditor when a new part is added.
- `void MusicEditorPartDeleted(int partNumber)`
  this function has to be called from the MusicEditor when a part is removed.

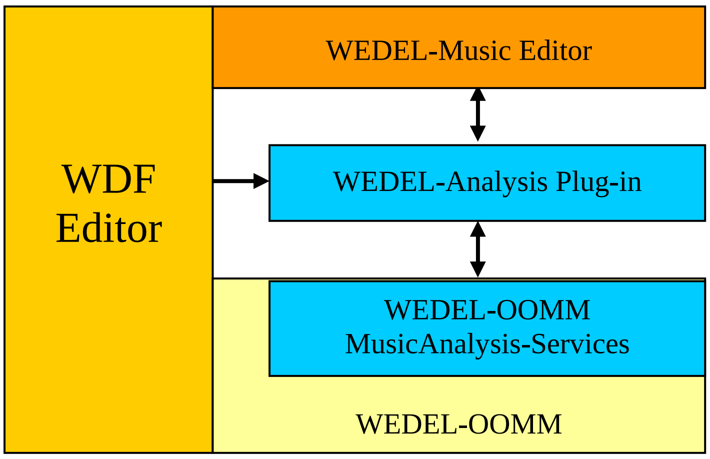
Once started the VIP-MusicEditor can use the VIP-Services to retrieve the information needed.

The partners involved into the development of the music editor for VIPs will provide a function for printing Braille and spoken music:

- `int PrintingBraille(WEDELFile *, Partitura *, PartNum, BraillePrintData & )`
- `int SpokenMusic(WEDELFile *, Partitura *, PartNum, SpokenMusicData & )`

An attribute for each entity of the OO hierarchy will be available. It has been implemented by including it as an attribute of DrawObject: int VIPUserData.

3.2 The WTK for Music Analysis:



This WTK is composed of:

- WDF-Editor, classes for user-interface (main window, dialogs) and classes for data representation of information stored in a WEDEL file.
- WEDEL-MusciEditor, classes for user-interface (score window, dialogs)
- WEDEL-OOMM, the music classes with service methods.

For the communication between the WEDELMUSICEditor and the WEDEL-Analysis plug in, a specific class is used:

```
class WEDELMUSICalAnalysis
{
public:
  WEDELMUSICalAnalysis(wxMEScoreFrame*, WEDELFile*, Partitura*);
  ~WEDELMUSICalAnalysis();
  int Initialize();
  wxMenu *CreateMenu();
  int HandleMenu(int menuID);
  bool DoIdle();
}
```

When the WEDEL-MusicEditor is started a new `WEDELMUSICalAnalysis` object is created specifying the music editor frame, the object representing the WEDELFile and the object representing the score and then method `Initialize`() is called. After that the `CreateMenu`() method is called and the wxMenu object returned is added to the menu bar. The IDs of the menus must start from 1000 and end before 1100.
The method `HandleMenu` is called when a menu with the menuID from 1000 to 1100 is selected.
The method `DoIdle` is called when the system is idle, it have to return TRUE if more idle time is needed, FALSE otherwise (in this case DoIdle is not called until a new event is received and the system becomes idle again). Finally the object is destroyed when the editor is closed.

The WEDELMUSICalAnalysis may access to the WEDELFile information using its interface and to the symbolic main score using the music analysis services implemented.
The wxMEScoreFrame is used by the WEDELMUSICalAnalysis object to highlight score elements.

A virtual method of class DrawObject has been added to highlight score elements:
`void DrawObject::Hilite(int processEvent,GDevice*)`
where :

- `processEvent` can be: BeforeDraw (0) ,AfterDraw (1),BeforePrint (2) ,AfterPrint (3)
- `GDevice*` is the device (GScreen or GPrint) where to draw the symbol.

Two attributes for each entity of the OO hierarchy are available for marking music objects. They have been implemented by including two attributes in the DrawObject class:

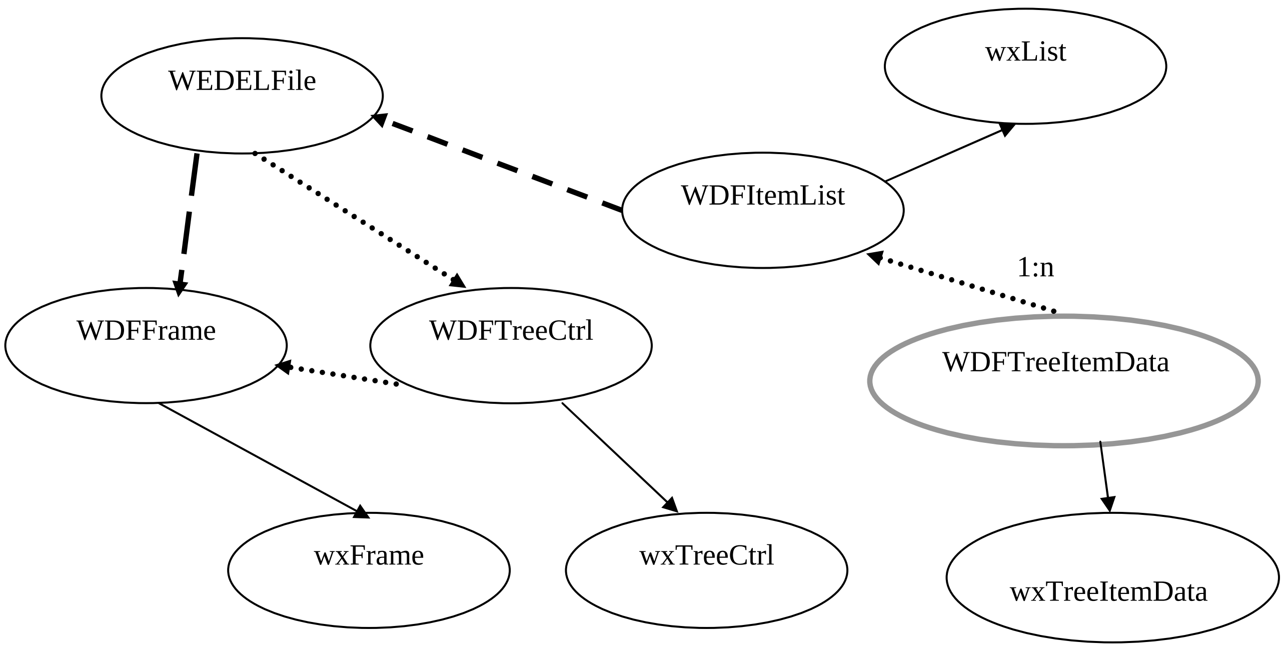
- int markType
- int markVariant

In the following section several details about the object oriented models used for modelling both the informative and symbolic music parts of the WDF objects are reported:

- WDF Editor Object Oriented Model
- WEDEL Object Oriented Music Model
- WEDEL Music Editor Object Oriented Model

4 WDF Editor Object Oriented Model

This is the so-called WDF Object Oriented Model and includes the aspects of classification, identification, protection, permission, printing, and the description of the WDF object structure.
The structure has been divided in two sections the GUI and Model of the WEDEL Object. The WEDELFile Class links them.



The above classes have the following functionalities:

- WEDELFile is the class to manage the WEDEL format file information: classification, identification, permission etc.
- WDFItemList is the class that collects all the data and information related to the WEDEL object
- WDFTreeCtrl is the class that manages the tree structure for visualising WEDEL items
- WDFFrame is the class related to the main window of WEDEL Editor
- WDFTreeItemData is the class that represents and models items into the WEDEL Object.

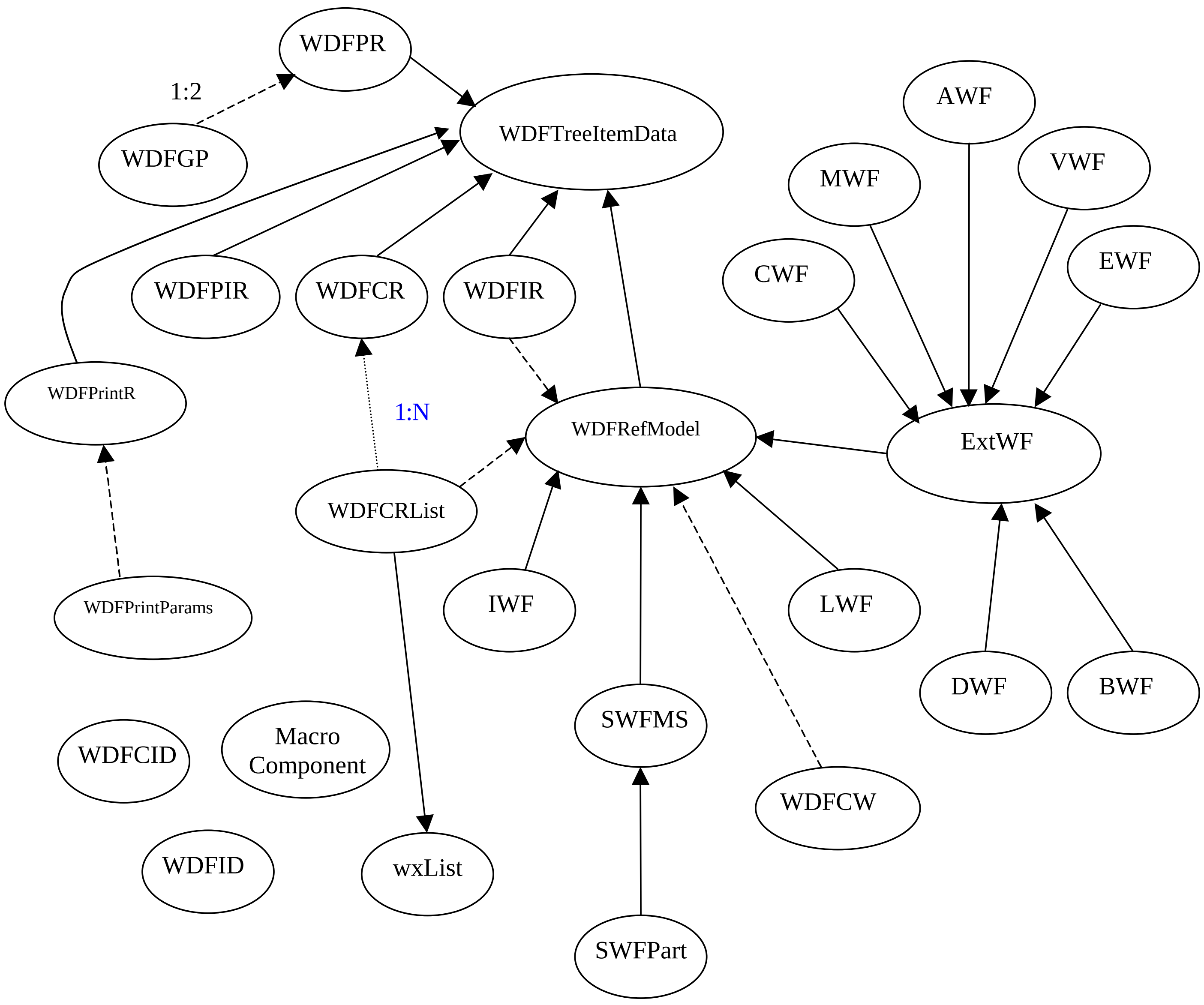
4.1 WDFTreeItemData structure: class diagram

In the previous sections, the structures of items, that compose the WEDEL Object, have been discussed. Following them, it's possible to build a table in which common attributes can be reported and then define classes and relationship.
The following table shows the relationship between items: columns are the common attributes while the rows are items.

	WDFID Father	WDF Checksum	Header Checksum	WDFDim	WDFIR	WDFCR	WDFCWP	Filename	Checksum	Dimension	WDFCID
EWF	X								X	X	X
MWF	X							X	X	X	X
CWF	X							X	X	X	X
AWF	X				X	X	X	X	X	X	X
VWF	X					X	X	X	X	X	X
BWF	X					X	X	X	X	X	X
DWF	X				X	X		X	X	X	X
LWF	X					X			X	X	X
IWF					X	X	X		X	X	X
SWFPart					X	X	X		X	X	X

SWFMS					X	X	X		X	X	X
WDFIR	X										
WDFCR											
WDFPIR											
WDFPR		X	X	X							

These relationships allow defining the following object oriented model:



WDFTreeItemData is the main class in which the WDFIDFather attribute has been included.
WDFRefModel class is derived from WDFTreeItemData in which the following attributes are included:

- Checksum
- Dimension
- WDFCID
- WDFIR (with an Is_Part_of link)
- WDFCWP (with an Is_Part_of link)
- WDFCR (an Is_Part_of link to WDFRCList, since it's possible to have more classifications)
- WDFIDFather inherited from WDFTreeItemData

ExtWDF class is derived from WDFrefModel. Its attributes are:

- Filename
- The attributes inherited from WDFRefModel

With this structure, some classes (see the table) can inherit attributes that wouldn't be necessary for them. For example, WDFIR inherits from WDFTreeItemData the WDFIDFather attributes, but it has not been defined in this class. For this reason, it's necessary to fix a default value.

Classes derived from WDFTreeItemData are:

1. WDFPR (WDFProtectionR)
2. WDFPIR (WDFPrintInfoR)
3. WDFCR (WDFClassificationR)
4. WDFIR (WDFIdentificationR).
5. WDFRefModel
6. WDFPrintR

Classes derived from WDFRefModel are:

1. IWF (ImgWF)
2. SWFMS (SymWFMainScore, SymWFPart)
3. LWF (LyrWF)
4. ExtWF

Classes derived from ExtWF are:

1. CWF (ChangesWF)
2. MWF (MillaWF)
3. AWF (AudioWF)
4. VWF (VideoWF)
5. EWF (ExecWF)
6. DWF (DocWF)
7. BWF (BitmapWF)

Class derived from SWFMS is:

1. SWFPart (SymWFPart)

Because it's possible to have a set of Classification Record, the WDFCRList class is are needed to manage it:

- WDFCRList is derived from wxList class and manages a list of WDFCR. It's linked with an Is_Part_of relationship to WDFRefModel.

WDFGPC class implements the WDF General Permission Code and it's linked with an Is_Part_of relationship to WDFPR, since it's included in the WDF Protection Record in accord to previous definition.

PrintParameter class implements the schema related to Music, Lyric and Braille Printing Parameters. They have the same attributes, so they can consider equal. It's linked to the PrintRecord class with an Is_Part_of relationship.

Other classes:

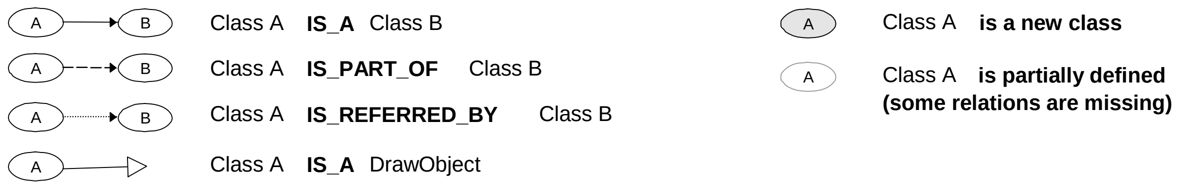
- WDFID: represents the identifier of the wedel object;
- WDFCID: represents the identifier of the component;
- MacroComponent: models a macro component;

For all leave classes selectors methods will be implemented to obtain needed information from the model, and in general a browsing mechanism will be developed to navigate into the structure. In this way from the WedelFile Class is possible to see all the items included in the wedel object file.

5 WEDEL Object Oriented Music Model

The class diagram has been divided in five sections:

1. The **Figure section**, where are reported the classes related to the notes, rests, chords and beams.
2. The **Symbols section**, where are reported the classes related to the symbols connected to a note/figure.
3. The **Measure section**, where are reported the classes related to the representation of a measure.
4. The **Score section**, where are reported the classes related to the score and the main score.
5. The **Graphic section**, where are reported the classes related to the graphic interface.


A
B
Class A IS_A Class B
Class A IS_PART_OF Class B
Class A IS_REFERRED_BY Class B
Class A IS_A DrawObject
Class A is a new class
Class A is partially defined (some relations are missing)

Legend

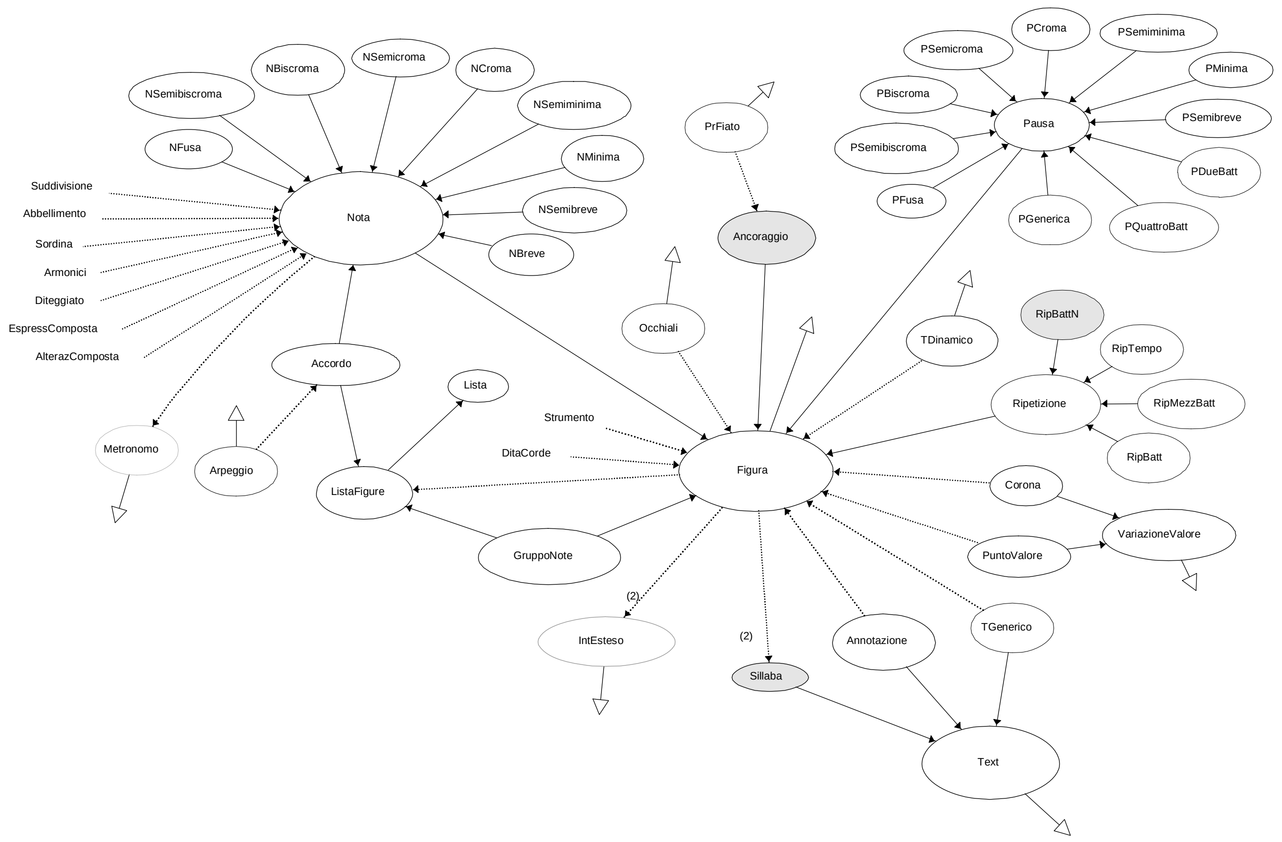
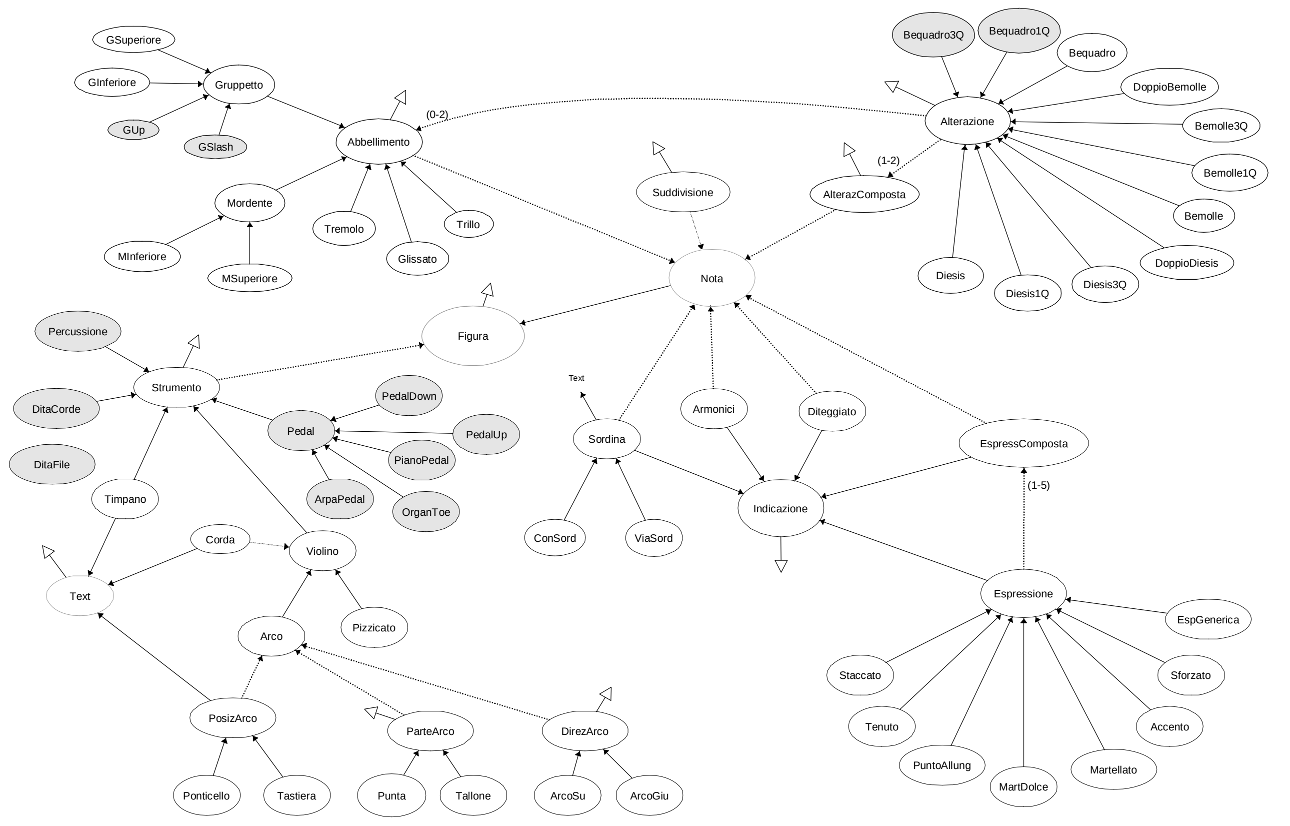
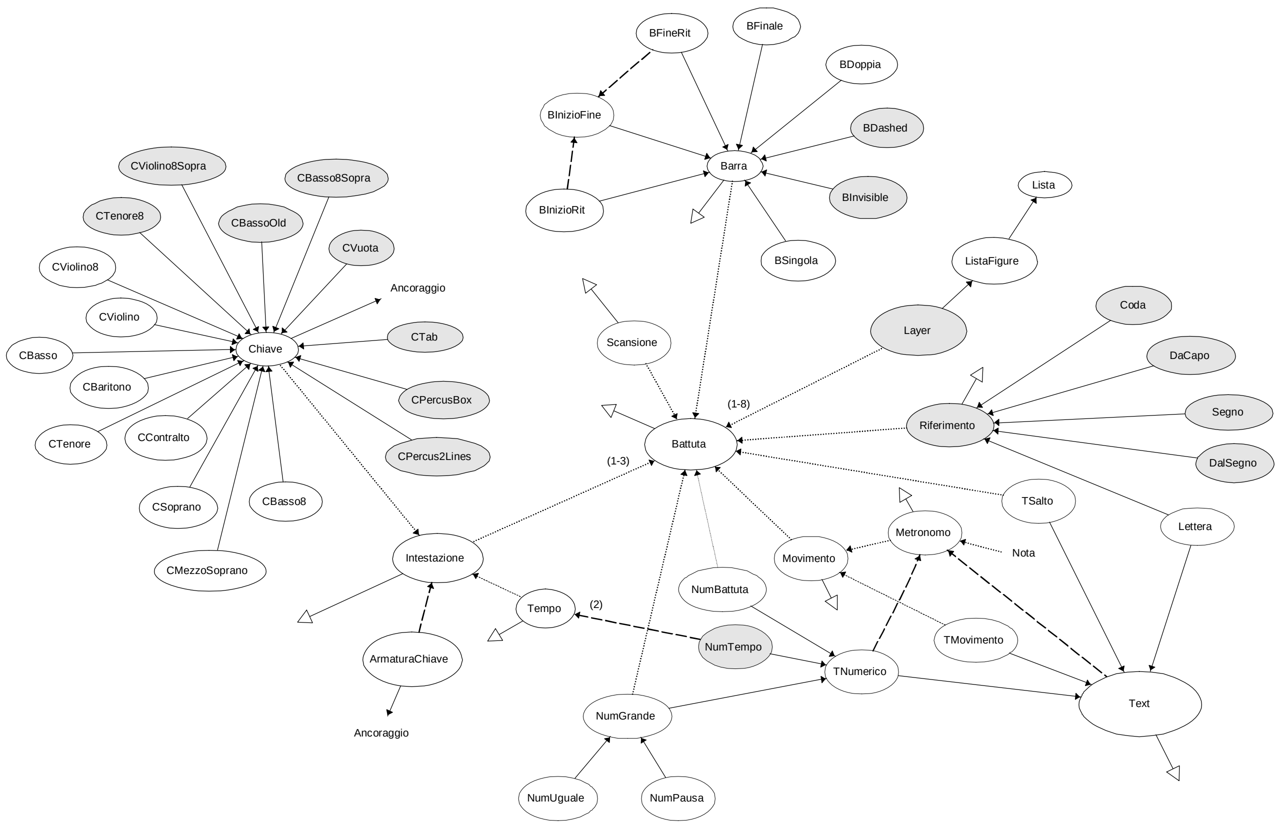


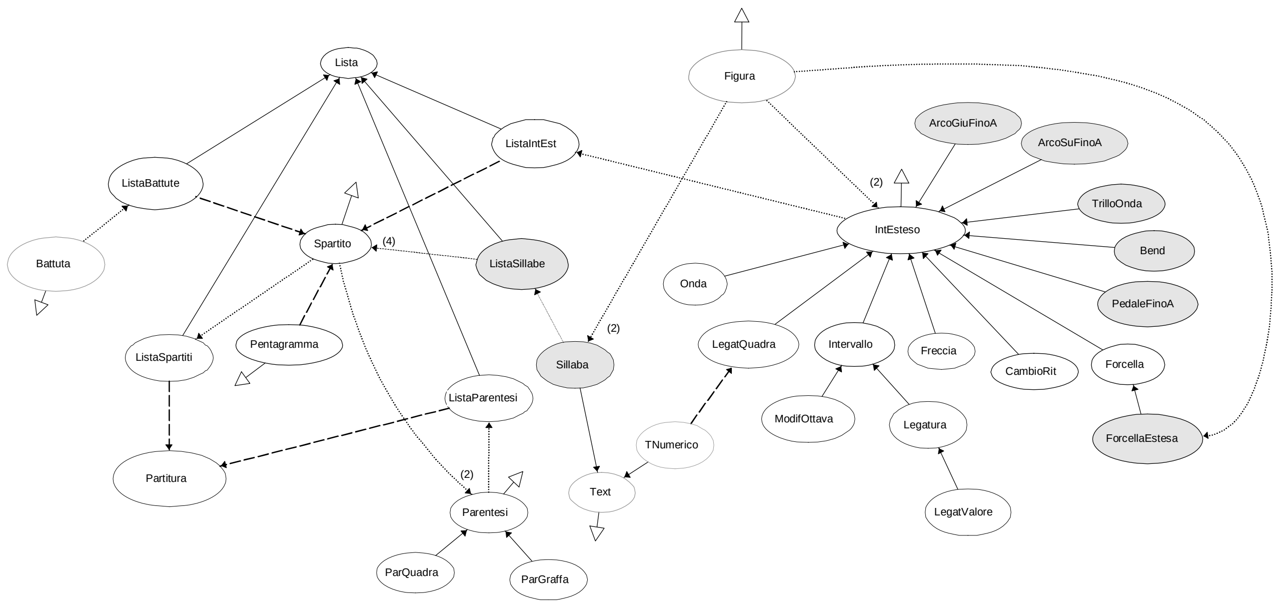
Figure section



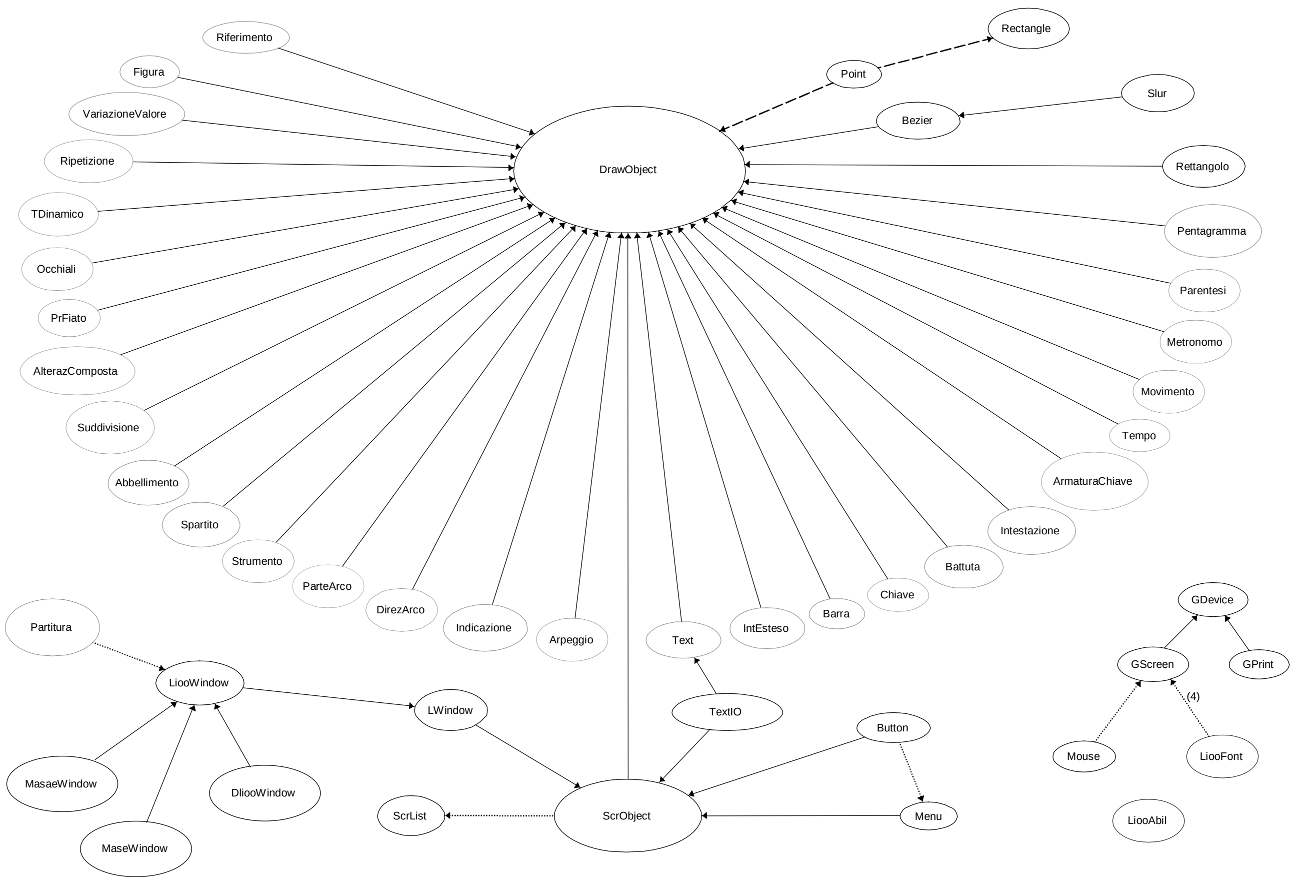
Symbol section



Measure section



Score section



Graphics section

5.1 WEDEL Object Oriented Model of Music and its analysis

The following Classes are the core of the Object oriented model for music. They are mainly used in the WEDEL music editor and in the WEDEL tool kit.
The WEDEL Object Oriented Music Model is also called in short WEDEL-OOMM.

The fundamental rule that has inspired this analysis was the realisation of a class for each musical symbol to be represented on the screen. To this end a class for each note, rest, chord and group of notes and one for each symbol that may accompany these figures or groups of figures have been identified. The remaining classes have been identified for better managing those just mentioned.
Links among classes that realise musical symbols have been studied to translate music syntax as much reliably as possible. Only in few cases the musical classification has not been respected, but this has been justified by the achievement of noticeable benefits in the organisation of the graphic part. Moreover, when the organisation of a class has presented more solutions, it has been chosen that makes the representation on screen easier.
All the classes that represents symbols to be drawn on the screen are derived from a root object called *DrawObject* that provides essentially a position, a color and other graphic information.

5.1.1 Notes, rests, chords, groups of notes

The analysis of the basic classes begins with the identification of all the symbols that refer to notes and rests: there are nine different kinds of notes which correspond to different duration values, and eleven kinds of rests. According to the names of the different notes and rests the following notes' classes:

1. *NSemibreve (Whole note)*
2. *NMinima (Half note)*
3. *NSemiminima (Quarter note)*
4. *NCroma (8th note)*
5. *NSemicroma (16th note)*
6. *NBiscroma (32nd note)*
7. *NSemibiscroma (64th note)*
8. *NFusa (128th note)*
9. *NBreve (two measure note)*

and the following rests' classes:

1. *PSemibreve (Whole rest)*
2. *PMinima (Half rest)*
3. *PSemiminima (Quarter rest)*
4. *PCroma (8th rest)*
5. *PSemicroma (16th rest)*
6. *PBiscroma (32nd rest)*
7. *PSemibiscroma (64th rest)*
8. *PFusa (128th rest)*
9. *PDueBatt (two measure rest)*
10. *PQuattroBat (four measure rest)*
11. *PGenerica (generic n-measure rest)*

have been named.

Each class must contain at least the method needed to be drawn. Since the different kinds of notes behave in the same way and have common graphic elements (e.g. full or empty notehead and the stem), the classes related to the notes have been generalised with the class *Nota*. Concerning the rests the same considerations can be made; the class *Pausa* has been introduced although the figures of rest have limited possibilities with respect to those of note. To generalise *Nota* and *Pausa* the class *Figura* has been then identified.

Concerning the spacing of figures inside a measure, each figure has an attribute that represents the distance to the next figure, this attribute is set by a justification algorithm depending on the type of justification chosen by the user (linear or logarithmic) and on the "stretching" parameter.

It has been judged necessary or at least convenient to organise some structures as a list of objects; to this end the class *Lista* has been created, which contains all the methods to manage this kind of organisation. The two following classes are organised as a list:

- *GruppoNote*: manages groups of notes linked with bars (beams).
- *Accordo*: it groups note non-sequentially disposed on the line but overlapped in vertical sense (chord).

These classes contain notes, but in rare cases the groups of notes can contain also generic figures, thus *Accordo* and *GruppoNote* have been made descend from a unique class (*ListaFigure*), which is directly derived from *Lista*.
GruppoNote has been defined also son of *Figura* with the scope to treat it as a single figure. *Accordo*, on the other hand, has been considered son of *Nota* since the chord has many features peculiar to the note (anyway, *Accordo* too descends from *Figura*).
Considering these choices it appears clear that the class *Figura* has a more general meaning of “musical figure”. To be remarked is also that identifying *GruppoNote* as a list of figures implies that in the group of notes is possible to insert also rests and chords.

5.1.2 Musical Symbols

In general musical symbols can be logically connected to:

1. a note (e.g. staccato, tenuto, accidentals etc.)
2. a rest (e.g. augmentation dot)
3. a note or a rest (e.g. augmentation dot)
4. between two consecutive notes (e.g. tremolo, glissando)
5. from a note to another note (e.g. slur)
6. from a point between two figures or from a figure to a point between two figure or to a figure (e.g. crescendo/diminuendo)
7. a measure (e.g. metronome indication)

So symbols can be classified in four categories:

1. symbols connected to a single figure (note/chord or rest), this type of connection has been modelled with a pointer to the symbol in the figure;
2. symbol between two consecutive figures, also this type of connection is modelled with a pointer to the symbol in the first figure;
3. symbol that span over many figures, this type of connection is modelled with two pointers to the starting and ending figures in the symbol;
4. symbol connected to a measure, this type of connection is modelled with a pointer to the symbol in the measure.

To be noted that to correctly model point 6. we need another type of figure called *Ancoraggio* (Anchorage) that is invisible and models a point between two figures, it has an attribute stating the distance of the anchorage from the preceding figure (note, rest or even another anchorage).

5.1.3 Accidentals and changes of value

A note may be preceded by one or two symbols of accidentals, to be drawn immediately left and at the same height of the note. Musical accidentals that have been identified are:

- *Diesis (Sharp)*
- *Bemolle (Flat)*
- *Bequadro (Natural)*
- *DoppioDiesis (Double sharp)*
- *DoppioBemolle (Double flat)*
- *Diesis1Q*
- *Diesis3Q*
- *Bemolle1Q*
- *Bemolle3Q*
- *Bequadro1Q*
- *Bequadro3Q*

A class is defined for each symbol and all are son of class *Alterazione.* Since a note can have one or two accidentals the class *AlterazioneComposta* that manage the composition of up to two accidentals has been introduced and this class is placed IS_REFERRED_BY Nota. Concerning the key signature, the class

ArmaturaChiave has been introduced, with the aim of drawing lists of accidentals on the staff. Because of this characteristic of the class it would have been possible to derive *ArmaturaChiave* from *Lista*, but, since it exists a limited number of possible accidentals, it has been preferred to codify them. In this way *ArmaturaChiave* is still an independent class.

Concerning the changes of value, the class that generalises the symbols (*VariazioniValore*) has been introduced and has been specialised in the classes:

- *PuntoValore*: it draws one or more value dots.
- *Corona*: it draws the fermata symbol in both the possible versions which depend on its position above or below the line.

Since the symbols that correspond to these two classes can be applied both to notes and rests, *PuntoValore* and *Corona* have assumed a relation IS_REFERRED_BY with the class *Figura*. Since the same figure can have both the dot and the fermata, it would be impossible substituting the two IS_REFERRED_BY with a single one directly from *VariazioniValore* to *Figura*.
Among the changes of value is included the tie. This one, because of its function of connection of two notes, has totally different features with respect to the dot and the fermata. From the graphic point of view, the tie can be considered a particular kind of slur, as well as the generic slur can be considered as one of the symbols that are extended on sequences of figures. These are defined as “interval symbols”.

5.1.4 Interval symbols

The 8-va symbol and the crescendo and diminuendo symbols belong, among others, to the category of interval symbols. The class that generalises the interval symbols has been called *IntEsteso*. Due to the characteristics of this category of symbols, it is appeared convenient to put a double relation IS_REFERRED_BY between *Figura* and *IntEsteso*; in this way it is the *IntEsteso* class that knows on which figure it has to start and on which one it has to end. Thus, unlike the symbols classified until now that are managed by the figure or the note which they are referred to, a class derived from *IntEsteso* is capable to place and draw the corresponding graphic object in a complete independence from the figures that it includes. The relation IS_REFERRED_BY has been put on *Figura* because some interval symbols can start and end on notes or rests. Anyway, the *IntEsteso* class has been specialised in the new class *Intervallo*, which generalises the symbols that can start and end only on notes. Therefore *Intervallo*, that inherits from IntEsteso two pointers to figures, will have to refuse the pointers to classes not derived from Nota. As son of *Intervallo* the class *Legatura*, which represents the bowed slur placed between two notes (not necessarily consecutive), has been introduced, and can represent a tie, a slur or a phrase mark. Since the tie begins on a note and ends on the successive and has a completely different meaning with respect to the other two kinds of slurs, it has been introduced in a distinct class (*LegatValore*), which derives from *Legatura*.

5.1.5 Irregular groups

From the graphic point of view an irregular group appears like a sequence of figures marked by a generic slur that includes, beside the above mentioned sequence, a number that shows the change of value that the figures undergo. Such slur can have the usual bowed shape or can be squared; in some cases can be also omitted. Since the squared slur is the most used in the irregular groups, it has been chosen to adopt always this kind of slur to represent irregular groups. A new class has therefore been introduced (*LegatQuadra*), son of *IntEsteso*, since the irregular groups can start and end with notes and rests it would not have been possible to derive *LegatQuadra* from *Legatura*.
It has been decided to consider the number that marks the irregular group as an independent symbol, to this end it has been necessary to insert classes for the management and performance of simple texts. The base class identified for this aim is *Text*, which offers the tools to manage generic texts and it is specialised in the class *TNumerico*. This latter is used for representing integer numbers and must contain an attribute of numerical kind that corresponds to the figure expressed by the types. Since the number that marks the irregular group represents the change of value undergone by the notes forming the group, each irregular group must necessarily contain a numerical text (it does not need to represent it but it must contain it) and thus TNumerico has been placed IS_PART_OF LegatQuadra.

5.1.6 Measure

With respect to the disposition of the figures on the staff is to be noticed that notes are grouped in measures and that the staff contains a set of measures (it is possible to interrupt the measure and complete it in a new staff but since it is very unusual in this paper such case is not considered). Then the features of this object have been identified: the measure is delimited by two vertical bar-lines that can assume 8 different forms, it

must contain figures and it represents a precise number of time unit. It has been decided to implement the measures with the class *Battuta*, which acts as container for the figures and can execute the control on the duration of these figures. Class *Layer* derived from *ListaFigure* has been introduced to represent the sequence of figures of a voice in a measure. For some instruments (e.g. piano, organ, harp) a measure is composed of sequences of figures over different staffs (two or three) while for others (e.g. violin) the figures sequences are over a single staff.
To model multi-staff measures up to eight figures layers can be used in a *Battuta* object and each figure has an attribute stating to which staff it belongs, in this way beaming across staff is quite easy to be realised.
In this way polyphonic scores can be written with all the figures of the layers belonging to the same staff.

Concerning the bar-lines, a class *Barra* has been introduced which has been specialised in different classes, one for each possible kind of bar-line:

1. *BSingola (single bar)*
2. *BDoppia (double bar)*
3. *BInizioRit (start repeat sign)*
4. *BFineRit (final repeat sign)*
5. *BFinale (final bar)*
6. *BDashed (dashed bar)*
7. *BInvisible (no bar)*

Each measure is delimited by two bar-lines. This has been modelled with the link *Barra* IS_REFERRED_BY *Battuta*, considering this bar-line as the right one, that is to say that that concludes the measure. This choice implies that in the middle of two measures it is impossible to place more than one bar-line, thus it is not possible to join the final repeat bar-line with that of start repeat. Instead of modifying this setting, by giving the possibility to insert more than one bar-line for each measure, it has been preferred to introduce a new class son of *Barra*, *BInizioFine (start and final repeat sign)*. Owing to the fact that this latter must (almost all the times) act as a fusion of the two repeat bar-lines, both *BInizioRit* and *BFineRit* have assumed a IS_PART_OF relation with *Barra*. It is important to notice that although the most natural relation between *Barra* and *Battuta* would be IS_PART_OF since all the measures have a right bar-line, a IS-REFERRED_BY relation has been preferred, because the polymorphism in C++ language can be implemented only with pointers. With the IS_PART_OF relation *Barra* would become and attribute of *Battuta* so if with this relation it would be always possible to assign to *Battuta* a specialisation of *Barra* on the other hand the virtual methods of the classes derived from *Barra* couldn't be used by *Battuta* because *Barra*'s methods would be executed.
In addition to those just mentioned the fundamental information that can be contained in the measure are clef, key and time signatures. The rule followed to place these elements into the measure is:

- The starting measure of each piece contains all these elements.
- The starting measure of each line has a clef and a key signature.
- The measure inside the line does not usually present these symbols.

When a measure presents clef, key and time signatures it is obvious that such symbols have an influence also on the following measures even if they do not appear. Only when the user want to change one of these symbols in a measure inside the line the symbol is placed at the beginning of the measure. The only exception is the clef which is represented with reduced dimensions in the measure preceding the change.
Focusing on the single classes, the class *Chiave*, specialised in the possible kinds of clefs, has been introduced:

- *CViolino (treble clef)*
- *CBasso (bass clef)*
- *CBaritono (baritone clef)*
- *CTenore (tenor clef)*
- *CContralto (alto clef)*
- *CMezzosoprano (mezzosoprano clef)*
- *CSoprano (soprano clef)*
- *CBasso8 (bass clef 8-vb)*
- *CBasso8Sopra (bass clef 8-va)*
- *CBassoOld (old bass clef)*
- *CViolino8 (treble clef 8-vb)*
- *CViolino8Sopra (treble clef 8-va)*

- *CTenore8 (tenor clef 8-vb)*
- *CVuota (empty clef)*
- *CTab (tablature clef)*
- *CPercusBox (clef for percussions)*
- *CPercus2Lines (clef for percussions)*

Since a change of clef can occur even in the middle of a measure class *Chiave* has been derived from *Ancoraggio* (and so from *Figura*), in this way this symbol can be inserted in a *ListaFigure* object and so in the middle of a *Layer*.
Since these classes present three different kind of symbols (except for tablature clef and percussion clefs), drawn in different heights on the staff, it would have been more convenient from a graphic point of view to introduce only three classes corresponding to the clefs of Violin, Fa and Do. It has been preferred the above described classification, in some way more "musical", because each class, beside representing a graphic symbol, could also express the meaning associated with the symbol.

The *ArmaturaChiave* class has been already considered, it is substantially represented as an enumeration type of all the possible key signatures. Since a change of key signature may occur even in the middle of a measure also *ArmaturaChiave* is son of *Ancoraggio.*

Concerning the *Tempo* class, it has been observed that usually the time is expressed as a fraction, so that it is appeared natural to associate to the class two texts of numerical kind but since also "3+2" have to be represented in the tempo fraction so class TNumerico has been specialised in *NumTempo* and *NumTempo* has a double relation IS_PART_OF with Tempo (numerator/denominator).
In order to collect the three classes to *Battuta* also a class *Intestazione* has been inserted putting *ArmaturaChiave* and *Tempo* IS_PART_OF *Intestazione*. Also *Chiave* should have assumed a IS_PART_OF relation with *Intestazione,* but IS_REFERRED_BY has been adopted, for the same reason explained for *Barra* (impossibility of using the virtual methods of the derived classes). Moreover *Intestazione* has been put in a IS_REFERRED_BY *Battuta* relation and a *Battuta* object may have from one to three *Intestazione* object one for every staff. Thanks to these choices each measure is always associated with a clef, a signature and a time; since all these symbols do not always appear together, *Battuta* must have the possibility to draw only a part of its *Intestazione*. The reason of all these choices is to confer to *Intestazione* not only a graphic but also a musical meaning. In fact the measures without Intestazione refer implicitly to the last drawn Intestazione. By the above described lay-out each heading has both a time and a key signature (although it is not obliged to draw them). A complete memorisation of the heading data has been preferred: only in drawing phase will be decided which data to represent.

5.1.7 The Score

A class (*Spartito*) has been associated to the whole score. This one has to contain first of all the sequence of measures that compose the score. To this end the class *ListaBattute* has been introduced in a IS_PART_OF *Spartito* relation. *ListaBattue* is created son of *Lista* and *Battuta* IS_REFERRED_BY *ListaBattute*. Since the interval symbols are independent of the measures, it was necessary to organise them as an independent list, modelled through the class *ListaIntEst* IS_A *Lista*, with *IntEsteso* IS_REFERRED_BY *ListaIntEst*; the new class has been put IS_PART_OF *Spartito*. Since *Spartito* manages the measures and the interval symbols also the design of the staffs has been entrusted to this class. The class *Pentagramma* has been thus placed in a IS_PART_OF relation with *Spartito*. This class has simply the task to draw the five lines of the staff while the management of the staffs will remain to *Spartito*.

5.1.7.1.1 Ornaments

A characteristic of the most part of the ornament symbols are the small notes; they appear in the trace note, in the appoggiatura and in the eventual preparation and conclusion of the trill. It is not the unique case in which the small notes are employed since these can also be represented in long series especially in the solo performances. Instead of creating distinct classes for the trace note, the appoggiatura, the preparation and conclusion of the trill and for the isolated small notes, a drastic decision has been taken by making uniform the management of the small notes and that of the notes. In order to make this possible the classes *Nota*, *Accordo* and *GruppoNote* have assumed the attribute Small Note that shows if the note (or chord or group of notes) has to be drawn with normal or reduced dimensions. This setting allows a small note to be independent from a note (e.g. the appoggiatura precedes a note) and therefore allows the writing of a small note in any position on the staff; this is particularly useful for writing the solo performances that contain long

phrases (even whole measures) consisting exclusively of small notes. Also for what concerns the simple trace note (that consists in a slashed 8th note) has been chosen an attribute (Acciaccatura) of the NCroma class, since only the 8th notes can become trace notes.
The other ornament symbols are meaningful only if they refer to a note. To generalise the ornament symbols a class *Abbellimento* has been introduced and specialised in:

- *Mordente*
- *Gruppetto*
- *Trillo*
- *Tremolo*
- *Glissato*

Since the mordent and the small group are present in two versions, "inferior" and "superior", it has been considered opportune to specialise *Mordente* with *MInferiore* and *MSuperiore* and *Gruppetto* with *GInferiore*, *GSuperiore, GUp* and *GSlash*. These last classes will draw the symbols whereas *Mordente* and *Gruppetto* become abstract classes. Owing to the fact that ornaments can refer only to notes and not to rests and since a note cannot have more than one ornament, Ornament has been placed in IS-REFERRED_BY relation with Nota.
The arpeggio symbol undergoes different classification. The corresponding class (*Arpeggio*) has been referred to *Accordo* instead of *Nota*. Such class draws the symbol in the two different versions: arpeggio upward and downward.
To be noted that *Tremolo* and *Glissato* are symbols connecting two consecutive notes and also all the *Gruppetto* symbols can be associated to a single note or can be placed between two consecutive notes (an attribute is used to select the proper meaning).
Moreover *Gruppetto* and *Mordente* can have associated up to two accidentals (one above and one below the symbol) while *Trillo* can have up to one accidental above the "***tr***" symbol, for this reason a double relation IS_REFERRED_BY of *Alterazione* with *Abbellimento* has been introduced.
Finally the trill can be drawn with the "***tr***" symbol followed by a sequence of little waves up to a certain point (the next figure or an anchor point), to model this type of symbol the class *TrilloOnda* son of *IntEsteso* has been introduced, and class Trillo has been made IS_PART_OF TrilloOnda.

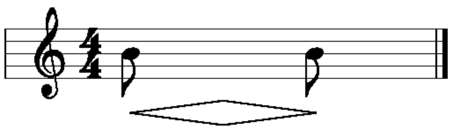
5.1.8 Agogic indications and dynamics

The agogic indications, which are written above the staff, are those that set the general movement of a music piece.
They appear above the first measure and can indicate the title of the corresponding movement (e.g. *Allegro maestoso*). For indications of this kind the class *Movimento* has been introduced; since for each measure it is possible to have a different movement indication it has been necessary to put Movimento IS_REFERRED_BY Battuta (the relation Movimento IS_REFERRED_BY Spartito would have been too limiting). Since the writing that expresses the movement can also be accompanied by a metronomic indication, it has been decided to introduce two new classes:

- *TMovimento* in a relation IS_A Text, it represents the text that specifies the movement (e.g. *Andante con moto*)
- *Metronomo* it contains a small note (eventually pointed), an integer number and some types (the equal sign or parenthesis). The classes *Nota*, *TNumerico* and *Text* are therefore been considered IS_PART_OF *Metronomo*. With this setting in the metronomic indication only one number will be specified instead of a couple of numbers divided by a dash.

Since a movement indication depends on a text, it has been considered *TMovimento* IS_PART_OF *Movimento*; the metronomic indication is optional so the relation *Metronomo* IS_REFERRED_BY *Movimento* has been considered.
Considering now the agogiche indications that can be placed anywhere in the piece of music (usually below the line), it has been referred to *Figura* a class derived from *Text*, called *TGenerico*, that allows representing a text above or below each figure (note, rest or space). This class can be used for texts of agogic kind (e.g. rallentando molto) or of dynamic kind (e.g. *cresc.*, abbreviation of crescendo), but also for textual indication of other kinds.
Dynamic indications in bold italics type (e.g. ***ff***) are represented by the class *TDinamico*, abbreviation of dynamic text. Such class has been considered in IS_REFERRED_BY relation with *Figura* even if a dynamic symbol has effects only on the notes. This choice has been made in order not to charge too much a note with symbols; by this setting, if a note has already symbol above or below it the dynamic sign can be moved in one of the spaces or rests preceding the note.

As already mentioned, the crescendo and diminuendo symbols have been classified as “interval symbols”, because they extend on more figures and can go beyond the bars delimiting the measures. Even if the crescendo and diminuendo symbols have effect only on the notes it would have been a mistake to consider the class *Forcella* son of *Intervallo*; with such a setting a *Forcella* could have started and ended only on a note. This would have made impossible representations as that reported in the following figure:



It has been therefore decided to allow the crescendo and diminuendo signs to start and end also on a anchorage or a rest, for this reason it has been necessary to consider *Forcella* IS_A *IntEsteso*.
Class *ForcellaEstesa* has been introduced as son of *Forcella* to represent a crescendo/diminuendo with three control points, the relation *Figura* IS_REFERRED_BY *ForcellaEstesa* has been introduced to model the middle point where there is the change of slope.

5.1.9 Expression signs

Two expression signs that we have already mentioned, the slur and the phrase mark, are comprised in the Legatura class. The *ListaIntEst* will have to permit the “nesting” of the slurs (for example two slurs included into a phrase mark).
The remaining expression signs have been generalised through the introduction of the class *Espressione*; from this the following classes have been derived:

- *Sforzato*: draws a small fork above or below the note.
- *Tenuto*: draws an horizontal dash above or below the note.
- *Accento*: draws a little wedge above or below the note.
- *Staccato*: draws a dot above or below the note.
- *PuntoAllung*: draws the punto allungato symbol above or below the note.
- *MartDolce*: draws the martellato dolce symbol above or below the note.
- *Martellato:* draws the martellato symbol above or below the note.
- *EspGenerica*: draws a user-defined expression symbol above or below the note.

Espressione class has been derived by a more generic class, *Indicazione*, introduced to generalise those symbols referred to *Nota* that do not modify the height or the value but that synthetizes execution prescriptions. Moreover since up to five expression signs can be associated to a note has been introduced the class *EspressComposta* that has to manage the composition of the symbols, it has been considered *EspressComposta* IS_REFERRED_BY *Nota* in order to give to each note the possibility to have a different expression symbol.

5.1.10 Abbreviations

The lines that cut the note stem, indicating particular methods of division, have been modelled with the class *Suddivisione*. Since these dashes can appear together with any kind of note and chord, the relation *Suddivisione* IS_REFERRED_BY Nota has been considered.
The dashes drawn between two notes that indicate the alternate repetition can appear in two forms, similar to classes already introduced.

- Two half notes connected through bars (from 1 to 4) present the same shape of a group of notes, so this kind of abbreviation has been represented by means of the class *GruppoNote*. This latter must present the following features:
    - Accept any sequence of notes and value rests strictly minor of a quarter. In this case *GruppoNote* has an essentially graphic function by improving the readability of a sequence without changing the value.
    - Accept a pair of half notes to indicate the alternate repetition of the notes. In this case *GruppoNote* represents an abbreviation and implies a cut into halves of the whole value of the notes contained as well.

- When the bars that connect two value notes are drawn on the staff they appear like a tremolo ornament, so that also in this case it has been judged unnecessary to add a new class. The class tremolo will have to consider the drawing of a number from 1 to 4 oblique bars.

The signs of time repetition, repetition of half measure, previous measure and the repetition of the previous measure n-times have been modelled in the classes *RipTempo, RipMezzaBatt, RipBatt, RipBattN,* respectively, which have been generalised with the class *Ripetizione*. These abbreviation symbols have to be placed on the staff in substitution of figures, as a result the *Ripetizione* class has been considered son of the *Figura* class. In this way the measures can be managed as lists of figures and thus the classification of *Battuta* does not change.
The refrain has been already introduced by means of the classes *BInzioRit, BFineRit* and *BInizioFine*, which represent the bars delimiting the extreme points of a refrain. The symbols necessary to distinguish the first time from the second can be extended on sequences of figures that can last some measures. To this end the class that represent them (*CambioRit*) has been derived from *IntEsteso*; this class will draw the horizontal line and the number (1. or 2.).
The textual indications of repetition (e.g. *D. C. to end*) have been introduced with the simple class *TSalto*, son of Text, which has to supply the user with the textual types and with the two conventional signs of jump. Such class has been considered in a IS_REFERRED_BY relation with *Battuta* since these indications are written over a bar in order to indicate the repetition of entire measures.

5.1.11 The main score

The Spartito class, previously introduced, has the requirements to model a single staff orchestral part. Since the main score consists essentially in the parallel superposition of several parts, the main score has been identified with a class (*Partitura*) containing a list of parts. The procedure is similar to those described for the other kinds of lists already mentioned: Introduction of the class *ListaSpartiti* IS_A List, with *Spartito* IS_REFERRED_BY *ListaSpartiti* and *ListaSpartiti* IS_PART_OF *Partitura*.
In order to allow grouping the staffs of a score by means of a brace or of a square bracket, the class *ParGraffa*, *ParQuadra* and a generic class *Parentesi* (bracket) have been introduced. The class *Parentesi* has two references to the *Spartito* objects that represent the score where the bracket starts and where the bracket ends. Moreover the class *ListaParentesi* has been introduced to collect all the brackets of the main score and so we have that *ListaParentesi* IS_PART_OF *Partitura*.
The textual indications that concern the methods of subdivision of an instrumental group (*UNITI, a 2, DIVISI, Solo*) can be introduced by using the class *TGenerico*. It has been decided not to confer any “musical” meaning to *TGenerico* because introducing many textual classes referred to *Figura* could lead to two drawbacks:
Each instance of *Figura*, which is already overloaded with several pointers, would acquire others.
Since it is necessary to allow the user to insert personalised texts, the user has to select a different command for each kind of text he wants to write (dynamic, agogic, etc.), unless the interface user is able to classify each introduced text instancing the appropriate class for each category of text.
Since in the *TGenerico* class several indications meet, it is considered necessary to introduce a new class for the textual annotations that the director introduces into the score but that the orchestral is not interested in reading. This class has been called *Annotazione*, related to *Figura* (like *TGenerico*) and son of *Text*; its principal feature is that it has not to be written on the orchestral lectern.

5.1.12 Demarcation symbols

Each measure can be marked with a progressive number by using an attribute of the Battuta class. To represent such number the *NumBattuta* class, son of *TNumerico*, has been introduced; since not all measures have to represent their own number, a IS_REFERRED_BY relation has been introduced between *NumBattuta* and *Battuta*.
Concerning capital letters used for the demarcation, only the case in which this symbol appears upon a bar has been considered (the score analysed does not present letters of demarcation inside the measure). The new class *Lettera* referred to Battuta and derived from Text has been also introduced; its function consists in setting a big font and in consenting to use next to the alphabetical letters numbers as well (numbers are used in some editions instead of letters).
Representing the page number of the score on an electronic lectern has been considered unnecessary for the following reasons:

- The directorial staff is scrolled horizontally one measure after the other, consequently the score on the MASE/MASAE screen does not appear with the same paging of a book. Thus the page number of the score has no meaning and cannot be automatically obtained.

- Since the scroll of the pages of the orchestral lecterns (DLIOO, PDLOO) is commanded by the directorial lectern, when the director (or the archivist) goes to a certain point of the score, all the other lecterns must reach automatically the corresponding page.
- By using the classes introduced in this paragraph the electronic lectern can already use two methods of demarcation. Projecting a third method is seemed useless.

5.1.13 Rests on several measures

Three classes sons of Pausa have been introduced:

- *PDueBatt:* it represents the rest of two measures (a vertical dash on the third space of the staff).
- *PQuattroBatt:* it represents the rest of four measures (a vertical dash on the second and third space of the staff).
- *PGenerica:* it represents the generic rest of several measures (of the two version available of the symbol and horizontal dash has been chosen).

When a rest of this kind compares on the staff a number is placed in the middle of the space occupied by the measure that represents the quantity of empty measures. There's an other kind of number the assumes the same position: it is the progressive number that appears in a sequence of equal measures. The only difference between the two numbers is the dimension of the character: the number that accompanies the rest is bigger. In order to generalise these two symbols a new class (*NumGrande*) has been derived from *TNumerico*. Such class has been specialised in *NumPausa* (it draws the numbers on the empty measures) and in *NumUguale* (it draws numbers on equal measures). As it has been already done in other cases, this hierarchy has been set taking into account the graphic features rather than the musical meaning. Since the measure must manage these numerical symbols, *NumGrande* has been considered IS_REFERRED_BY *Battuta*. In this way a measure can assume only one type of "big" number, so it cannot contemporaneously present the number related to the rest and that related to the equal measures; this is not a limitation but corresponds to what practically happened in music.

5.1.14 Time scanning

In order to consent to the user the annotation of time scanning, which is employed by the director, it has been decided to adopt the symbol consisting of several vertical dashes, which seems the more commonly employed, instead of using textual indications as "in 2". In order to manage the representation of the dashes the *Scansione* class has been introduced, related IS_REFERRED_BY with *Battuta* (since the director can change scanning only between a measure and the successive).

5.1.15 New agogic indications and critical passages

For the representation of symbols in the shape of wave or arrow the classes *Onda* and *Freccia* have been introduced. Considered the graphic characteristics of the two symbols these classes have been derived from *IntEsteso*.
The two symbols used to attract the attention (glasses and exclamation mark) are considered equivalent. Since the glasses seemed more effective for the scope, it has been decided to represent only this symbol by introducing the class *Occhiali*. This has been referred to *Figura* in order to be drawn in any position on the staff.
To highlight the bar-lines that delimit the refrains no new classes have been introduced. So *BInizioRit*, *BFineRit* and *BInizioFine* will draw the corresponding bar-lines with or without the oblique lines for the highlightment.
The symbols that indicate to turn page ("turn straightaway" and indication of empty measures) are considered useless in an electronic lectern, whose principal scope is the automatic scrolling of the score. For this reason they have not been classified.

5.1.16 Instrumental indications

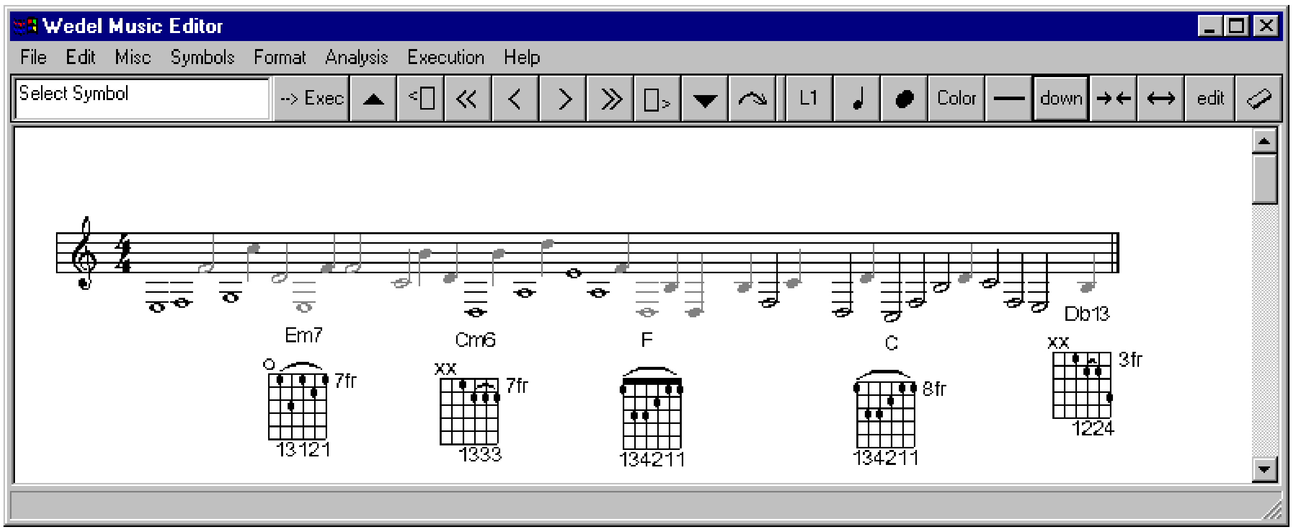
In order to represent the fingering a simple class (*Diteggiato*) has been introduced, son of *Indicazione*.
Concerning the indication of mute, the *Sordina* class, created son of *Indicazione* and of *Text* and specialised in the *ConSord* and *ViaSord* classes, has been introduced. An attribute of Sordina indicates the type of representation: for bow instruments (a comb-like symbol), for ottoni (+/-) or as text ("con sord."/"via sord.").
Since the fingering and the mute are independent each other and with respect to the other symbols referred to *Nota*, the relation IS_REFERRED_BY has been introduced both for *Diteggiato* and *Nota* and for *Sordina* and *Nota*.
For managing the signs that belong to the bow family the class *Violino* has been introduced. Since each note that appears on the bow instruments scores can be accompanied, besides by the fingering, by the indication

of the string a new class *Corda* in a IS_REFERRED_BY relation with Violino has been created. This latter has the aim to represent Roman numbers (from 1 to 4), and has been derived directly from Text. Concerning the remaining symbols related to bow instruments, it has been observed that the majority refers to the execution with the bow; as a consequence if a note must be executed as a pizzicato the only indication that can appear, besides those already introduced, is the text *pizz.*. This fact has suggested to specialise Violin in two classes corresponding to the two fundamental methods of producing the sound: *Arco* and *Pizzicato.* These classes have to represent the corresponding textual symbol (*arco* and *pizz.* respectively), but *Arco* has to manage also the representation of the signs that can be placed on a note when this must be played with a bow. The direction of the bow can be specified with the symbols *bow up* and *bow down* (the presence of one of them exclude the other), therefore the class *DirezArco*, in a IS_REFERRED_BY *Arco* relation is specialised in *ArcoSu* and *ArcoGiu*, has been introduced. Also the setting of the part of the bow to be used foresee the employ of two selfexcluding symbols, so the same setting of the *DirezArco* class has been used: the *ParteArco* class with the sons *Punta* and *Tallone*, in a IS_REFERRED_BY Arco relation, has been introduced. Totally analogue is the classification of *PosizArco* with the two son-classes (*Ponticello* and *Tastiera*); the only difference consists in the derivation of PosizArco from Text, due to the fact that *ponticello* and *tastiera* are textual indications. When on a note is present an indication like *bow up* or *tallone* it is obvious that such note is to be played with the bow, thus in these cases the Arco class has to give the possibility to represent or not the text *arco*. Concerning the symbol + (that means pizzicato with the left hand), it has been decided not to add a new class because the same sign, although with an other meaning, is available from the Diteggiato class.
As regards the remaining instrumental indications, two new classes have been identified:

- *PrFiato*: it represents one of the three symbols adopted to indicate to breathe.
- *Timpano*: it allows managing the indications for timpani or harp, consisting of squared texts eventually preceded by an arrow.

The former is a simple graphic class, but the latter having to manage texts that can have a different length assigned by the user, has been considered son of Text.
To generalise the classes related to the instrumental indications the *Strumento* class has been introduced, connected to Nota through a relation IS_REFERRED_BY. By this choice each note can assume symbols referred to a unique category of instruments. The PrFiato class has not been inserted in this hierarchy for the following reasons:
From the musical point of view the indication *breathe* is referred to a note but the symbol is placed between the note and the successive. Such behaviour is different from that of the other symbols referred to figures that have been previously described because these are placed by the figure (above, below, on the right or on the left).
Since the indication *breathe* is used by the majority of instruments, cannot depend on a particular family of instruments.
These considerations have suggested to consider a IS_REFERRED_BY relation between *PrFiato* and *Ancoraggio*; in this way the corresponding symbol can be easily placed between two notes and is not bound by the presence on the notes of other instrumental indications. On the other hand the *Timpano* and *Violino* classes have been considered sons of *Strumento*: since the two classes have little in common the class *Strumento* is an abstract class. Moreover, since most of the classes referred to *Figura* or *Nota* have the function to “represent themselves” on the screen, it must be made clear that *Strumento*, generalising *Violino*, has to manage several symbols.
To represent the percussion symbols to be added above or below the note class *Percussione* son of *Strumento* has been introduced, an attribute is used to state which symbol is represented.
Moreover for instrument like guitar is possible to associate to a figure a freat board, the class *DitaCorde* son of *Strumento* has been introduced to represent a freat board. Class *DitaFile* has been introduced to manage a database of the possible freat boards.
Finally to represent indications for instruments with pedals have been introduced the classes:

- *PedalDown*
- *PedalUp*
- *PianoPedal*
- *ArpaPedal*
- *OrganToe*



All these classes are son of *Pedal* that has been derived from *Strumento*.

5.1.17 Lyrics

The lyrics below a score has been represented as a list of syllables and each syllable is synchronised with a certain figure (note). In particular a syllable can start on a figure and end on the same figure or end on another figure (in this way the syllable is extended up to the specified figure). So *ListaSillabe* son of *Lista* has been introduced to collect the lyric text and class *Sillaba* son of *Text* has been defined to represent the text of the syllable.
Since a score can have up to four lyric lines *ListaSillabe* IS_PART_OF *Spartito* four times.

5.1.18 Guitar symbols

The guitar presents notations on the score that are now of common use; among others the most important are:

1. the explanation of the finger configuration for playing a chord (*fretboard*);
2. the indication of the positions on the guitar's keyboard corresponding to the execution of a classical musical text (*tablature*).

The purpose is, therefore, to permit to the user:

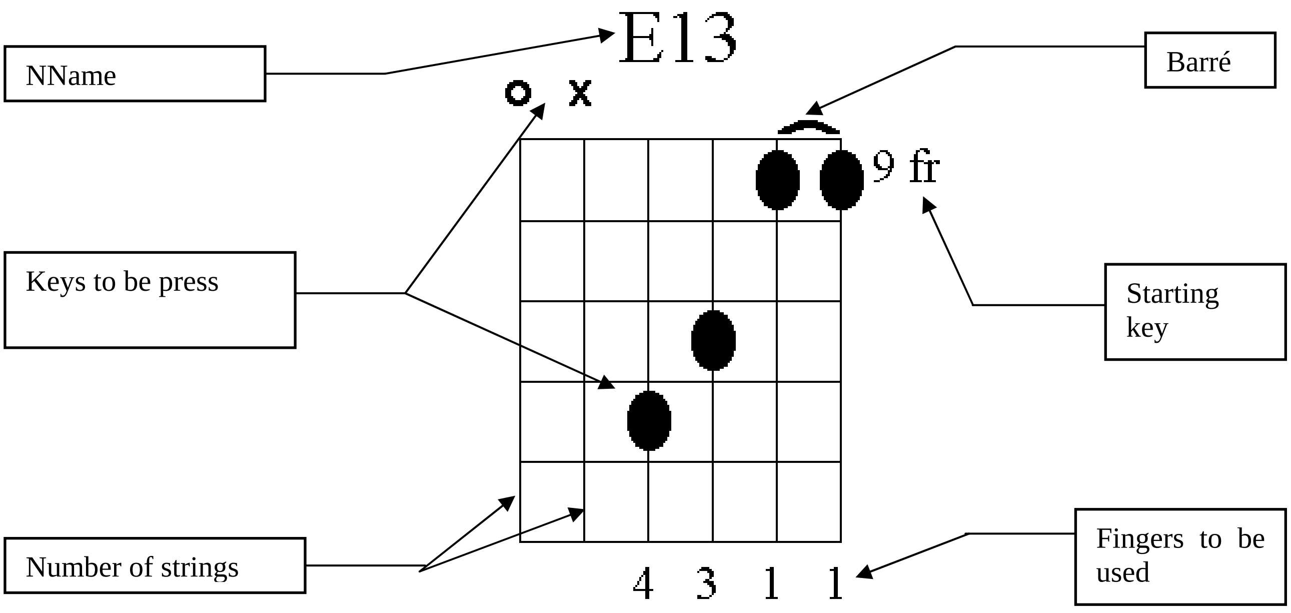
1. the choice and the insertion of fretboards on the score;
2. the possibility to insert notes with a notehead different from the normal one, that eventually includes alphanumerical indications inside itself.

The problem is clearly divided into two independent parts:

1. the creation of a class that contains the information and the methods for the fretboards.
2. the broadening of the class Nota in order to include and manage the information on the different types of notehead.

5.1.19 Fretboard

The standard positions of the chords played on the guitar have been already defined in the today's musical culture and our analysis starts exactly from an archive containing this data.
Here below an example of fretboard that includes all the most important symbols is shown.

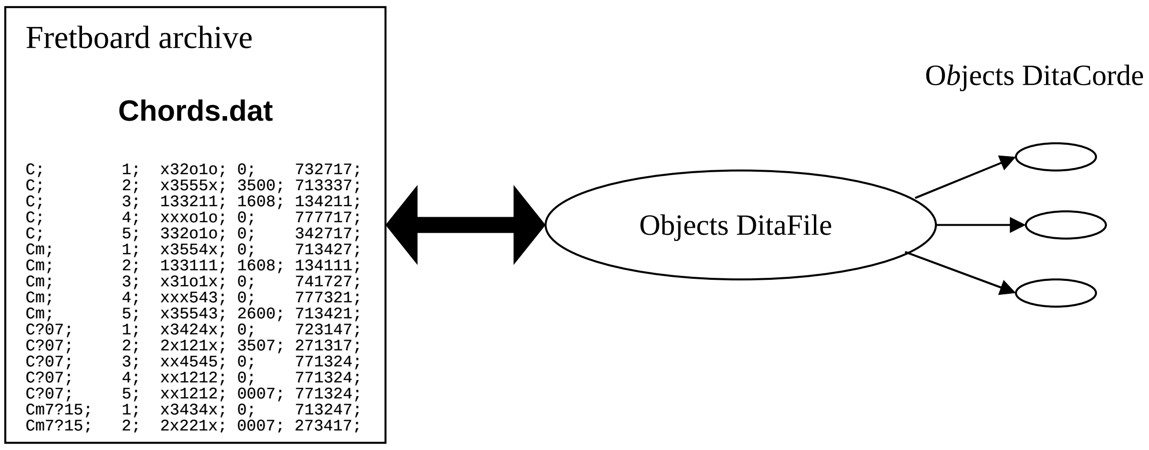


The fretboard has to contain the following information:

- the name;
- the number of strings of the instrument;
- the keys to be press on each string;
- the fingers to be used;
- the starting key;
- the eventual barré.

The main methods of the class have to describe the fretboard according to the different cases of output:

- visualisation in the musical window (or in the preview);
- saving;
- printing.



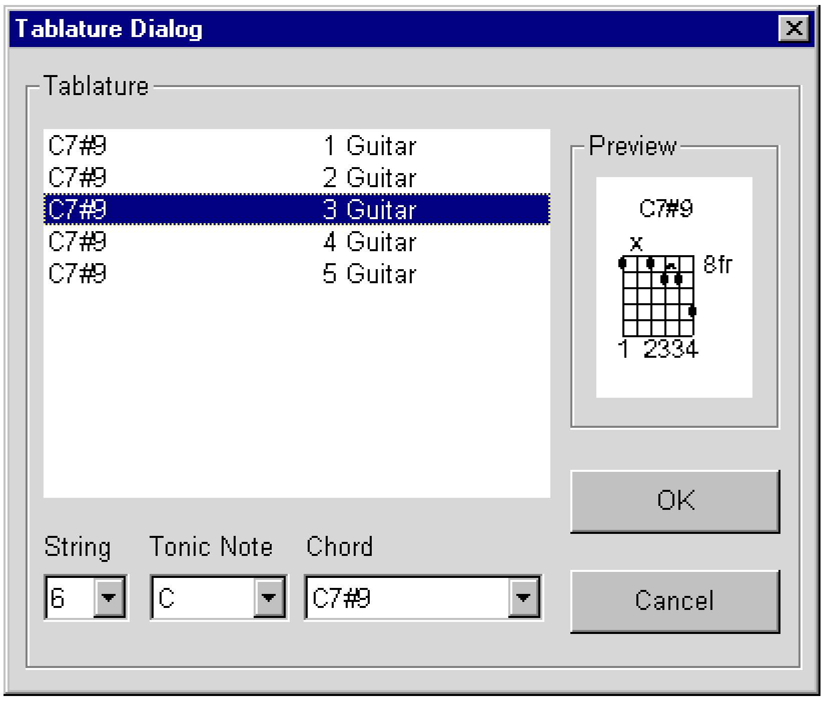
```
C;        1;  x32o1o; 0;    732717;
C;        2;  x3555x; 3500; 713337;
C;        3;  133211; 1608; 134211;
C;        4;  xxxo1o; 0;    777717;
C;        5;  332o1o; 0;    342717;
Cm;       1;  x3554x; 0;    713427;
Cm;       2;  133111; 1608; 134111;
Cm;       3;  x31o1x; 0;    741727;
Cm;       4;  xxx543; 0;    777321;
Cm;       5;  x35543; 2600; 713421;
C?07;     1;  x3424x; 0;    723147;
C?07;     2;  2x121x; 3507; 271317;
C?07;     3;  xx4545; 0;    771324;
C?07;     4;  xx1212; 0;    771324;
C?07;     5;  xx1212; 0007; 771324;
Cm7?15;   1;  x3434x; 0;    713247;
Cm7?15;   2;  2x221x; 0007; 273417;
```

The archive of positions (of text type), from which we started our research, needs a class that provides the methods to acquire the information. In order to be used with the planned interface this class has to contain the following methods (besides those for the research and for the reading):

- Counting and list of all the fretboards available for a certain nomenclature;
- Counting and list of all the nomenclatures for a certain tonic;
- Counting and list of all the tonics available for a certain instrument (distinguishable from the number of strings).

The reading method has thus to allocate and return an instance of the class containing the information on the fretboard that has been read.

The last class to be defined in order to solve this problem is a dialog-box of interface with the user that permits selecting the number of strings, the tonic and the nomenclature. According to the different choices it must provide a list of available positions and visualise, inside a frame made for this purpose, a preview in order to provide the user with a visual confirmation of the chosen fretboard.



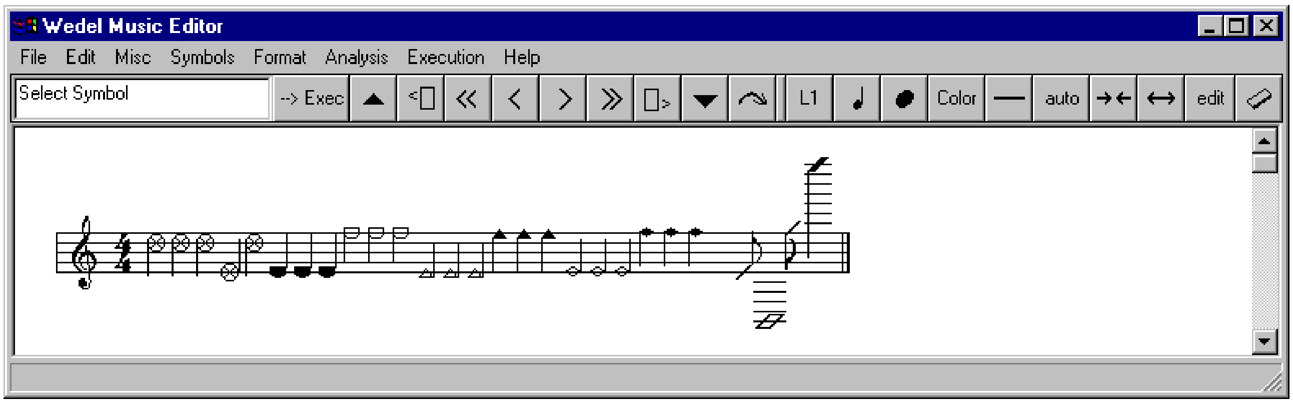
C7#9	1 Guitar
C7#9	2 Guitar
C7#9	3 Guitar
C7#9	4 Guitar
C7#9	5 Guitar

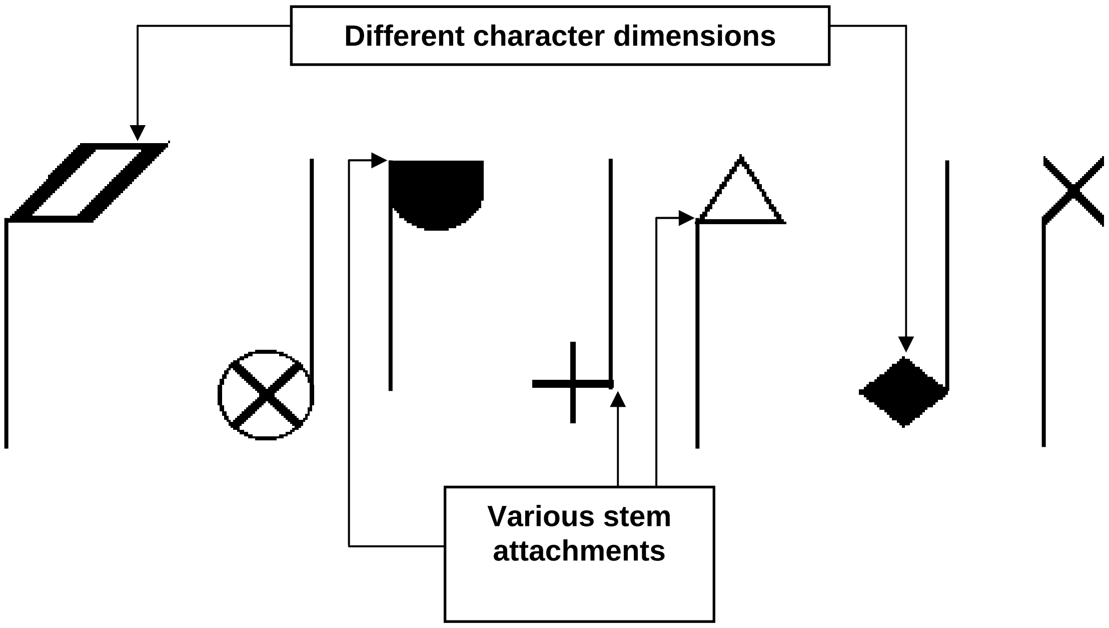
String: 6
Tonic Note: C
Chord: C7#9

The file of fretboard appears coded in a rather readable way and allows modifications to his content that are surely understandable. Anyway an applicative is necessary to the user in order to insert, in a visual way, personalised positions in the archive used by Moods, which must remain separated from the program (because of the rare use).
We realised, therefore, exploiting the same classes, an executable able to perform this function. Its user manual is described in the following.

5.1.20 Noteheads

The notehead problem doesn't concern only the guitar; new symbols are increasingly used in the musical notation (percussion instruments, special techniques …).
We can see how each notehead can have different shapes and sizes and each one follows particular laws for the stem attachment.

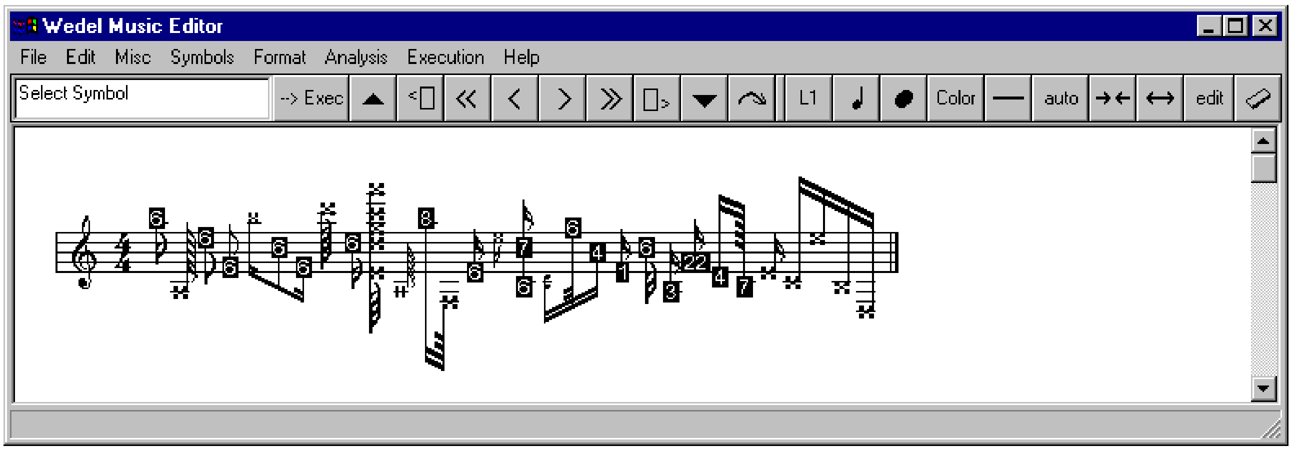




The first modification to be done to the class Nota is the inclusion of the attributes necessary to record the following information:

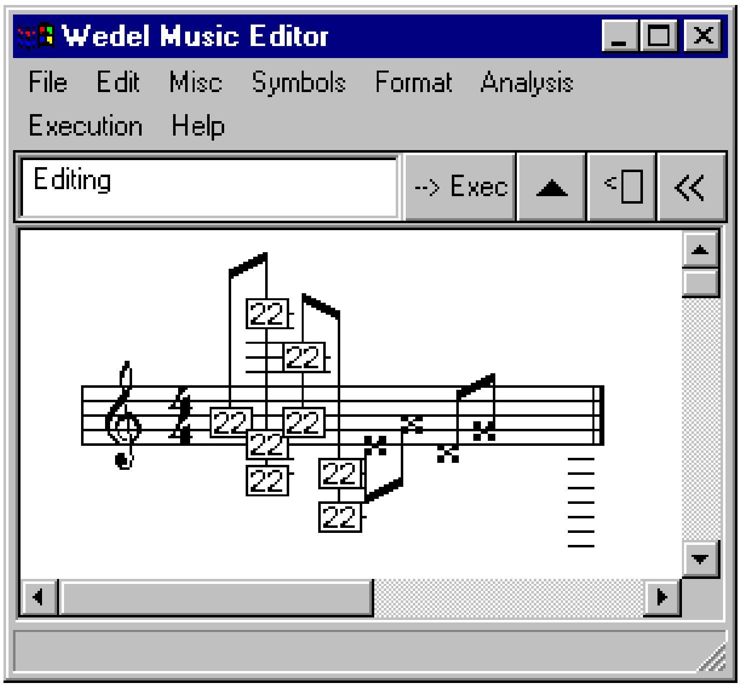
- type of head;
- if inside the alphanumerical heads one or two characters have to be written. The characters maintain also the ASCII character that has to be printed with the musical font in the case of other types of noteheads;
- information concerning the distance of the stem with respect to the centre of the notehead.

The most difficult thing in this part of the work is the revision of the code for the drawing of the note on the score that, previously, was the task of the subclasses of Nota as NBreve, NSemiBreve, NMinima…
In fact the note must find out the symbol to be used for its drawing on the basis of the kind of notehead and the assigned duration; this "intelligent" choice is supported by the Milla language that indicates the ASCII code of the symbol in the musical font (e.g: 165 for the symbol of the notehead with a half moon shape). Moreover, Milla owes, for each type of note head, all the necessary information to determine the position of the stem with respect to the centre of the note.
Because the size of the notehead and the origin of the stem are no more constant, each method that depends on those measures endures variations before being generalised; the measurement of the region of space that contains the note and the beaming applied to beams are two among the procedures that we have corrected.



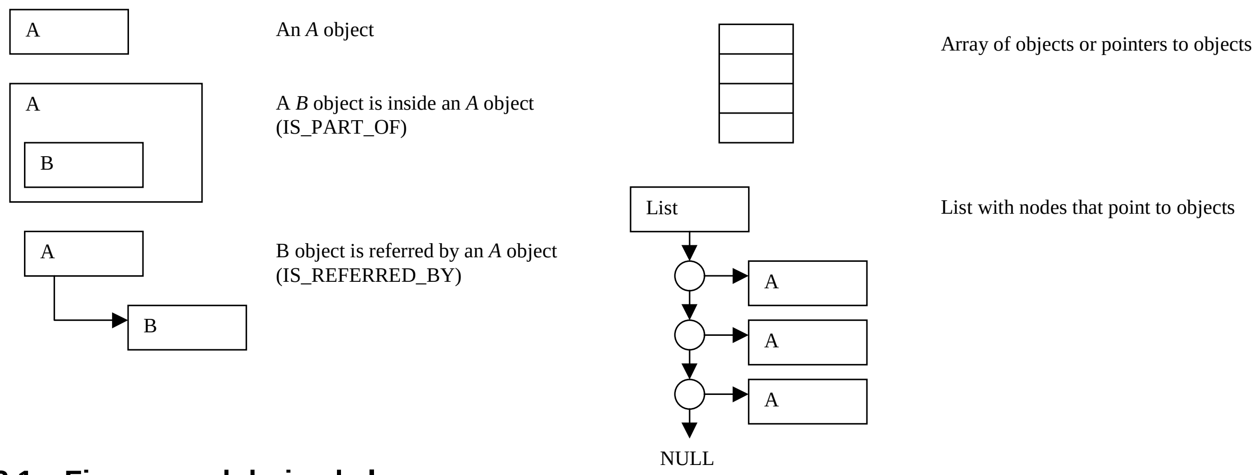
The modifications concerning the insertion of a note, the acquirement of the character, the positioning of the stem and the calculation of its length are extended to the class Accordo.
The methods for printing are similar to those for drawing and thus lead to the same modifications. The saving operation is enlarged in order to record the added information; the attention is pointed in particular to the Accordo class because is fundamental not to duplicate information.

Previously in the construction of the object note the “bounding box” of the note was calculated with constants measured on the classical notehead; now the size varies and nothing is known until the code is find by Milla through the call in **Adjust(…)**. After the acquirement of the code we can obtain all the measures of the character consulting the table (MUSICA__.tlb). By these conditions it is possible to set the frame of the notehead (that can be useful for other procedures of the same method **Adjust(…)**).



5.2 WEDEL Object Oriented Model of Music - some design issues

In this section the principal problems related to the design of the classes defined in the previous section are faced; in particular the most significant attributes and methods of the classes and the type of relation implemented by IS_PART_OF and IS_REFERRED_BY will be discussed (e.g. 1/1, 1/N). Not all the classes will be considered: those that implement only the draw of a symbol, and thus contain only the methods for drawing and saving, will not be mentioned. In the same way, if a relation IS_PART_OF or IS_REFERRED_BY is not discussed, it implies that it corresponds to a relation 1/1. Unless it is differently specified the realisation of each class introduced has to be intended the same both on the main score editor and on the score editor.
The symbols of the *object diagram* presented in this chapter is described in the following figure:


A
An *A* object
A
B
A *B* object is inside an *A* object (IS_PART_OF)
A
B
B object is referred by an *A* object (IS_REFERRED_BY)
Array of objects or pointers to objects
List
List with nodes that point to objects
A
A
A
NULL

5.2.1 Figures and derived classes

The *Figura* class generalises the concepts of note and rest. Like almost all the classes introduced, *Figura*, and the classes derived from it, must be able to represent itself on the screen. For each note the most significant information is the height and the duration (or value). Concerning the duration, it is determined for each class son of *Nota*, since each of them represent a precise value; on the other hand, to let each note “recognise” its own height the class *Nota* has been endowed with an apposite attribute (*Altezza*, of an integer type). Such number is set 0 for the notes placed on the inferior line of the staff (in the tremble cleft it corresponds to a Mi), and grows of a unit for each superior position (e.g. in the tremble cleft the Sol on the second line has height 2); it can be also negative. From the musical point of view, a rest is identified only by a duration and does not have a height, but since in the polyphonic music it is necessary to place the rest on different heights, it has been decided to endow *Pausa* too with the attribute *Altezza*. As a result the whole class *Figura* has been endowed with the attribute *Altezza*, and the classes derived from it inherit it.
Other attributes of Figura are the pointers to the classes that are related IS_REFERRED_BY with this class; each one of these links corresponds to a relation 1/1 and exploits the polymorphism of the connected classes; at attributes level this means a pointer for each connected class. Considering all the classes in relation IS_REFERRED_BY with *Nota* and *Figura*, it is possible to formulate the diagram presented in fig. 1, that represents an instance of NCroma. The example could have been formulated in the same manner with any other class son of *Nota*, but *Accordo (chord)*.

In the analysis phase, together with notes and rests, the List class has been introduced. This must offer basic functionalities, as:

- Insertion of an element
- Deletion of an element
- Search of an element
- Scanning of an element
- Deletion of the list and, eventual deallocation of the pointed elements.

One of the classes derived from List is *ListaFigure*; the relation IS_REFERRED-BY of *Figura* with this class is to be considered a relation 1/N. The classes *Accordo* and *GruppoNote* are lists of figures. Besides the typical characteristics of the lists they have also to present the following methods:

- Control on the type of figure that is to be inserted: e.g. *Accordo* accepts only notes and all of the same kind.
- Positioning of figures in the list.
- Positioning of symbols around the figure.
- Drawing the figure and of the symbols referred to it.

For these classes the attribute *Altezza*, inherited from Figura, is not important. Besides being a list, *Accordo* is different from *Nota* also for the following features:

- Each note that composes a chord must be able to represent its own augmentation dot, its own accidentals and fingering. Therefore *Accordo*, even if inherits from *Nota* the pointers to *PuntoValore*, *AlterazioneComposta* and *Diteggiato*, does not use them, but refers to the pointers included in each note that it contains. As regards other symbols it refers to the inherited pointers so that it will be possible to place on a chord only one dynamic text, only one fermata and so on.
- With respect to *Nota*, *Accordo* has to foresee another attribute which consists in a pointer to an eventual arpeggio symbol. This sign must be placed on the left of the chord (if there are accidentals, on the left of these).

All these specific features are represented in figure 2.

Differently from *Accordo*, a instance of *GruppoNote* can contain notes of a different kind; moreover it can contain also objects *Ancoraggio* and *Pausa*. To this end *GruppoNote* has to foresee the design of broken bars. *GruppoNote* inherits from *Figura* the pointers to the various symbols but does not use them. In figure 3 the scheme of a possible instance of this class is represented; in order not to overload the picture, no object connected to the figures of the group has been drawn.

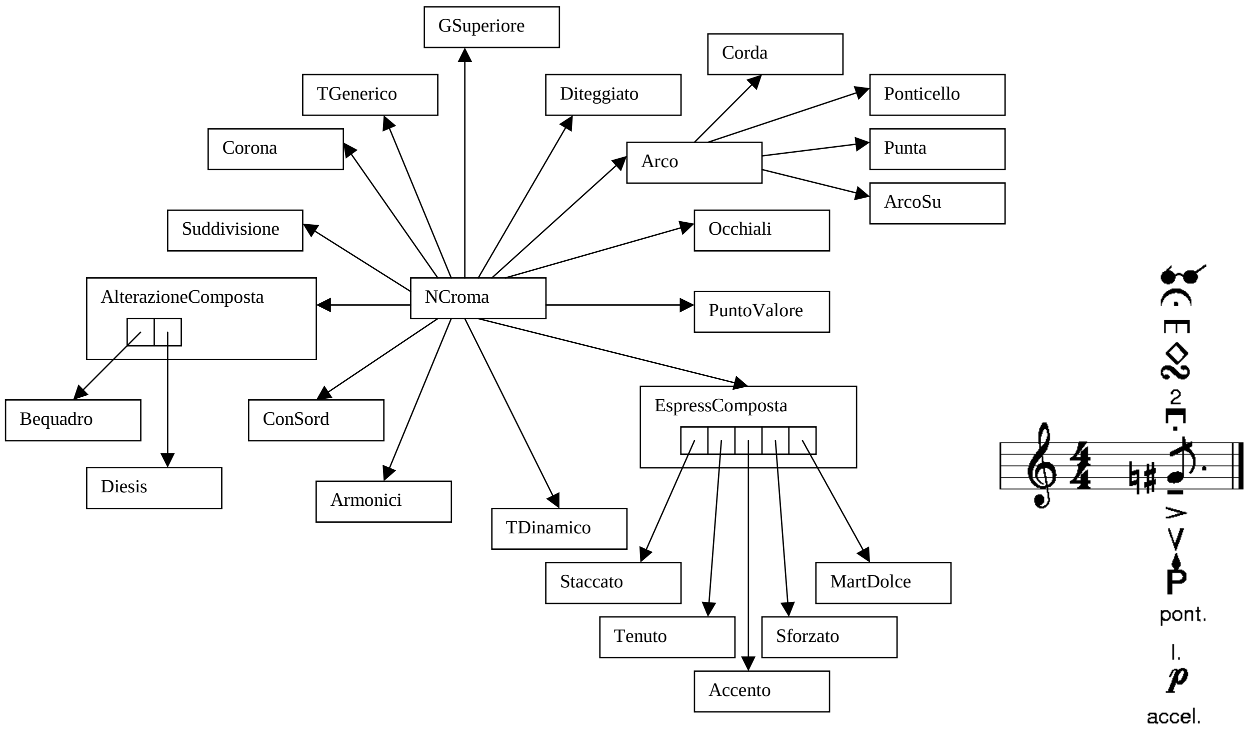


Fig. 1 - Example of note

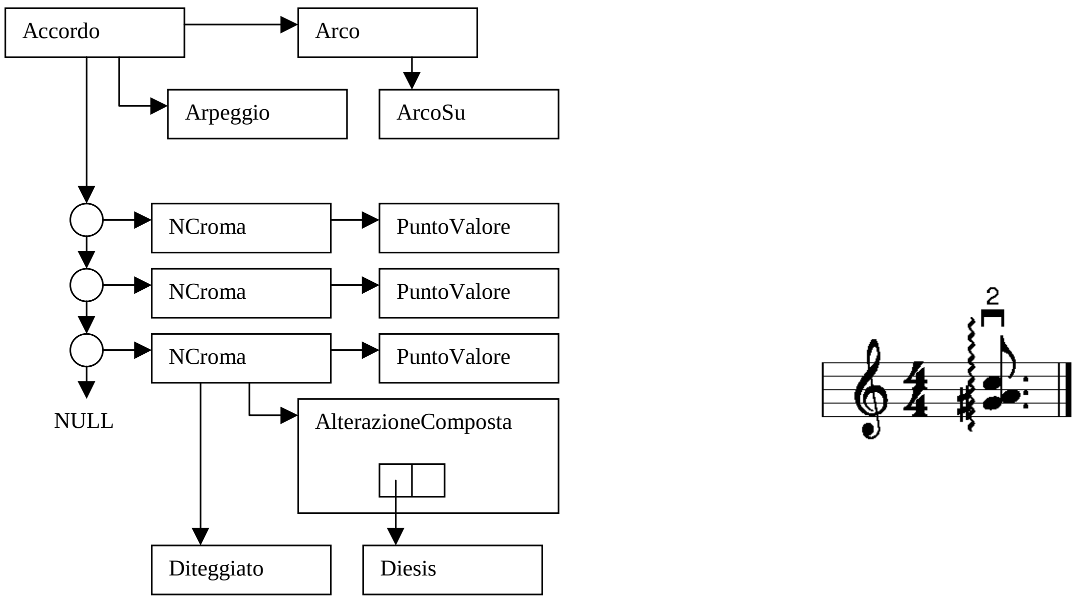


Fig. 2 - Example of chord

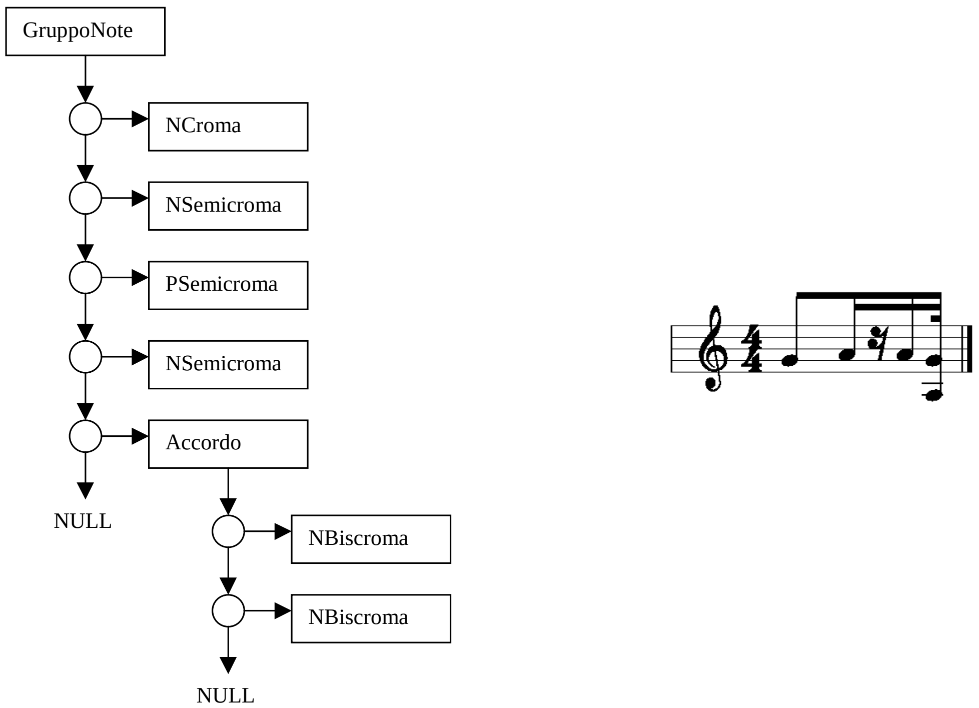


Fig. 3 - Example of beam

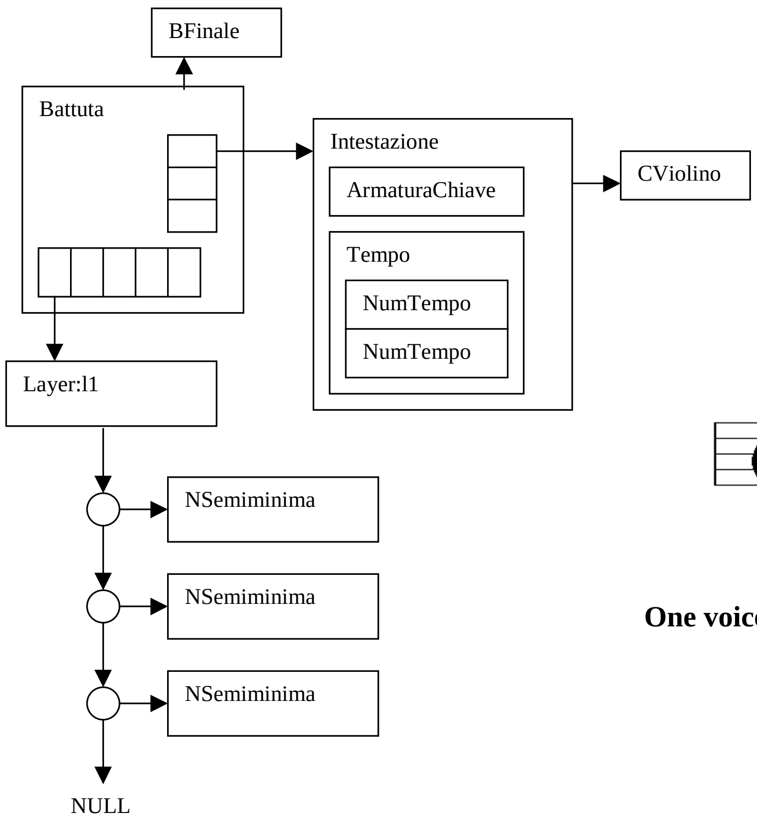
5.2.2 Measure

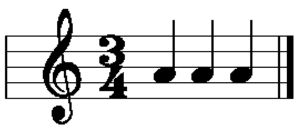
Both from a musical and a graphic point of view the measure can be considered a container of musical figures (notes, chords, rests and groups of notes), that must include not only a unique melodic line but must also consent a polyphonic writing. Moreover for some instruments (e.g piano, organ, arpa) a measure is composed of two or three staffs and a melodic line can start in a staff and end in another.
For this reason as already mentioned in the analysis a *Staff* attribute has been added to figura, staff 0 is the first staff (from top) and is the default, staff 1 is the second and staff 2 is the third. So an attribute of battuta is the number of staffs. An array of three pointers to Intestazione is used to store the heading information of the measure, pointers in this case are used to minimise the storage needed for a Battuta object, since many measures have only one staff.
The heading contains always a *Chiave* symbol that indicates the clef valid for the whole measure staff. In order to make easier the management of the heading the clef is represented always at the beginning of the measure. Possessing the information related to the clef, the class Battuta is able to obtain from the attribute

Altezza of each note the pitch of the note. The class *Tempo*, a mandatory part of *Intestazione* and consequently of *Battuta*, has a double relation IS_PART_OF with *NumTempo*. This class must foresee almost two arguments:

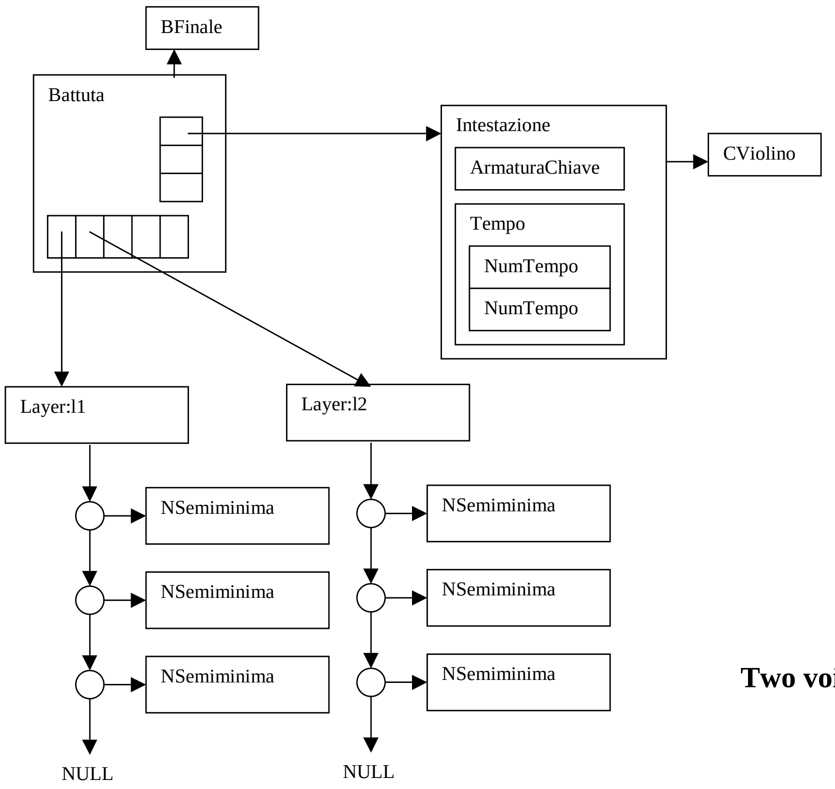
- Pointer to a string of characters, that represents the text.
- Integer number represented by a text.
- Integer sequence separated by ‘+’ (e.g. “3+2”)

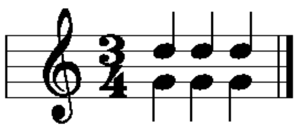
It must in addition present the methods for the conversions string -> number and number ->string.
Coming back to *Tempo*, this class has to draw not only the two numerical texts but also the two graphic symbols C and C dashed.
An array of eight pointers to Layer object is used to store the layers of the measure, for a polyphonic score (one staff) all the figures of the layers will belong to the same staff, for multi-staff measures some layer will have figures on a staff and others on other staffs and some could have figures on a staff and others on other staff.
Another object that characterises the *Battuta* class is *Barra*. In the analysis phase it has been already explained the reason (exploitation of the polymorphism) that has led to consider *Barra* IS_REFERRED_BY *Battuta* even if the bar must always be present. Consequently in this case the relation IS_REFERRED_BY does not indicates – as usual – an option but an obligation. A particular case is represented by the BInizioFine class which consists of the bar that concludes a refrain and that that begins it. The critical case verifies when the bar is at the end of a musical line since it must split in two parts: one closes the measure of the staff while the other must be placed inside the first measure of the successive staff. For this reason the class Measure has to include a method for the placement of an additional bar on the right of its own heading. Each kind of bar must be predisposed to be extended in order to reach the bottom of the score (an implied reference to the main score lectern is made). It is in any case evident that the setting of the length of a bar is a task of the object *Battuta*.
Some object diagrams for the *Battuta* class is reported in the following figures. In this scheme not all the classes in a relation IS_REFERRED_BY with *Battuta* have not been inserted; Barra is represented because it is always present in the measure.



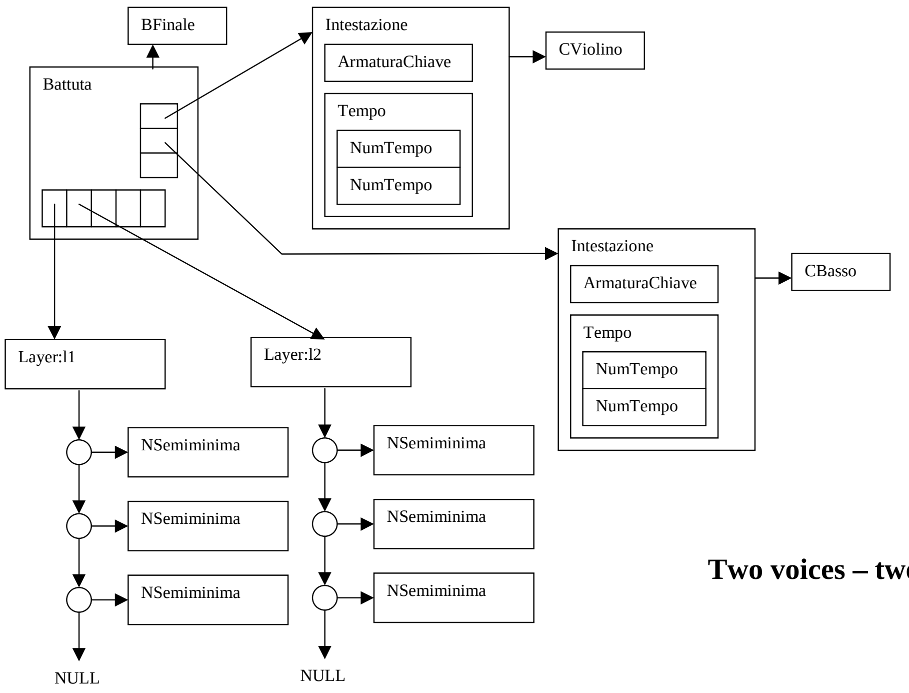


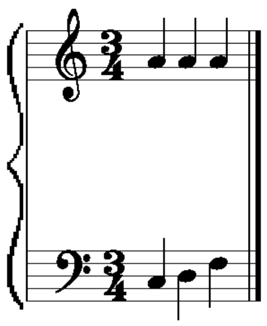
One voice – one staff measure





Two voices – one staff measure





Two voices – two staffs measure

5.2.3 IntEsteso and derived classes

The relation *Figura* IS_REFERRED_BY *IntEsteso* corresponds to a 1/2 relation and does not implies an option but an obligation. It can be translated in two attributes of *IntEsteso* that represent the pointer to the figure on which the interval symbol starts and the pointer to the figure on which it ends. The class that manages the interval symbols, in order to place each symbol in a right way, will have to use these two pointers and skim the staff to avoid the superposition of this symbol on those already existing. As already said, the *Intervallo* class differs from *IntEsteso* only because it cannot begin or end on rests or anchor-points. For this reason, an *Intervallo* method has to control the kind of pointers that are given to the class and refuse those that are not compatible with the class. Concerning the example diagram the one concerning the class *Spartito* is to be compared; from this diagram is clear that the relations *Figura* IS_REFERRED_BY *ListaFigure* and *Figura* IS_REFERRED_BY *IntEsteso* origin a *data sharing.*

From the graphic point of view, the interval symbols have to be represented above or below the staff, the only exception is *LegatValore* that can be placed inside the line; moreover some symbols have to be represented in two different versions:

Legatura: When it is drawn above the line it appears like a bow; when it is drawn below it appears like a bow capsized. The subclass LegatValore is similar; when it is inside the staff it can be directed upward or downward.

LegatQuadra: it is analogue to Legatura.

ModifOttava: If it is placed above the line it refers to the “octave/15th above”, otherwise “octave/15th below”. The design is slightly different in the two cases: in the former the dashed line is above with respect to the 8va symbol, whereas in the latter case it is below.

The remaining symbols do not change according to their position on the staff. To be highlighted is that the *Forcella* must be able to be represented in the two versions of *crescendo* and *diminuendo* and that *CambioRit* must integrate the design with the number of refrain.

The LegatQuadra class must be separately considered. It is particularly important because it permit realising the tuplets and big groups in general. Each interval symbol must foresee to be designed in many parts; this specification is necessary when a symbol (e.g. a slur) begins on a staff and ends on the successive (on a score): practically two slurs are designed. This considered, it is obvious that for the class InEsteso one position and one dimension are nor enough (like all the classes derived from DrawObject); it has been therefore decided to endow IntEsteso of methods suitable for managing inside the class a list of positions and dimensions. The necessity of breaking the interval symbols is present typically on the score editor which has the possibility of representing many lines on the same page; in main score editor it is instead represented only one line for each page.

5.2.4 Score

The *Spartito* class represents a musical part with a unique line. It contains, therefore, the *ListaBattute* class (link IS_PART_OF, relation 1/1) derived from *Lista*. In analogy with *ListaFigure*, the link *Battuta* IS_REFERRED_BY *ListaBattute* represents a 1/N relation; the link *IntEsteso* with *ListaIntEst* has the same meaning. This last class too has a connection IS_PART_OF (relation 1/1) with *Spartito*. The figure highlights, by means of dotted lines, the connections that exist between the figures contained in the measures of *ListaBattute* and the symbols belonging to *ListaIntEst*. As already said, the double link IS_REFERRED_BY starting from *Figura* is used for the data sharing among the objects; this kind of situations are delicate especially concerning deletions. The *Spartito* class has thus the task of “superintend” the deletions that occur inside the measures, taking all the necessary measures on *ListaIntEst*: for example when a note is deleted also the eventual slur that might begin on it must be deleted. In the figure, for the sake of simplicity, only one of the two lists that compose each measure has been represented. The order in which the interval symbols appear in ListaIntEst is meaningless. It is appropriate to remark that separated lists for slurs, forcelle and other categories of symbols are not foreseen: the list of intervals is unique for each score.

Spartito must, first of all, present the methods for the disposition of the musical symbols that it contains, limiting the choice to those really represented on the screen. This activity must be performed in three phases consecutive:

1. Assignment of the position and of the length to each measure belonging to *ListaBat*. After this operation only the symbols inside the measures are placed.
2. Allocation of the symbols belonging to *ListaIntEst*. The symbols of this category must be placed outside the staff, above or below the symbols already placed in the previous phase. The only exception is represented by the tie which can be inserted inside the staff because is placed at the same height of the notes it connects.
3. Allocation of the external symbols with respect to the measure (e.g. the indication of movement) above or below all the other sign previously placed.

The behaviour of *Spartito* in these activities varies according to what the lectern visualises either a main score or a single part. In the first case, corresponding usually to the lectern MASE/MASAE, the class Spartito must act according to the settings deriving from the *Partitura* class; for example it is *Partitura* who must indicate to *Spartito* the position and the length of each measure in order to maintain the vertical alignment of the measures and of the notes contained inside them. In the case in which only one part appears on the screen (DLIOO/PDLOO), it is *Spartito* that has to divide the measures among the staffs present in the page represented; it can be convenient to realise this function as a method of the *ListaBattute* class and let it be recalled by *Spartito*. Concerning the placement of the slurs and the other signs with analogue behaviour, *Spartito* must present a method that analyses the list of measures and the layers of figures contained in each measure; for each figure the method must control if some sign of *ListaInt* is extended on it and consequently place it. If on the page more than one line is present (tha case of DLIOO/PDLOO), the above mentioned method has the task to eventually separate in many parts the interval symbols.

The class *Pentagramma* is related IS_PART_OF with *Spartito*. In the case of the main score editor Spartito needs up to three staff for each page, while in the case of score editor, it must dispose of more than one staff. For this reason an array of objects Pentagramma has been inserted among the attributes of Spartito (relation 1/N). In the case of MASE/MASAE the object *Spartito* must be implemented by Partitura which places the unique staff in its same position while in the other case it is implemented directly by the object *LiooWindow*, which sets also the number of staffs.

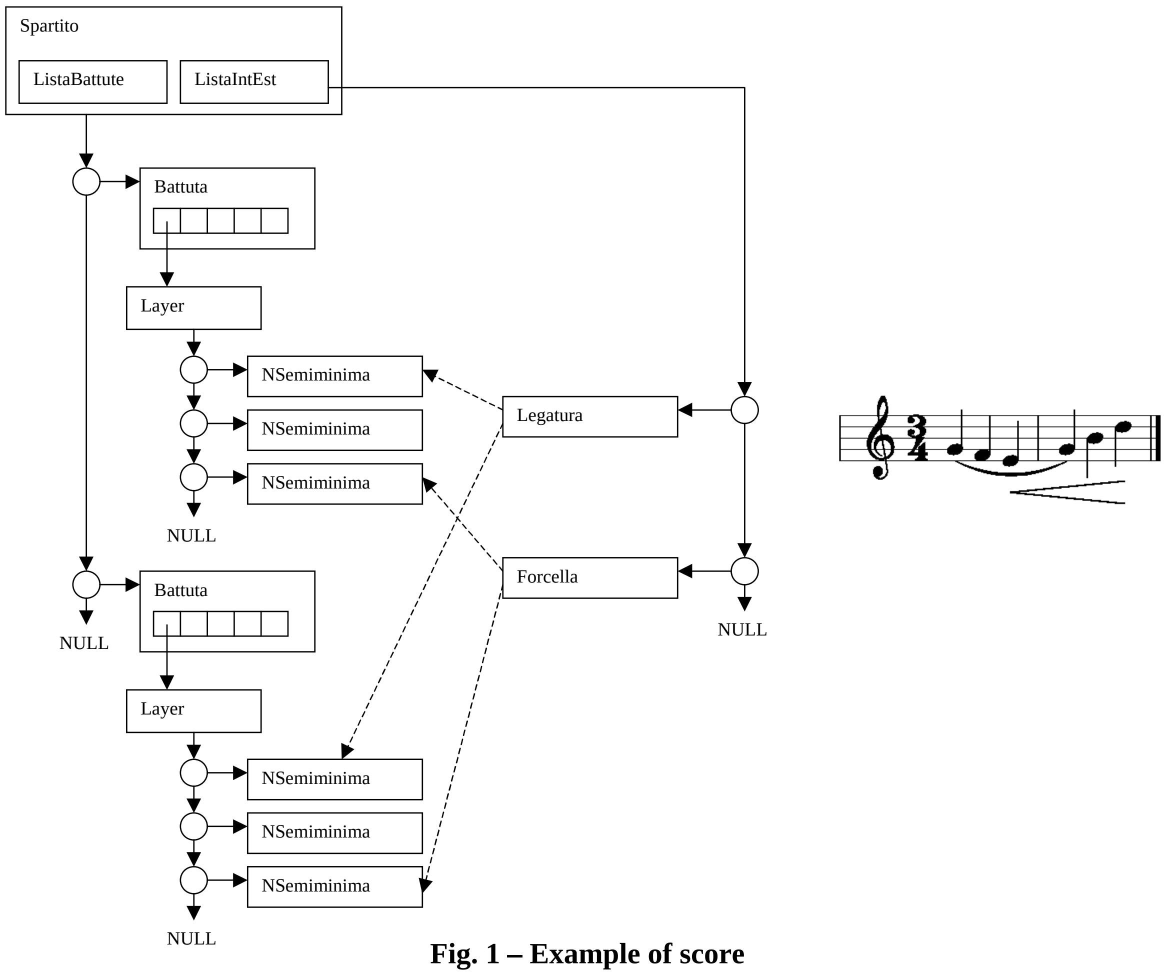
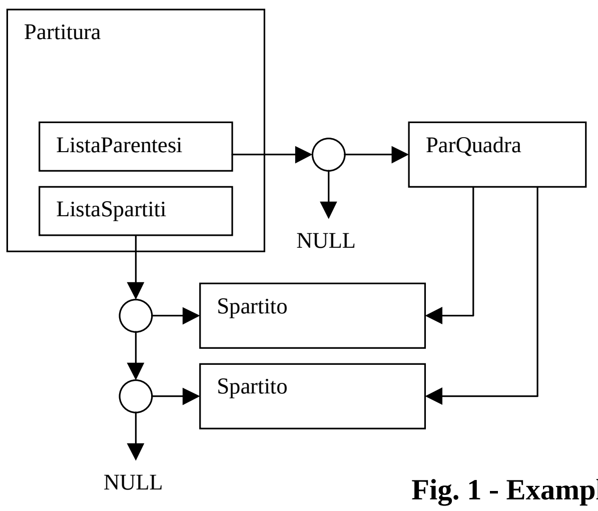


Fig. 1 – Example of score

5.2.5 Main Score

The *Partitura* class is instanced exclusively on the lectern MASE/MASAE since the DLIOO/PDLOO is limited to the single part. An instance of *Partitura* always includes an object *ListaSpartiti*, that, in analogy with the other classes derived from *Lista* can contain an arbitrary number of objects *Spartito* (relation 1/N). An example of instance of Partitura is represented in figure 1.
The most complex function to be performed by *Partitura* and *ListaSpartiti* is the correct superposition of the measures of the line and of the figures inside each measure. Since for each measure a fixed number of figure is maintained in the two layers, it has been decided to maintain the same concept also for *Partitura*.

Partitura has also the task of the management of braces trought class *ListaParentesi* whose main task is to position the braces in particular it has to establish the level of the brace to avoid overwriting.



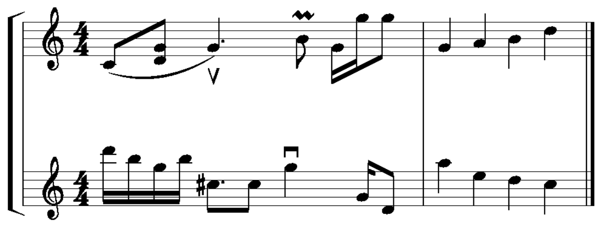


Fig. 1 - Example of main score

5.2.6 Different Indications

5.2.6.1 Ornaments

The classes derived from *Abbellimento* belong to the category of those that have as unique aim the representation of the corresponding symbol, moreover some ornaments can have accidentals (above and/or below). The only exceptions are *Trillo*, *Glissato* and *Tremolo*. The trill is managed by two classes *Trillo* and *TrilloOnda*, the first has to drawn the "***tr***" symbol eventually with a accidental while the second have to draw the tr symbol followed by some waves up to a certain point. The object *Tremolo* and *Glissato* represents an exception because while all the ornaments are aligned on the note they refer to, tremolo and glissato are placed to the heads of two consecutive notes. For this reason this object must be managed from the Battuta class instead of Figura. Class *Glissato* has an attribute indicating the type of draw that can be as a straight or waved line and class *Tremolo* has an attribute indicating the number of bars to be drawn.

5.2.6.2 Agogic and dynamic indications

The class *TMovimento* must offer the possibility to insert texts composed by the user as for example "Allegro ma non troppo", or "Moderato cantabile molto espressivo". The same specification can be made for the texts *TDinamico, TGenerico* and *Annotazione*.
As regards the class *Scansione* it has been decided that the vertical bars must be represented all together above the bar that delimits two measures. Also the other classes that are related IS_REFERRED_BY with Battuta (all relations 1/1) must foresee the vertical alignment on the bar; an exception is represented by *NumGrande* because the objects like *NumPausa* and *NumUguale* are to be placed in the middle of the measure. In these classes is often made a recall to the class *TNumerico* because three links IS_PART_OF start from it. It is not a data sharing because each object *TNumerico* is used by only another object.

5.2.6.3 Expression signs and classes referred to Figura

For the classes derived from *Espressione* a unique peculiarity is to underline: *Accento* and *AccentoForte* are drawn in a different way according as they are placed above or below the note. In order to allow each symbol of this category to know if it is superior or inferior with respect to the line, is appropriate to introduce into the class Indicazione a boolean attribute. This consideration is to be extended also to the classes *Corona, TDinamico, PrFiato, Occhiali, TGenerico* and *Annotazione* that are derived from *Indicazione*. Among these *Corona* is the only one that has a different direction with respect to the position in the staff. The use of the object *Annotazione* is reserved for the director and the archivist, thus such symbol does not appear on the orchestral lecterns.

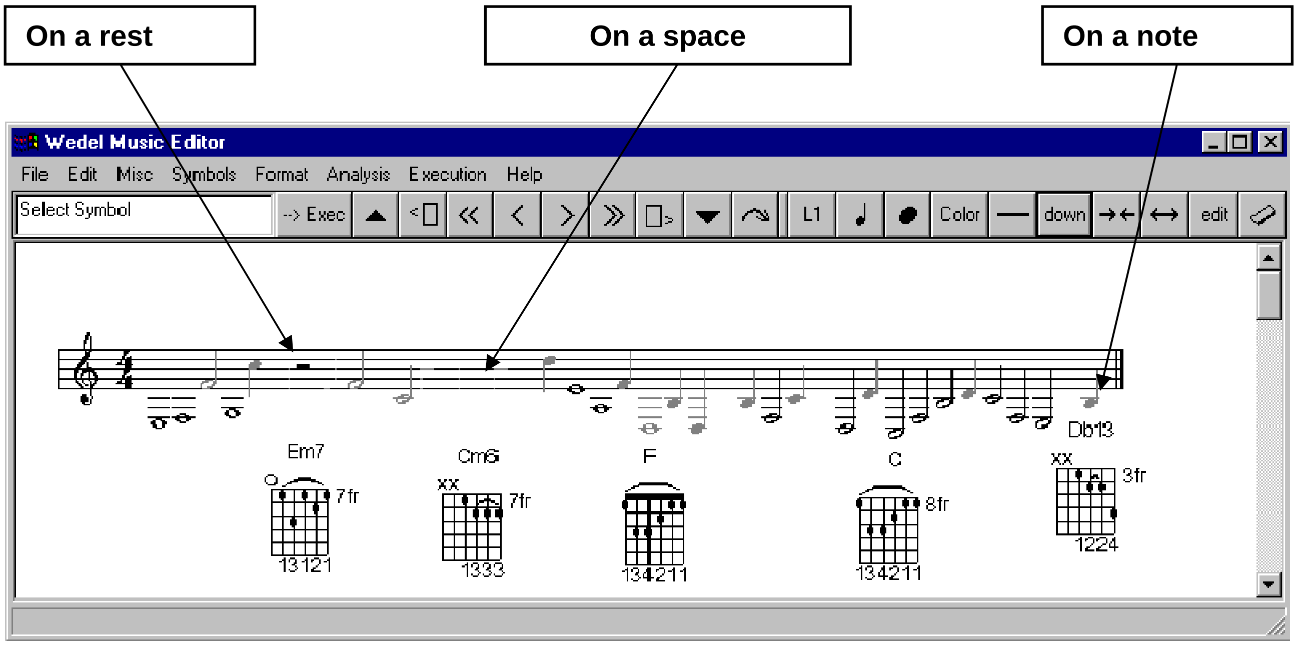
5.2.6.4 Instrumental Indications

The Violino class is used for managing the representation of symbols typical of bow instruments. Differently from other subclasses of *Indicazione* it does not represent a unique symbol but it can arrive to the representation of four symbol at one time if the note is to be played with the bow. This case is exemplified in figure ?? with the representation of a possible instance of the *Arco* class, son of *Violino*. Since the symbols

related to violins can refer both to notes and chords is appropriate foreseeing that the *Corda* class can foresee multiple indications (till 4 that is the number of the strings of the bow instruments).
The object *Timpano*, as already said, consists of a text in a frame that can be preceded by an arrow.

5.2.7 Fretboard

The class DitaCorde is inherited by Strumento because the fretboard is an instrumental indication that shows how to play the instrument (like the pizzicato); it is referred by Figura because can be placed in any position inside a measure (on a note, a rest, an empty space).



The development of DitaCorde is, as already stated, connected with the coding style of the archive information: the reading difficulty is minimised.
As a matter of fact the memorisation of the positions concerning the keys to be pushed and the fingers to be used are not optimised for what concerns the space or the representation on the screen. Two strings containing many characters, one for each string of the instrument, appear in order to find the keys and the fingers.
Up to 8 bits are wasted for a data that actually can present about ten possible values. The order on the string, that represents its position in the fretboard from left to right, doesn't correspond to the real order of the instrument's strings in the music (it is inverted). Also the barré doesn't present an appropriate resource because even in this case the two necessary values (numbers from 1 to 9) are inserted with their own ASCII code on a string of two characters.

Examples of fretboards coded in the archive.

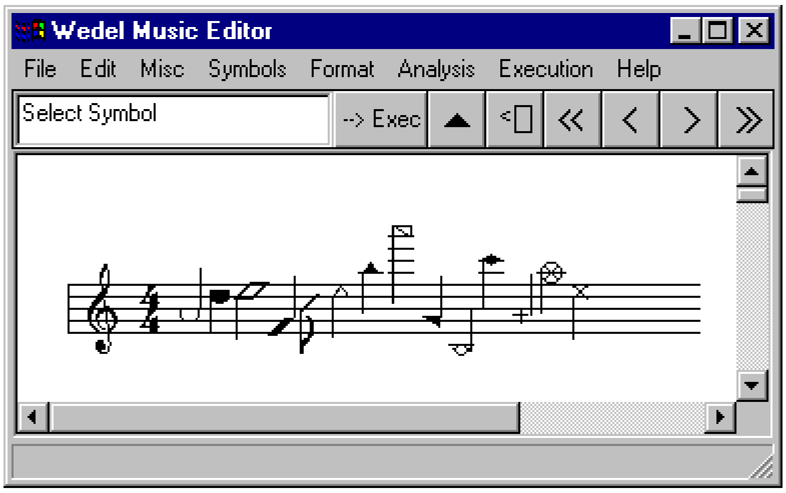
```
C;          6;      1;       x32o1o;    0;      732717;     “Guitar”
F7;         6;      1;       131211;    1600;   131211;     “Guitar”
```

Ditafile is an independent class, but it uses DitaCorde because it produces instances of this class in the reading moment: it is in a “use” relationship.
Tabdialog has only the function of interface with the user; it has been realised with DialogEd and contains a reference to DitaFile in order to use the information of the fretboard archive.

5.2.8 Noteheads

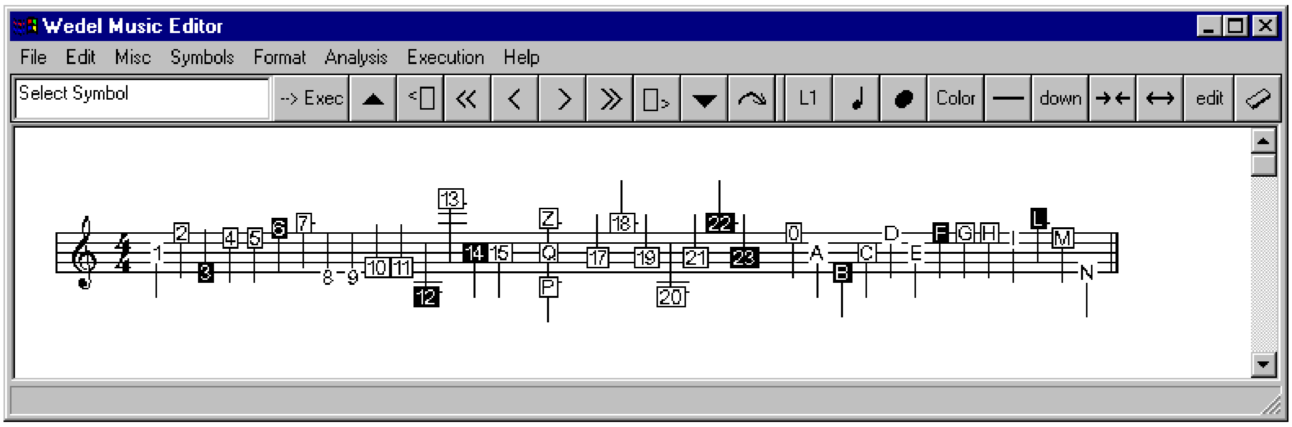
In the class Nota a variable of the NoteHead_ID kind, defined through construct enum in the main header (lioo.hpp), has been added; this statement involves all the types of noteheads requested by the specifications.

```
enum NoteHead_ID {
  HEAD_CLASSIC,
  HEAD_ALPHANUM,
  HEAD_ALPHANUM_SQUARE,
  HEAD_ALPHANUM_REVERSE,
  HEAD_CIRCLEX,
  HEAD_CLUSTER,
  HEAD_X,
  HEAD_CROIX,
  HEAD_DDIESIS,
  HEAD_DIAMOND,
  HEAD_TRIANG,
  HEAD_TRIANG_UP,
  HEAD_TRIANG_DOWN,
  HEAD_TRIANG_LEFT,
  HEAD_TRIANG_RIGHT,
  HEAD_TRIANG_ROUND,
  HEAD_MOON,
  HEAD_PLUS,
  HEAD_RHYTHMIC,
  HEAD_SQUARE,
};
```

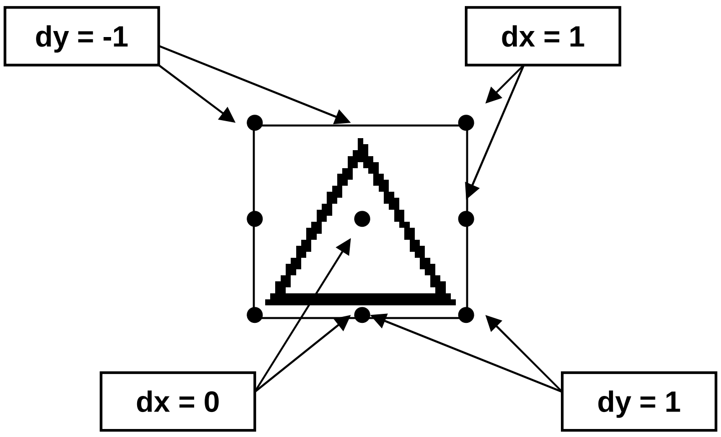


It is of course necessary the insertion, among the attributes, of a string of two unsigned chars with a double use: in the case of a notehead of the alphanumerical type it contains the characters to be written inside itself; otherwise contains the code of the character (chosen by Milla) inside the musical font.

With the same procedure the two integer numbers that contain the horizontal and vertical distances of the

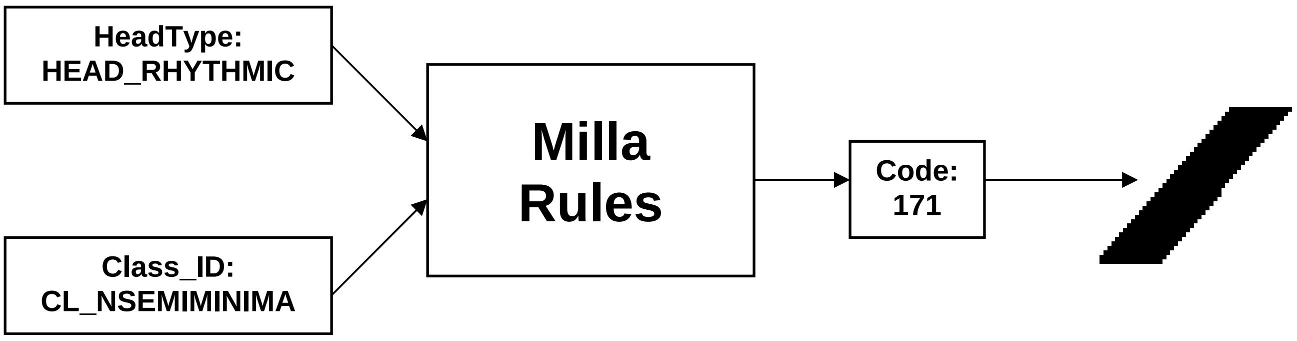

Wedel Music Editor
File Edit Misc Symbols Format Analysis Execution Help
Select Symbol
--> Exec
L1
Color
down
edit
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 Z Q P 17 18 19 20 21 22 23 0 A B C D E F G H I L M N

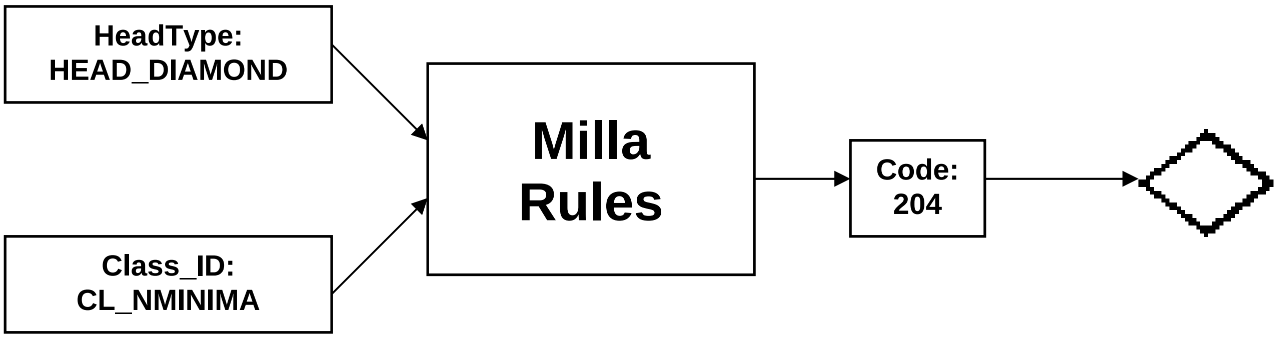
stem attachment from the notehead are foreseen (also these determined by Milla); these values are normalised with respect to the size of the notehead and can assume three values (-1,0,1). In this way all the 9 possible starting points of the stem are obtained.



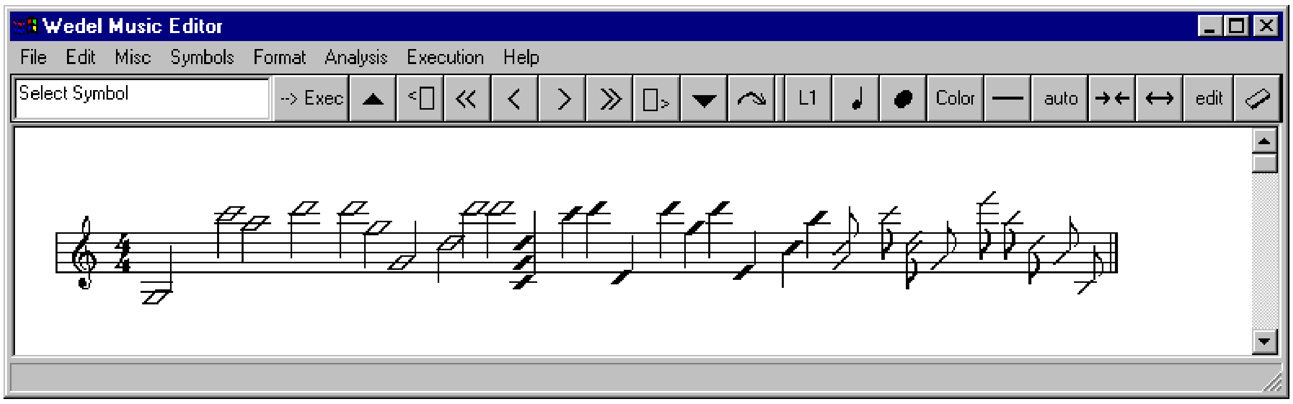
From the normalised distances and from the information on the size of the notehead it is easy to reconstruct the absolute position of the starting point of the stem; we have implemented appropriate methods in order to execute this calculation because it is required by many drawing procedures. In order to avoid, as much as possible, the code duplication we subdivided the different phases of the Draw method in several functions obtaining more generic methods (DrawTesta, DrawTagli).

The choice of Milla is excellent because it permits the combination of multiple conditions in order to determine the code of the note: we noticed that some type of noteheads don't distinguish the symbol to be used according with the duration whereas others have up to three symbols for being represented. If in addition we consider the cases in which some noteheads, positioned in "small note", can require an additional symbol with a smaller size, the setting of all these laws in a table results to be inappropriate, also for the excessive statics. One of the future developments of Milla is the modification of the formatting in real time that allows the user to modify, in any moment, the symbol associated to a notehead, using conditions on the musical text in its completeness. We didn't want to call Milla for each drawing so the character is determined from the call of the Adjust method (that is performed a bit less often than Draw) and memorised in the appropriate string of characters.
This procedure doesn't concern the notes with alphanumerical noteheads that use the string in order to record the code set by the user.
Let's now see how each note acquires its symbol taking into account different factors.





The code inside the Draw method has been lighted of all the duplication in the different cases of Nota/Notina because the design of the notehead is not different in terms of code (we have just to print a character); it still remains the drawing of the stem that maintains this duplication (concerning in particular its length) which is anyway reduced and in the future will be entrusted to Milla.



5.3 WEDEL Object Oriented Model of Music and its Navigation

Navigation in the object oriented music model is essentially done using the methods defined for class *Lista*, they permit to iterate throught the list of objects (forward and backward) or to retrive one by position.

Have to be noted that generally objects don't have a pointer to the parent, so a figure doesn't know in which measure it is or a measure in what score it is.

In some classes have been added methods to facilitate the navigation, for example *ListaFigure* provides methods to iterate throught the list flattening beams so to obtain the figures in a list as it whould be without beams.

Another feature that simplifies the navigation regards the interval symbols, some specific methods have been defined in *Figura* to retrive the interval symbols that are starting from the figure, are over the figure or are ending to the figure.

Another problem related to the navigation is the retrieving of a specific symbol in the score, this problem is analysed in the following section.

5.3.1 Selection description

Problem analysis
The selection proceeding is useful to identify an object present on the screen starting from the position of the mouse cursor on the graphic screen. The identification of an object can be performed through:

1. **a physic pointer**: that is to say through the *address* of the object in the central memory.
2. **a logic pointer**: that is to say through a *path*, an array of numerical codes that identifies the objects on the screen; each entry of the array corresponds to a different level of the structure of the main score (e.g. the scores level, the measures level…).

The first method is of easy employment, but it limits the validity of the selection's result to a unique LIOO process in execution, so that it reveals unsuited to the communication of the selection among many processes. For this reason the studying of a *symbolic* identification of each object, coinciding in each LIOO process in execution, has been necessary. In the first case the selection's result is nothing else than a pointer to a memory address, whereas in the second case, the one that has been chosen, is a *SymPath* structure defined in the following:

```
typedef struct{
  NumCode Array [PATH_LENGTH];
  short Level;
} SymPath;
```

The information is totally contained in the *Array* of a NumCode kind (defined as unsigned short), whose entries indicate a different level in the path; the levels have the following parameters:

- **`Array [SPARTITO_PATH_LEVEL]`** it shows the score numerical code to which the object belongs.
- **`Array [LISTA_PATH_LEVEL]`** its value is 1 if the object belongs to the list ListaBattute associated to the score, 2 if it has to be searched inside, between the intervals (that is to say using the list ListaIntEst associated to the score).
- **`Array [BATTUTA_PATH_LEVEL]`** it shows the numerical code of the measure inside the measures' list.
- **`Array [LAYER_PATH_LEVEL]`** it indicates the layer (1 to 8) inside the measure.
- **`Array [FIGURA1_PATH_LEVEL]`** it shows the numerical code of the figure inside the layer.
- **`Array [FIGURA2_PATH_LEVEL]`** it shows the numerical code of an eventual figure which is part of the figure of the FIGURA1_PATH_LEVEL.
- **`Array [FIGURA3_PATH_LEVEL]`** it indicates the numerical code of an eventual figure which is part of the figure of the FIGURA2_PATH_LEVEL.
- **`Array [SIMBOLO_PATH_LEVEL]`** it is used to identify generic symbols different from those already listed. Typically this entry of the array is filled with the ClassID of the selected symbol. If the position LISTA_PATH_LEVEL has value 2, the numerical code of the interval symbol inside the list is placed in this entry.

The structure is completed by the information *Level*, useful to keep trace of the array's current position during the selection process.
All the entries of the structure SymPath are initialised to 0 as non-selection indication, whereas the numerical codes related to any selection are all different from 0: in this way an entry that is not void indicates a selection; for instance, if in the level BATTUTA_PATH_LEVEL of Array there is not a void value and in the levels below there are only 0, the identified object is a measure.
The introduction of three figures is due to the fact that beams may contain chords; in this case the FIGURA1_PATH_LEVEL shows the beam, the FIGURA2_PATH_LEVEL shows the chord inside the beam and the FIGURA3_PATH_LEVEL indicates the single note inside the chord. This is the limit case: in most cases the FIGURA3_PATH_LEVEL remains meaningless (with void entry).
Concerning the generation of an unequivocal numerical code for each element of each list of figures, measures, scores and intervals, the classes Figura, Battuta, Spartito and Intervallo have been endowed of the attribute NumericCode of the NumCode kind. The classes ListaFigure, ListaBattute, ListaSpartiti and ListaIntEst have a counter to assign the codes; for each new insertion in the list, the attribute NumericCode of the inserted element takes the value of the counter and the counter is augmented of 1. The same thing happens when a method of the Change type is performed: a new NumericCode is assigned to the new object

and the counter is augmented. When an object is deleted from the list, the counter is not decreased: in this way the codes are unequivocal and take into account also the deleted objects. Another possibility that has been assessed is to assign to NumericCode the position of the object in the list: in this way sending through the web messages like "delete note 3 in layer 1 of measure 82" would have been possible. This possibility appears more intuitive than that employed, but practically it is less safe: coming back to the above proposed example, it might happen (due to previous anomalies) that in the position 3 of the first layer of measure 82 there is not the desired note or there is not a note at all, but a rest. The proposed method ensures, from this point of view, a greater strength.
On the basis of what already said, it appears obvious that MASE/MASAE will perform a different selection with respect to DLIOO. From the position and the status of the mouse buttons, the MASE/MASAE performs a method (*Hit*) that produces:

- **`One pointer`** that is used from MASE/MASAE to recall methods for the command execution (*SymCommand*).
- **`One or two paths`** that are used from MASE/MASAE to find the objects interested to the command, and to pass the command to DLIOO: through the decoding of path (by means of the method *SymbolicHit*), the pointer needed to execute the command is obtained.

In order to find the objects present on the screen two kind of selection are foreseen:

- By means of a point on the screen where the user has pushed the left button of the mouse (*single selection*).
- By means of a rectangle specified from the user (*multiple selection*).

In the former case the selection methods return a path, in the latter a couple of path corresponding to the first and the last struck figure. Each graphic object derived from DrawObject has associated the rectangle that delimits it. In the selection of musical objects this feature is exploited to establish if an object contains a point (single selection) and if an object is contained in a rectangle (multiple selection). The parameters to be assigned to selection methods are:

1. Point or rectangle in which the selection is performed.
2. Type of object to be selected.

Construction of the symbolic Path

Single Selection

The problem that has been dealt with is the selection of an object present in the main score. Two kind of objects are distinguished:

1. simple objects that have no internal connections with other objects.
2. composed objects which, on the other hand, have connections with other objects that must be selectable.

In the DrawObject class the *Hit* method is present which allows controlling if a point on the screen selected by the mouse is contained in a precise object. In the case of simple objects the version of the method for DrawObject is enough, while for composed objects it is necessary to redefine it. The method is defined as follows:

Bool Hit(const Point& p, SelObj selObj, DrawObject*& drwObjSel, SymPath& symPath)

The following parameters are necessary:

const Point& p the point of the graphic screen that is to be controlled if it intersects a musical object designed on the main window.
SelObj objSel it shows the type of object to be selected and can have one of the following values:

- ANY_SEL selects any kind of object.
- SPARTITO_SEL selects a Spartito (Score).
- BATPART_SEL selects a Battuta (Measure) taking into account only the horizontal position of *p* (it is useful to select columns of measures).
- BATTUTA_SEL selects a Battuta.
- FIGURA_SEL selects a Figura (a Figure such as Note, Rest or Chord).
- NOTA_SEL selects a Nota (Note or Chord).

- NOTASING_SEL selects a Nota (it is useful to select a note inside a chord)
- GRUPPONOTE_SEL selects a beam.

DrawObject*& drwObjSel is an exit parameter that contains the pointer to the object or to the sub-object (musical objects connected to the examined object) that intersects the p point. It has NULL value if the point does not intersects any objects or if the struck object is not of the correct kind.

SymPath& symPath is the other exit parameter and represents the path obtained from the selection (obviously each class that performs this method sets one or at most two components of the PathArray).

Such method returns:

- A Boolean value that shows if any kind of object has been struck independently from whether the selection was correct or not.

The algorithm to recognise the object eventually selected is the following:

1. Control if the point is inside the symbol currently investigated.
2. If the type of object investigated is compatible with the class of investigation and its position on the screen intersects the selected point, place the numerical code of the object in the suitable position of PathArray and set the musical symbol in question as selected object.
3. If the object and the point do not intersect and a generic selection is not likely, the returned Boolean value is *false*.
4. If no valid selection on the main object has been performed and the type of object to be selected is compatible with some connected objects, then recall the Hit method on the connected objects.

A particular attention has to be paid to the selection with the parameter of the ANY_SEL kind, which has been mainly used in the process of objects' deletion. In this case, for the classes Accordo e GruppoNote we have two possibilities for the management of the selection:

1. Interpreting the selection of the main object (Chord or Beam).
   This means that the selection stops on the main object and, as a direct result, neither the single notes of a chord or of a beam nor the symbols to them associated can be deleted.
2. Transmit the selection to the connected objects (the notes of a chord or a beam)
   In this case the selection is moved to the sub-objects and it is not possible to delete chords or beams with a single selection but only a note in turn: this fact can result in having chords or beams without notes (by deleting all the notes of a chord, the chord is not deleted but it would result as a chord without notes).

The first solution has been chosen because it had fewer drawbacks.
The following table shows, for each class that has to redefine the Hit method:

- the name of the class
- kinds of selection to which the class has to answer
- kinds of selection that have to be transmitted to the connected objects.

Class	Object	Connected Object
Spartito	SPARTITO_SEL	BATTUTA_SEL, BATPART_SEL, FIGURA_SEL, NOTA_SEL, NOTASING_SEL, ANY_SEL, GRUPPONOTE_SEL
InEsteso	ANY_SEL	
Battuta	BATTUTA_SEL, BATPART_SEL	FIGURA_SEL, NOTA_SEL, NOTASING_SEL, ANY_SEL, GRUPPONOTE_SEL
Figura	FIGURA_SEL, ANY_SEL	ANY_SEL
Nota	FIGURA_SEL, NOTA_SEL, NOTASING_SEL, ANY_SEL	ANY_SEL
Spazio	FIGURA_SEL, NOTASING_SEL, ANY_SEL	ANY_SEL
Accordo	FIGURA_SEL, NOTA_SEL, ANY_SEL	NOTASING_SEL, ANY_SEL
GruppoNote	GRUPPONOTE_SEL, ANY_SEL	FIGURA_SEL, NOTA_SEL, NOTASING_SEL
Violino	ANY_SEL	ANY_SEL
Arco	ANY_SEL	ANY_SEL

Multiple selection

This kind of selection is the extension to the selection through rectangle of what previously seen with the single selection. Since transmitting lists of objects via web can lead to the exchange of too long messages, it has been decided that the multiple selection does not have to produce a list of objects but only a couple of them: the first and the last found in the rectangle and belonging to the type that has been set.
The method is defined as follows (in the class DrawObject):

Bool Hit (const Rectangle& r, SelObj objSel, SymPath& SP1, SymPath& SP2)

The following parameters are necessary:

const Rectangle& r shows the rectangle where we want to check if the object is "internal" and thus potentially selected.

SelObj objSel shows the type of object that is to be selected and can take one of the values previously listed.

SymPath& SP1 is the exit parameter corresponding to the path of the first object inside the rectangle that belongs to the type objSel. Since each list is scanned from left to right, concerning the physical position of the objects, the "first" object is that on the extreme left.

SymPath& SP2 is the exit parameter corresponding to the path of the last object inside the rectangle and belonging to the objSel type.

Such method returns:

- a Boolean value (true/false), that shows, independently from the fact that the selection is correct or not, if a musical object is really contained in the rectangle.

Although the definition is generic and allows selecting each kind of object that can be specified through one of the described types, currently the implementation of the method is focused on the search of generic figures on an unique layer, not necessarily belonging to a unique measure. In order to specify the layer of reference, the Hit method of the class Battuta needs a further entry parameter showing the selected layer.
Since this method does not produce pointers, after its execution can be necessary to recall the *SymbolicHit* method where an appropriate type of selObj has been set, in order to obtain the pointer to the object that allows performing the command set (it depends on the object in question producing, if necessary, the two pointers to the selected objects).

Identification of the physical pointer

Once identified the symbolic path that unequivocally characterises the musical symbol represented on the screen, it is necessary to identify the physical pointer to the object. For this reason in the DrawObject class also the *SymbolicHit* method, that obtains the pointer to the object starting from the symbolic path, is defined.

Bool SymbolicHit (SymPath& symPath, SelObj selObj, DrawObject*& drwObjSel)

The following parameters are necessary:

SymPath& symPath is the path to be "decoded".

SelObj objSel shows the kind of object that is to be searched (see the description of the preceding method).

DrawObject*& drwObjSel is the exit parameter and contains the pointer to the object or to the sub-object identified with symPath. If it has NULL value either nothing has been found or the object identified is not of the correct type.

This method returns:

- a Boolean value that shows if some object has been found (independently from the fact that the selection is correct or not).

Like Hit, also this method must be redefined in the classes derived from DrawObject that have sub-objects. This method works like Hit, the difference is that symPath is an entry parameter and it is on the basis of it that the object is searched (in the scanning of sub-objects is useful to employ the field Level of symPath). To be noticed is that this method takes into account the parameter objSel in the same manner as that previously described: for example, if objSel=BATTUTA_SEL and symPath has non-void components even beyond the BATTUTE_PATH_LEVEL, these ones are ignored and drwObjSel takes the pointer to the measure in question. This feature makes the method SymbolicHit useful to search, for instance, which measure a symbol shown by a path belongs to (it is enough to set as entry parameters the path associated to the symbol and the type BATTUTA_SEL).

6 WEDEL-OOMM interface

In this section is reported an excerpt of the interface of some classes, in particular are reported the protected attributes used to model music and not attributes used for visualisation, moreover are reported the methods that can be used to navigate in the model. Not all the classes have been reported only the most important ones, generally classes that represent only a symbol without specific attributes are not reported. The aim of this section is not to provide a complete documentation of the interfaces but to present how to navigate the music model.

DrawObject

This class represents the root class of the class tree, it can be considered a *WDFObject*.

Parent class

None

Protected attributes

ClassID ID	Unique identifier of the class.
Point AbsPos	Position of the object on the screen.
short Color	Color to be used to draw the object.

Public methods

ClassID GetID()	It returns the identifier of the class the object belongs.
Bool is(ClassID id)	It returns TRUE if the object belongs to the class identified by id.

Comments & examples

The enum type ClassID enumerates all the classes son of DrawObject.
Methods GetID() and is(…) can be used to see if an object belongs to a class.
For example:

```
  if(o->is(CL_ABBELLIMENTO))
    ...
```

checks if object o is an ornament.

Lista (list)

Class *Lista* is a general container of pointers to objects derived from class *DrawObject*.
An internal class, *Node*, is used to store the pointer of the object and the pointers to the previous and next nodes of the list.
In this way an object can be in more than one list.
Class *Node* is hidden from the outside and only the derived classes can access to the information stored in a node.
The Node is defined as follows:

```
struct Node
{
  DrawObject *ObjectPtr;
  Node *Next;
  Node *Prev;

  Node(void);          // initialises pointers to NULL
  Node(DrawObject *p); // initialises Next and Prev pointers to NULL
                       // and ObjectPtr to p
};
```

Parent class

None

Protected attributes

Node* first	The first node of the list.
Node *last	The last node of the list.
Node *lastRef	Last referred node.
Unsigned long numobject	The number of objects in the list.

Protected methods

Node* GetNext(Node * n)	It returns the node after node n, it returns NULL if it is the last.
Node* GetPrev(Node * n)	It returns the node before node n, it returns NULL if it is the

	first.
Node* FindNodePtr(DrawObject* ptr)	It returns the node that points to object ptr, it returns NULL if the object is not found.
void InsertAfter(Node *n, DrawObject *o)	It inserts after node n a new nodethat points to object o. If n is NULL object o is added as first element.
Public methods	
Lista()	Default constructor of empty list.
~Lista()	Destructor that deallocates nodes of the list but not objects in the list.
DrawObject* GetFirst()	It returns the first object in the list, it return NULL if the list is empty.
DrawObject* GetLast()	It returns the last object in the list, it return NULL if the list is empty.
DrawObject* GetNext(DrawObject* o)	It returns the object after object o in the list, it return NULL if o is the last object.
DrawObject* GetPrev(DrawObject* o)	It returns the object before object o in the list, it return NULL if o is the first object.
unsigned long GetNumObj()	It returns the number of objects in the list.
Bool isEmpty()	It returns TRUE if the list is empty.
Bool FindPtr(DrawObject* ptr)	It returns TRUE if pointer ptr is in the list.
long GetPos(DrawObject* ptr)	It returns the position of ptr in the list, -1 if not found, 0 the first, 1 the second, etc.
DrawObject* AtPos(unsigned long pos)	It return the object at position pos (0 the first, 1 the second, etc). It returns NULL if pos is not valid (<0 or >= numobject)
void InsertEnd(DrawObject *o)	It inserts the object o in the list as last element.
void InsertTop(DrawObject *o)	It inserts the object o in the list as first element.
Bool InsertAfter(DrawObject *o, DrawObject* ref)	It inserts the object o after object ref. If ref is NULL, o is added in front of the list and it returns TRUE. If ref is not found o is added in front of the list and it returns FALSE.
DrawObject* DelFirst()	It erases the first node of the list and it returns the object pointer
DrawObject* DelLast()	It erases the last node of the list and it returns the object pointer
void DelPtr(DrawObject* o)	It deletes from the list the node that points to object o, object o is not deleted.
void DelPtrNode(Node* n)	It deletes node n from the list.
void Clear()	It deallocates the nodes of the list.
void ClearDeep()	It deallocates the nodes and the objects in the list.
void Swap(DrawObject*o1,DrawObject*o2)	It swaps the objects o1 and o2 in the list.
DrawObject*Change(DrawObject*o1, DrawObject*o2)	It changes in the list the reference to o1 with a reference to o2.
Comments & examples	

1. Iteration of elements of a list from outside:
```
Lista lista;
…
DrawObject *o=lista.GetFirst();
while(o!=NULL)
{
  … use object o …
  o=lista.GetNext(o);
}
```
2. Iteration of elements of a list from a method of a derived class:
```
Node *n=first;
while(n!=NULL)
{
  DrawObject *o=n->ObjectPtr;
  … use object o …
  n=GetNext(n);
}
```

Partitura (Main Score Model)

Parent class

None

Protected attributes

ListaSpartiti listaSpa
ListaParentesi listaPar

Public methods

Spartito* GetFirstSpa()	It returns the first score (the upper).
Spartito* GetNextSpa(Spartito* s)	It returns the score after score s, NULL if s is the last.
Spartito* GetPrevSpa(Spartito* s)	It returns the score before score s, NULL if s is the first.
Spartito* GetSpartNum(int n)	It returns the score number n, NULL if not present.
int GetNumSpa()	It returns the number of scores

ListaSpartiti (list of parts)

Parent class

Lista

Protected attributes

Public methods

Comments & examples

Since *ListaSpartiti* is a *Lista*, the methods of *Lista* can be used to iterate through the scores.
For example:

```
ListaSpartiti *pScoreList;
…
Spartito *pScore=(Sparito*)pScoreList->GetFirst();
while(pScore!=NULL)
{
  … use object pScore …
  pScore=(Spartito*)pScoreList->GetNext(pScore);
}
```

While iteration of scores from a method of *ListaSpartiti* is:

```
Node *n=first;
while(n!=NULL)
{
  Spartito *pScore=(Spartito*)n->ObjectPtr;
  … use object pScore …
  n=GetNext(n);
}
```

Parentesi (parenthesis)

Parent class

DrawObject

Protected attributes

NumCode NumericCode	Identifier of the braket.
Spartito *PSpaStart	Score where the bracket starts.
Spartito *PSpaEnd	Score where the bracket ends.
int livello	Level of the bracket, it is calculated to avoid overlapping brackets.

Public methods

NumCode GetNumericCode()
Spartito *GetPSpaStart()
Spartito *GetPSpaEnd()
int GetLivello()

ListaParentesi (list of parenthesis for grouping parts)

Parent class

Lista

Protected attributes

Public methods

Parentesi *GetFirstPar()	It returns the first bracket.
Parentesi *GetNextPar(Parentesi *r);	It returns the bracket after r.

Comments & examples

For example:

```
  ListaParentesi *pBraList;
  …
  Parentesi *pBra=pBraList->GetFirstPar();
  while(pBra!=NULL)
  {
    … use object pBra …
    pBra=pBraList->GetNextPar(pBra);
  }
```

While iteration of scores from a method of *ListaParentesi* is:

```
  Node *n=first;
  while(n!=NULL)
  {
    Parentesi *pBra=(Parentesi*)n->ObjectPtr;
    … use object pBra …
    n=GetNext(n);
  }
```

Spartito (a single part)

Parent class

DrawObject

Protected attributes

ListaBattute listaBat	List of measures.
ListaIntEst listaInt	List of interval symbols.
ListaSillabe listaSil[4]	Lyric lines
char strumesec[STRUM_NAME_LEN]	Name of the instrument playing the score
NumCode NumericCode	Identifier of the score.
int numStaffs	Number of staffs used by the score (1,2 or 3)
int NCorde[3]	Number of lines of the pentagram of each staff.

Public methods

NumCode GetNumericCode()	
char *GetStrumExec()	
int GetNumberOfStaffs()	
int GetNCorde(int staff=0)	
Battuta* GetBattNum(int n)	It returns the measure n in the list (starting from 0)
Battuta* GetBattNumProg(int np)	It returns the measure number np (starting from 1).
Battuta* GetBatt(NumCode nc)	It returns the measure with numeric code nc.
Battuta* GetFirstBat()	It returns the first measure.
Battuta* GetNextBat(Battuta* pbt)	It returns the measure after a given measure.
Battuta* GetPrevBat(Battuta* pbt)	It returns the measure before a given measure.
Battuta* GetLastBat()	It returns the last measure.
void SetupIntRefs()	It builds the data structures associated to figures in the score to permit to know what intervals are “over” the figure.

ListaBattute (list of Measures)

Parent class

Lista

Protected attributes

Public methods

Battuta* GetBatt(NumCode nc)	It returns the measure with numeric code equal to nc.
Battuta* GetBattNumProg(int np)	It returns the measure with a specific number.
Battuta* GetBattWithFig(Figura* pFig)	It returns the measure with the specified figure.

Comments & examples

Since *ListaBattute* is a *Lista*, the methods of *Lista* can be used to iterate through the measures.
For example:

```
ListaBattute *pMeasureList;
…
Battuta *pMeasure=(Battuta*)pMeasureList->GetFirst();
while(pMeasure!=NULL)
{
  … use object pMeasure …
  pMeasure=(Battuta*)pMeasureList->GetNext(pMeasure);
}
```

While iteration of measures from a method of *ListaBattute* is:

```
Node *n=first;
while(n!=NULL)
{
  Battuta *pMeasure=(Battuta*)n->ObjectPtr;
  … use object pMeasure …
  n=GetNext(n);
}
```

Battuta (Measure)

Parent class

DrawObject

Protected attributes

NumCode NumericCode	Identifier of the measure.
int NumProgress	Progressive number of the measure.
int NumberOfStaffs	Number of staffs used for the measure (the same as the score)
Layer* layer[12]	The layers used in the measure.
Intestazione *intest[3]	The headers of the measure for the different staffs (if more than one are present)
Barra *ptrBarra	Pointer to the barline.
Barra *ptrBarraPrec	Pointer to the barline of the previous measure.
TipoBarra tpBarra	Type of barline for the measure
NumBattuta *ptrNumBattuta	Pointer to object used to display the measure number.
Scansione *ptrScansione	Pointer to a measure scanning object.
Riferimento *ptrRiferimento	Pointer to a reference object.
TSalto *ptrTSalto	Pointer to a TSalto object.
Movimento *ptrMovimento	Pointer to a movement object.

Public methods

int GetNumProgress()	It returns the number of the measure.
int GetNumberOfStaffs()	It returns the number of staffs that are used by the measure.
Intestazione *GetIntest(int staff=0)	It returns the intestazione of the staff (starting from 1)
Chiave* GetChiave(int staff=0)	It returns the clef of the staff (starting from 1)
ArmaturaChiave GetArmatura(int staff=0)	It returns the key signature of the staff (starting from 1)
Tempo GetTempo()	It returns the time signature of the measure.
Layer* GetPLayer(int nlayer)	It returns the layer number *nlayer (0,1,…11).*
Barra* GetBarra();	it returns the bar line of the measure.
NumBattuta *GetPNumBattuta()	
Riferimento *GetPRiferimento()	
TSalto *GetPTSalto()	
Bool CheckBatt()	It returns TRUE if the measure is time consistent with the time signature.
float GetMetronomoTimeExec()	It returns the metronome indication of the measure (0 if not present), it is calculated as 60/(note duration)*1000/(figures per minute).
float GetTempoTimeExec()	It returns the fraction numerator/denominator.

ListaFigure (list of Figures: rests, notes, chords, beams, etc.)

Parent class

Lista

Protected attributes

int nFig	Number of figures considering also figures in the beams.

Public methods

Figura *GetFirstF()	It returns the first figure of the list
Figura *GetLastF()	
Figura *GetNextF(Figura*)	
Figura *GetPrevF(Figura*)	
Figura *GetFirstFig()	It returns the first figure of the list entering in beams.
Figura *GetNextFig()	It returns the following figure.
Figure *AtPosFig(int pos)	It returns the figure at position pos without considering beams.
int GetPosFig(Figure*)	It returns the position of a figure without considering beams.

Comments & examples

The following example shows how to iterate trough the figures without considering if a figure is in a beam:

```
ListaFigure *pFigList;
…
Figura *pFig=pFigList->GetFirstFig();
while(pFig!=NULL)
{
  … use object pFig …
  pFig=pFigList->GetNextFig();
}
```

Layer (each signgle voice in a measure)

Parent class

ListaFigure

Protected attributes

Public methods

Comments & examples

Since a Layer is a ListaFigure, the methods of ListaFigure can be used to iterate through the figures of the layer.

Intestazione (clef, key signature, time signature)

Parent class

DrawObject

Protected attributes

ArmaturaChiave armChiave	Key signature of the measure
Tempo tempo	Time of the measure
Chiave *ptrChiave	Clef of the measure

Public methods

Chiave *GetChiave()
ArmaturaChiave GetArmatura()
ArmaturaChiave* GetPtrArmatura()
TipoArmaturaChiave GetTipoArmatura()
Tempo GetTempo()
Tempo* GetPtrTempo()

ArmaturaChiave (key Signature)

This class uses the following enumerator type:

```
enum tipoArmaturaChiave {
  DO_maggiore,SOL_maggiore,RE_maggiore,LA_maggiore,MI_maggiore,
  SI_maggiore,FAd_maggiore,DOd_maggiore,FA_maggiore,SIb_maggiore,
  MIb_maggiore,LAb_maggiore,REb_maggiore,SOLb_maggiore,DOb_maggiore,
  LA_minore,MI_minore,SI_minore,FAd_minore,DOd_minore,SOLd_minore,
  REd_minore,LAd_minore,RE_minore,SOL_minore,DO_minore,FA_minore,
  SIb_minore,MIb_minore,LAb_minore
};
```

Parent class

Ancoraggio

Protected attributes

tipoArmaturaChiave tipoArm	Type of key signature.

Public methods

TipoArmaturaChiave GetArmatura()

Tempo (time signature)

Parent class

DrawObject

Protected attributes

Bool LetteraC	If it is TRUE time is represented with C or dashed C
NumTempo Numeratore	Numerator of tempo fraction
NumTempo Denominatore	Denominator of tempo fraction

Public methods

int GetNumeratore()	It returns the numerator of tempo fraction as an integer
int GetDenominatore()	It returns the denominator of tempo fraction as an integer

Bool SetTempo(char* tempo)	It sets the attributes parsing the string tempo:
	if tempo=”c” then LetteraC is set to TRUE and Numeratore to 4 and Denominatore to 4.
	if tempo=”C” then LetteraC is set to TRUE and Numeratore to 2 and Denominatore to 2.
	If tempo is of the form ”num/den” then LetteraC is set to FALSE and Numeratore to num and Denominatore to den.

Figura (generic figure)

Parent class

DrawObject

Protected attributes

int Altezza	Height of the musical figure, 0= first line, 1=first space, 2=second line, 3=second space etc..
float durata	Natural duration of the musical figure, without considering augmentation dots.
Bool Tie	It indicates if a tie is starting from this note to the following note.
int Staff	Staff where the figure have to be drawn.
Bool Figurina	Determines if the figure ha to be drawn small (Figurina=TRUE) or normal (Figurina=FALSE)
NumCode NumericCode	Identifier of the figure
LegatQuadra *terz	Pointer to the LegatQuadra representing a terzina starting from this figure.
Corona *PCorona;	Pointer to a fermata object.
Occhiali *POcchiali;	Pointer to a glasses object.
PuntoValore *PpuntoValore	Pointer to an augmentation dot object.
Tdinamico *PTDinamico	Pointer to a dynamics object.
Tgenerico *PTGenerico	Pointer to a generic text object.
Strumento *Pstrumento	Pointer to an Instrument specific object
Annotazione *Pannotazione	Pointer to a main score annotation object.
DitaCorde *PditaCorde	Pointer to a fret board object

Public methods

float GetDurata()	It returns the duration of the figure, if it is a small figure it returns 0 and if augmentation dots are present are considered to evaluate the proper duration.
Bool GetTie()	
int GetAltezza()	
int GetNStaff()	It returns –1 if it is a multistaff figure (chord/beam)
virtual void GetMinMaxStaff(short& min, short& max)	
Bool GetNotina()	It returns TRUE if the figure is small.
Bool IsANote()	It returns TRUE if the object is a note.
Bool IsARest()	It returns TRUE if the object is a rest.
Corona *GetPCorona()	
Occhiali *GetPOcchiali()	
PuntoValore *GetPPuntoValore()	
Tdinamico *GetPTDinamico()	
Tgenerico *GetPTGenerico()	
Strumento *GetPStrumento()	
Annotazione *GetPAnnotazione()	
DitaCorde *GetPDitaCorde()	
Status_ID GetStatus()	It returns the status of the figure, it can be: NORMAL – visible and with visual duration GRACED – visible and without visual duration HIDDEN – invisible and with visual duration GHOSTED – invisible and without visual duration
int GetNIntRefs()	It returns the number of interval symbols insisting over the figure.
IntRef *GetIntRef(int n)	It returns a pointer to : `struct IntRef {` `IntEsteso *intRef;` `enum {INT_START,INT_OVER,INT_END} type;` `}` where *intRef* is a pointer to the interval symbol and *type* indicates if the interval is starting on the figure, is active on the figure or is ending on the figure.

Nota (note)

Parent class

Figura

Protected attributes

Suddivisione *PSuddivisione
Abbellimento *PAbbellimento
Sordina *PSordina
Armonici *PArmonici
Diteggiato *PDiteggiato
EspressComposta *PEspressComposta
AlterazComposta *PAlterazComposta

Public methods

Suddivisione *GetPSuddivisione()
Abbellimento *GetPAbbellimento()
Sordina *GetPSordina()
Armonici *GetPArmonici()
Diteggiato *GetPDiteggiato()
EspressComposta *GetPEspressComposta()
AlterazComposta *GetPAlterazComposta()

Pausa (rest)

Parent class

Figura

Protected attributes

Public methods

Accordo (choard)

Class *Accordo* represents a chord.

Parent class

Nota, ListaFigure

Protected attributes

ClassID NoteID	Class identifier of the notes in chord (e.g. CL_NMINIMA,CL_NCROMA etc.)
Arpeggio *PArp	Pointer to an arpeggio object
Bool DiteggiatoUp	It indicates if fingering have to be placed above (TRUE) or below (FALSE) the chord

Public methods

Arpeggio *GetPArpeggio()	
ClassID GetNoteID()	It returns the class identifier of the notes of the chord.
Bool AddNota(Nota *pn)	It adds a note to the chord.

Comments & examples

Since Accordo is a ListaFigure, the methods of ListaFigure can be used to iterate through the notes of the chord.

For example:

```
Accordo *pAcc;
…
Nota *pNote=(Nota*)pAcc->GetFirstF();
while(pNote!=NULL)
{
  … use object pNote …
  pNote=(Nota*)pAcc->GetNextF(pNote);
}
```

While iteration of notes from a method of Accordo is:

```
Node *n=first;
while(n!=NULL)
{
  Nota *pNote=(Nota*)n->ObjectPtr;
  … use object pNote …
  n=GetNext(n);
}
```

GruppoNote (beam of notes)

This class is used to represent beamed notes.

Parent class

Figura, ListaFigure

Protected attributes

Public methods

Comments & examples

Since *GruppoNote* is a *ListaFigure*, the methods of *ListaFigure* can be used to iterate through the figures (notes, rests and anchorage).

For example:

```
   GruppoNote *pBeam;
   …
   Figura *pFig=pBeam->GetFirstF();
   while(pFig!=NULL)
   {
     … use object pFig …
     pFig=pBeam->GetNextF(pFig);
   }
```

While iteration of figures from a method of *GruppoNote* is:

```
   Node *n=first;
   while(n!=NULL)
   {
     Figura *pFig=(Figura*)n->ObjectPtr;
     … use object pFig …
     n=GetNext(n);
   }
```

AlterazComposta (accidentals even composite)

Parent class

DrawObject

Protected attributes

int n_alterazioni	Number of accidentals
Alterazione* alterazioni[N_ALT_MAX]	Vector of pointers to the accidentals objects

Public methods

int Getn_alterazioni()	It returns the number of accidentals.
Alterazione* GetPAlterazione(int i)	It returns the accidental at position i, starting from 1.

Comments & examples

The type of accidental can be retrieved using the GetID() method of *DrawObject*, for example:

```
for(i=1; i<=Getn_alterazioni(); i++)
  switch(GetPAlterazione(i)->GetID())
  {
    case CL_DIESIS:
      … sharp …
      break;
    case CL_BEMOLLE:
      … flat …
      break;
    case CL_BEQUADRO:
      … natural …
      break;
    … etc …
  }
```

EspressComposta (composite expression)

Parent class

Indicazione

Protected attributes

Int n_espressioni	The number of expression symbols
Espressione* espressioni[N_ESP_MAX]	Vector of pointers to the expressions symbols

Public methods

Int Getn_espressioni()	It returns the number of expressions symbols.
Espressione* GetPEspressione(int i)	It returns the expression at position i, (position starts from 1).
Espressione* GetPEspressione(ClassID, int Idsym=0)	It returns the expression (if it exists) of class identified by the ClassID and for generic expressions also by the symbol

identifier.

Comments & examples

The kind of expression can be retrieved using the GetID() method of *DrawObject*, for example:

```
for(i=1; i<=Getn_espressione(); i++)
  switch(GetPEspressione(i)->GetID())
  {
    case CL_STACCATO:
      … staccato …
      break;
    case CL_TENUTO:
      … tenuto …
      break;
    case CL_SFORZATO:
      … sforzato …
      break;
    … etc …
  }
```

Abbellimento (ornaments)

Parent class

DrawObject

Protected attributes

Bool aboveNota	Is TRUE if the symbol is placed above the note.
Bool Doppio	Is TRUE for double Mordente.
Alterazione *PAltSopra	Pointer to the accidental over the symbol
Alterazione *PAltSotto	Pointer to the accidental below the symbol

Public methods

Bool GetAboveNota()
Bool GetDoppio()
Alterazione* GetPAltSopra()
Alterazione* GetPAltSotto()

Violino (violin symbols)

Parent class

Strumento

Protected attributes

Corda *Pcorda

Public methods

Corda *GetPCorda()

Arco (arc)

Parent class

Violino

Protected attributes

Bool ScrivereArco	If it is TRUE the “arco” symbol is drawn.
DirezArco *PDirezArco	
ParteArco *PParteArco	
PosizArco *PPosizArco	

Public methods

Bool GetScrivereArco()
DirezArco *GetPDirezArco()
ParteArco *GetPParteArco()
PosizArco *GetPPosizArco()

IntEsteso (horizontal symbols: slur, tie, crescendo, etc…)

Parent class

DrawObject

Protected attributes

NumCode NumericCode	Identifier of the interval.
Figura *PFigStart	The starting figure of the interval symbol
Figura *PFigEnd	The ending figure of the interval symbol
Bool Sopra	It is TRUE if the interval is above the figures.

Public methods

Figura* GetPFigStart()
Figura* GetPFigEnd()
Bool GetSopra()

ListaIntEst (list of horizontal symbols)

Parent class

Lista

Protected attributes

Public methods

Comments & examples

Since *ListaIntEst* is a *Lista*, the methods of *Lista* can be used to iterate through the interval symbols. For example:

```
ListaIntEst *pIntList;
…
IntEsteso *pInt=(IntEsteso*)pIntList->GetFirst();
while(pInt!=NULL)
{
  … use object pInt …
  pInt=pIntList->GetNext(pInt);
}
```

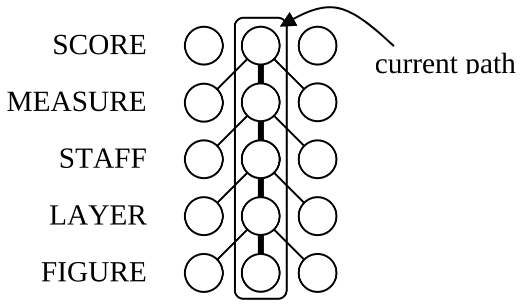
While iteration of intervalsfrom a method of *ListaIntEst* is:

```
Node *n=first;
while(n!=NULL)
{
  IntEsteso *pInt=(IntEsteso*)n->ObjectPtr;
  … use object pMeasure …
  n=GetNext(n);
}
```

7 WEDEL-OOMM Service example

In this section is presented a simple service that permits the navigation of the main score. It is only an example, it doesn't pretend to be complete or perfect, it is only used to show how a service can be implemented.

The idea is to have a current path (score, measure, staff, layer, figure) and with specific commands permit to change the current score (go to first, last, next and previous score), when the current score is chosen another command is used to iterate trough the measures of the score selecting the current measure, then the staffs of the measure can be selected , the same for the layer of the staff and finally for the figures in the layer.



Class *ExServiceCommand* is defined to represent a navigation command, it is composed of two parts:
1. the movement command, that can be FIRST, NEXT, PREV and LAST.
2. the level on which the movement have to be applied, it can be SCORE, MEASURE, STAFF, LAYER, FIGURE.

```
enum Op {FIRST,NEXT,PREV,LAST};
enum Level {SCORE,MEASURE,STAFF,LAYER,FIGURE};

class ExServiceCommand
{
  protected:
    Op op;
    Level level;
  public:
    ExServiceCommand() { Set(FIRST,SCORE); };
    ExServiceCommand(Op o,Level l) { Set(o,l); };
    void Set(Op o, Level l) { op=o; level=l; };
    Op GetOp() { return op; };
    Level GetLevel() { return level; };
};
```

Class *ExServicePath* is used to store the current path, it presents the methods to get and set the attributes, and if an attribute is set the attributes of the lower levels are reset to NULL values :

```
class ExServicePath
{
  protected:
    Spartito    *score;
    Battuta     *measure;
    int          staff;
    int          layer;
    Figura      *fig;

  public:
    ExServicePath()
    {
      score=NULL;
      measure=NULL;
      staff=0;
      layer=0;
      fig=NULL;
    };

    Spartito* GetScore() { return score; };
    void SetScore(Spartito* s) {
       score=s; measure=NULL; staff=0; layer=0; fig=NULL;
    };

    Battuta* GetMeasure() { return measure; };
    void SetMeasure(Measure* m) {
       measure=m; staff=1; layer=1; fig=NULL;
    };

    int GetStaff() { return staff; };
    void SetStaff(int s) { staff=s; layer=1; fig=NULL; };
    void NextStaff(int max) {
      staff++; layer=0; fig=NULL;
      if (staff>max) staff=0;
    };
    void PrevStaff() {
      staff--; layer=0; fig=NULL;
      if (staff<0) staff=0;
    };

    int GetLayer() { return layer; };
    void SetLayer(int l) { layer=l; fig=NULL; };
    void NextLayer(int max) {
      layer++; fig=NULL;
      if (layer>max) layer=0;
    };
    void PrevLayer() {
      layer--; fig=NULL;
      if (layer<0) layer=0;
    };

    Figura* GetFig() { return fig; };
    void SetFig(Figura* f) { fig=f;};
};
```

Class *ExServiceData* is the data structure given to all the ExService methods in the music classes.
It is composed of the ExServiceCommand (the command to be executed) and the exServicePath (the current path):

```
class ExServiceData
{
  public:
    ExServiceCommand cmd;
    ExServicePath    current;
};
```

In method ExService of class Partitura, for first is checked the level where the command is to be applied, if it is the SCORE level the command is executed using the information accessible from Partitura (listaSpa). If the level is not correct the command is executed calling the ExService method of Spartito (the lower level).

```
int Partitura::ExService(ExServiceData *data)
{
  if(data->cmd.GetLevel()==SCORE)
  {
    switch(data->cmd.GetOp())
    {
      case FIRST:
        data->current.SetScore(listaSpa.GetFirstSpa());
        break;
      case LAST:
        data->current.SetScore(listaSpa.GetLastSpa());
        break;
      case NEXT:
        data->current.SetScore(listaSpa.GetNextSpa(
                                data->current.GetScore()));
        break;
      case PREV:
        data->current.SetScore(listaSpa.GetPrevSpa(
                                data->current.GetScore()));
        break;
    }
    if(data->current.GetScore()!=NULL)
      return 1; // all OK!
    return 0; // at the end/begin or something wrong!
  }
  return data->current.GetScore()->ExService(data);
}
```

The method ExService of class Spartito has the same structure of the previous one. If the level is correct the command is executed from the method otherwise the execution is delegated to the lower level (Battuta).

```
int Spartito::ExService(ExServiceData *data)
{
  if(data->cmd.GetLevel()==MEASURE)
  {
    switch(data->cmd.GetOp())
    {
      case FIRST:
        data->current.SetMeasure((Battuta*)listaBat.GetFirst());
        break;
      case LAST:
        data->current.SetMeasure((Battuta*)listaBat.GetLast());
        break;
      case NEXT:
        data->current.SetMeasure(
                 (Battuta*)listaBat.GetNext(data->current.GetMeasure()));
        break;
      case PREV:
        data->current.SetMeasure(
                 (Battuta*)listaBat.GetPrev(data->current.GetMeasure()));
        break;
    }
    if(data->current.GetMeasure()!=NULL)
      return 1; // all OK!
    return 0; // at the end/begin or something wrong!
  }
  return data->current.GetMeasure()->ExService(data);
}
```

The ExService method of Battuta have to manage the commands for the staffs and for the layers.

```
int Battuta::ExService(ExServiceData *data)
{
  switch(data->cmd.GetLevel())
  {
    case STAFF:
      switch(data->cmd.GetOp())
      {
        case FIRST:
          data->current.SetStaff(1);
          break;
        case LAST:
          data->current.SetStaff(GetNumStaffs());
          break;
        case NEXT:
          data->current.NextStaff(GetNumStaffs());
          break;
        case PREV:
          data->current.PrevStaff();
          break;
      }
      if(data->current.GetStaff()!=0)
        return 1; // all OK!
      return 0; // something wrong!
    case LAYER:
      switch(data->cmd.GetOp())
      {
        case FIRST:
          data->current.SetLayer();
          break;
        case LAST:
          data->current.SetLayer(4);
          break;
        case NEXT:
          data->current.NextLayer(4);
          break;
        case PREV:
          data->current.PrevLayer();
          break;
      }
      if(data->current.GetLayer()!=0)
        return 1; // all OK!
      return 0; // something wrong!
    default:
      pLayer=GetPLayer(data->current.GetStaff(),
                       data->current.GetLayer());
      if(pLayer!=NULL)
        return pLayer->ExService(data);
  }
  return 0; // oops… something wrong!
}
```

The method ExService of class Layer executes the command for iterating the figures, in this implementation the methods GetFirstFig and GetNextFig are used to iterate through the figures of the layer without considering beams as a figure. The LAST and PREV command are not implemented since ListaFigure doesn’t have methods for this.

```
int Layer::ExService(ExServiceData *data)
{
  // the level have to be FIG.
  // assert(data->cmd.GetLevel()==FIGURE)

  switch(data->cmd.GetOp())
    {
      case FIRST:
        data->current.SetFig(GetFirstFig());
        break;
      case LAST:
        break;
      case NEXT:
        data->current.SetFig(GetNextFig());
        break;
      case PREV:
        break;
    }
  if(data->current.GetFig()!=NULL)
    return 1; // all OK!
  return 0; // at the end/begin or something wrong!
}
```

An example of use of this service is the following, it iterates trough the figures of the first measure of the first score (one staff and one layer is assumed) and prints the duration of each object:

```
Partitura *pPartit;
…

ExServiceData data;

data.cmd.Set(FIRST,SCORE);
pPartit->ExService(&data);
data.cmd.Set(FIRST,MEASURE);
pPartit->ExService(&data);

data.cmd.Set(FIRST,FIGURE);
pPartit->ExService(&data);
while(data.current.GetFig()!=NULL)
{
  printf(“durate = %f\n”,data.current.GetFig()->GetDurata());
  data.cmd.Set(NEXT,FIGURE);
  pPartit->ExService(&data);
}
```

8 Commands

In this section is reported the detailed documentation of the most important editing commands, together with some code examples. A command is represented with an object of struct SymCmd (defined in lioo.hpp). Each command has the following attributes:

CmdId CommandId	the id of the command
SelType SelObjType	the type of object to be selected
SymPath Path1	symbolic path to the object to wich the command applies
SymPath Path2	eventual other object (mainly used for horizontal symbols)
int ArgV[]	array of arguments of the command
char *Text	eventual text argument

In the following subsections each command id (CMD_…) is followed by the description of the meaning of each command argument (Path1, Path2, ArgV[], Text)

8.1 Add a new score command

CMD_PARTITURA_ADD_SPARTITO:

Path1	path to the reference score
ArgV[0]	if TRUE adds a new score before the score indicated in Path1
ArgV[1]	Number of staffs of the score (1,2 or 3)
Text	Name of the score.

Example:
The following piece of code is used to add a 2-staff score before the first score:

```
Spartito *psp=mainScore->GetSpartNum(0); //the first score

SymCmd cmd;
cmd.CommandId=CMD_PARTITURA_ADD_SPARTITO; // the command
cmd.SelObjType=SPARTITO_SEL; // select a score
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode();
cmd.Path1.Level=SPARTITO_PATH_LEVEL;
  //cmd.Path1 identifies the reference score
cmd.ArgV[0]=TRUE; // adds a new score before the first score
cmd.ArgV[1]=2;    // two staffs score
strcpy(cmd.Text,"Piano staff"); // sets the name of the score

mainScore->SetNetSymCmd(cmd);
  //sets the command as given form the network
mainScore->SetUsrSymCmd(cmd);
  //sets the command as given from the user interface
mainScore->DoNetSymCmd();  // executes the command set in NetSymCmd
((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
  // adds the command to the log of commands but it uses also UsrSymCmd...
```

8.2 Add a new measure command

CMD_PARTITURA_ADD_BATTUTA:

Path1	measure after/before which a new measure has to be added
ArgV[1]	if TRUE adds the measure after.

Note:
this command modify only one score not all the scores, in this version it is not safe to add a measure to only one score, you have to add a measure to all the scores.

Example:
```
Spartito *psp=mainScore->GetSpartNum(0);
while(psp!=NULL)
{
      Battuta* pbat=psp->GetBattNum(0);

      SymCmd cmd;
      cmd.CommandId=CMD_PARTITURA_ADD_BATTUTA; // the command
      cmd.SelObjType=BATTUTA_SEL; // select a measure
      cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
      cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
      cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
      cmd.Path1.Level=BATTUTA_PATH_LEVEL;
      cmd.ArgV[1]=TRUE;   // adds the measure after

      mainScore->SetNetSymCmd(cmd);
      //sets the command as given form the network
      mainScore->SetUsrSymCmd(cmd);
      //sets the command as given from the user interface
      mainScore->DoNetSymCmd();  // executes the command set in NetSymCmd
      ((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
      // adds the command to the log of commands but it uses also UsrSymCmd...
      psp=mainScore->GetNextSpa(psp);
}
```

8.3 Clef commands

CMD_BATTUTA_CVIOLINO
CMD_BATTUTA_CVIOLINO8
CMD_BATTUTA_CVIOLINO8SOPRA
CMD_BATTUTA_CBASSO
CMD_BATTUTA_CBASSO8
CMD_BATTUTA_CBASSO8SOPRA
CMD_BATTUTA_CBASSOOLD
CMD_BATTUTA_CBARITONO
CMD_BATTUTA_CTENORE
CMD_BATTUTA_CTENORE8
CMD_BATTUTA_CCONTRALTO
CMD_BATTUTA_CMEZZOSOPRANO
CMD_BATTUTA_CSOPRANO
CMD_BATTUTA_CPERCUSBOX
CMD_BATTUTA_CPERCUS2LINES
CMD_BATTUTA_CTAB
CMD_BATTUTA_CVUOTA

To change the clef in the header:

```
Spartito *psp=mainScore->GetSpartNum(0);
Battuta* pbat=psp->GetBattNum(0);

SymCmd cmd;
cmd.CommandId= CMD_BATTUTA_CTENORE;
cmd.SelObjType=BATTUTA_SEL; // select any symbol
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
cmd.Path1.Array[STAFF_PATH_LEVEL]=0; //staff where the clef has to be changed
cmd.Path1.Level=BATTUTA_PATH_LEVEL;
cmd.ArgV[2]=0;    // change in the header of the measure

mainScore->SetNetSymCmd(cmd);
mainScore->SetUsrSymCmd(cmd);
mainScore->DoNetSymCmd();
((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
```

To change the clef inside the measure:

```
Spartito *psp=mainScore->GetSpartNum(0);
Battuta* pbat=psp->GetBattNum(0);
Figura * pfig=pbat->GetPLayer(0)->GetFirstFig();

SymCmd cmd;
cmd.CommandId= CMD_BATTUTA_INS_SPAZIO;
cmd.SelObjType=BATTUTA_SEL;
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
cmd.Path1.Array[STAFF_PATH_LEVEL]=0; //staff where the clef has to be changed
cmd.Path1.Level=BATTUTA_PATH_LEVEL;
cmd.ArgV[1]=1; // layer
cmd.ArgV[2]=CMD_BATTUTA_CTENORE;    //which clef
cmd.ArgV[4]=pfig->GetNumericCode(); // after which figure the clef has to be changed
cmd.ArgV[5]=FALSE; //not the first
cmd.ArgV[9]=0; //staff (0, 1, 2)

mainScore->SetNetSymCmd(cmd);
mainScore->SetUsrSymCmd(cmd);
mainScore->DoNetSymCmd();
((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
```

8.4 Key Signature commands

CMD_BATTUTA_ARMCHV_0D
CMD_BATTUTA_ARMCHV_1D
CMD_BATTUTA_ARMCHV_2D
CMD_BATTUTA_ARMCHV_3D
CMD_BATTUTA_ARMCHV_4D
CMD_BATTUTA_ARMCHV_5D
CMD_BATTUTA_ARMCHV_6D
CMD_BATTUTA_ARMCHV_7D
CMD_BATTUTA_ARMCHV_1B
CMD_BATTUTA_ARMCHV_2B
CMD_BATTUTA_ARMCHV_3B
CMD_BATTUTA_ARMCHV_4B
CMD_BATTUTA_ARMCHV_5B
CMD_BATTUTA_ARMCHV_6B
CMD_BATTUTA_ARMCHV_7B
CMD_BATTUTA_ARMCHV_0Dm
CMD_BATTUTA_ARMCHV_1Dm
CMD_BATTUTA_ARMCHV_2Dm
CMD_BATTUTA_ARMCHV_3Dm
CMD_BATTUTA_ARMCHV_4Dm
CMD_BATTUTA_ARMCHV_5Dm
CMD_BATTUTA_ARMCHV_6Dm
CMD_BATTUTA_ARMCHV_7Dm
CMD_BATTUTA_ARMCHV_1Bm
CMD_BATTUTA_ARMCHV_2Bm
CMD_BATTUTA_ARMCHV_3Bm
CMD_BATTUTA_ARMCHV_4Bm
CMD_BATTUTA_ARMCHV_5Bm
CMD_BATTUTA_ARMCHV_6Bm
CMD_BATTUTA_ARMCHV_7Bm

To change the key signature in the header of the measure:

```
Spartito *psp=mainScore->GetSpartNum(0);
Battuta* pbat=psp->GetBattNum(0);

SymCmd cmd;
cmd.CommandId= CMD_BATTUTA_ARMCHV_1Bm;
cmd.SelObjType=BATTUTA_SEL; // select any symbol
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
cmd.Path1.Array[STAFF_PATH_LEVEL]=0; //staff where the keysign. has to be changed
cmd.Path1.Level=BATTUTA_PATH_LEVEL;
cmd.ArgV[2]=0;    // change in the header of the measure

mainScore->SetNetSymCmd(cmd);
mainScore->SetUsrSymCmd(cmd);
mainScore->DoNetSymCmd();
((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
```

To change the key signature inside the measure:

```
Spartito *psp=mainScore->GetSpartNum(0);
Battuta* pbat=psp->GetBattNum(0);
Figura * pfig=pbat->GetPLayer(0)->GetFirstFig();

SymCmd cmd;
cmd.CommandId= CMD_BATTUTA_INS_SPAZIO;
cmd.SelObjType=BATTUTA_SEL;
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
cmd.Path1.Array[STAFF_PATH_LEVEL]=0; //staff where the clef has to be changed
cmd.Path1.Level=BATTUTA_PATH_LEVEL;
cmd.ArgV[1]=1; // layer
cmd.ArgV[2]= CMD_BATTUTA_ARMCHV_1Bm;    //which key signature
cmd.ArgV[4]=pfig->GetNumericCode();
cmd.ArgV[5]=FALSE; //not the first
cmd.ArgV[9]=0; //staff (0, 1, 2)

mainScore->SetNetSymCmd(cmd);
mainScore->SetUsrSymCmd(cmd);
mainScore->DoNetSymCmd();
((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
```

CMD_BATTUTA_TEMPO

```
Spartito *psp=mainScore->GetSpartNum(0);
Battuta* pbat=psp->GetBattNum(0);

SymCmd cmd;
cmd.CommandId= CMD_BATTUTA_TEMPO;
cmd.SelObjType=BATTUTA_SEL;
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
cmd.Path1.Array[STAFF_PATH_LEVEL]=0; //staff where the keysign. has to be changed
cmd.Path1.Level=BATTUTA_PATH_LEVEL;
strcpy(cmd.Text,”6/8”);

mainScore->SetNetSymCmd(cmd);
mainScore->SetUsrSymCmd(cmd);
mainScore->DoNetSymCmd();
((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
```

8.5 Note insert commands

CMD_BATTUTA_INS_NBREVE:
...
CMD_BATTUTA_INS_NCROMA:
...
CMD_BATTUTA_INS_NFUSA:

Path1	measure where the note has to be added
ArgV[0]	note height (pitch) always positive sign is coded with ArgV[7]
ArgV[1]	layer (1-12)
ArgV[2]	stem direction SOPRA/SOTTO/AUTOMATICO (up/down/automatic)
ArgV[3]	in chord (TRUE/FALSE)
ArgV[4]	Numeric Code (id) of the previuous figure (not used if ArgV[5] is TRUE)
ArgV[5]	insert as first figure (TRUE/FALSE)
ArgV[6]	after the execution contains the id of the figure inserted.
ArgV[7]	TRUE if note height is positive (TRUE/FALSE)
ArgV[8]	head type (HEAD_CLASSIC)
ArgV[9]	staff of the figure (0=upper, 1=middle/lower, 2=lower)

Example:
The following piece of code is used to add a croma to the first measure of the first score.

```
Spartito *psp=mainScore->GetSpartNum(0); //the first scoret measure
Battuta* pbat=psp->GetBattNum(0); //the first measure

SymCmd cmd;
cmd.CommandId=CMD_BATTUTA_INS_NCROMA; // the command
cmd.SelObjType=BATTUTA_SEL; // select a measure
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
cmd.Path1.Level=BATTUTA_PATH_LEVEL;
  //cmd.Path1 identifies the measure where the figure have to be added

cmd.ArgV[0]=0;    // altezza (pitch) always positive sign is coded with argv 7
cmd.ArgV[7]=TRUE; // altezza is positive
cmd.ArgV[1]=1;    // the layer (1-12)
cmd.ArgV[2]=AUTOMATICO; // SOPRA/SOTTO/AUTOMATICO (stem up/stem down/automatic)
cmd.ArgV[3]=FALSE; // in chord
cmd.ArgV[4]=0; // numeric code of the preceeding figure, in this case it doesn't matter
cmd.ArgV[5]=TRUE; // insert as first note;
cmd.ArgV[8]=HEAD_CLASSIC; // head type
cmd.ArgV[9]=0; // first staff  (0=upper, 1=middle, 2=lower)

mainScore->SetNetSymCmd(cmd);
  //sets the command as given form the network (the other lecterns)
mainScore->SetUsrSymCmd(cmd);
  //sets the command as given from the user interface
mainScore->DoNetSymCmd();  // executes the command set in NetSymCmd
```

```
((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
  // adds the command to the log of commands but it uses also UsrSymCmd...
```

8.6 Pause insert commands

CMD_BATTUTA_INS_PBREVE:
...
CMD_BATTUTA_INS_PCROMA:
...
CMD_BATTUTA_INS_PFUSA:

Path1	measure where the pause has to be added
ArgV[0]	pause height (pitch) always positive sign is coded with ArgV[7]
ArgV[1]	layer (1-12)
ArgV[2]	AUTOMATICO = default height SOPRA/SOTTO=height from ArgV[0],ArgV[7]
ArgV[4]	Numeric Code (id) of the previuous figure (not used if ArgV[5] is TRUE)
ArgV[5]	insert as first figure (TRUE/FALSE)
ArgV[6]	after the execution contains the id of the figure inserted.
ArgV[7]	TRUE if pause height is positive (TRUE/FALSE)
ArgV[9]	staff of the figure (0=upper, 1=middle/lower, 2=lower)

8.7 Symbol commands

CMD_NOTA_STACCATO
CMD_NOTA_ACCENTO
CMD_NOTA_MARTELLATO
CMD_NOTA_SFORZATO
CMD_NOTA_MARTDOLCE
CMD_NOTA_PUNTOALL
CMD_NOTA_GRUPPETTO_SUP_1
CMD_NOTA_GRUPPETTO_INF_1
CMD_NOTA_MORDENTE_SUP
CMD_NOTA_MORDENTE2_SUP
CMD_NOTA_MORDENTE_INF
CMD_NOTA_MORDENTE2_INF
CMD_NOTA_CONSORDINAARCHI
CMD_NOTA_CORDA_1
CMD_NOTA_CORDA_2
CMD_NOTA_CORDA_3
CMD_NOTA_CORDA_4
CMD_NOTA_CORDA_5

Path1	Figure where the symbol has to be added
ArgV[0]	position SOPRA/SOTTO/AUTOMATICO (over/below/automatic)

Example:
The following piece of code is used to add a tenuto symbol to the first note of the first measure of the first staff

```
Spartito *psp=mainScore->GetSpartNum(0);
Battuta* pbat=psp->GetBattNum(0);
Figura * pfig=pbat->GetPLayer(0)->GetFirstFig();
   //the first figure is a note/chord in a beam or not in a beam.

SymCmd cmd;
cmd.CommandId=CMD_NOTA_TENUTO;
cmd.SelObjType=NOTA_SEL; // select a note
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
cmd.Path1.Array[LAYER_PATH_LEVEL]=1; //1st layer
cmd.Path1.Array[FIGURA1_PATH_LEVEL]=pfig->GetNumericCode(); //id of 1st figure
cmd.Path1.Array[FIGURA2_PATH_LEVEL]=0 ; // not set
cmd.Path1.Array[FIGURA3_PATH_LEVEL]=0 ; // not set
cmd.Path1.Array[SIMBOLO_PATH_LEVEL]=0; // it is not a symbol
cmd.ArgV[0]=AUTOMATICO;
  //adds the symbol over(SOPRA)/below(SOTTO) the note, AUTOMATICO uses MILLA.

mainScore->SetNetSymCmd(cmd);
mainScore->SetUsrSymCmd(cmd);
mainScore->DoNetSymCmd();
```

```
((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
```

8.8 Horizontal insert commands

CMD_SPARTITO_ADD_LEGATURA
CMD_SPARTITO_ADD_FORC_CRESC
CMD_SPARTITO_ADD_FORC_DIMIN
CMD_SPARTITO_ADD_FRECCIA
CMD_SPARTITO_ADD_ONDA
CMD_SPARTITO_ADD_LEG_QUADRA
CMD_SPARTITO_ADD_CAMBIO_RIT
CMD_SPARTITO_ADD_MODIF8
CMD_SPARTITO_ADD_MODIF15MA
CMD_SPARTITO_ADD_MODIF8BA
CMD_SPARTITO_ADD_MODIF15BA
CMD_SPARTITO_ADD_TRILLOONDA
CMD_SPARTITO_ADD_BEND
...

Path1	starting figure
Path2	ending figure
ArgV[1]	for LEG_QUADRA is the number (used for tuplets)
ArgV[3]	SOPRA/SOTTO/AUTOMATICO
ArgV[4]	for LEG_QUADRA draw the line (TRUE/FALSE)
ArgV[7]	Line type

Example:
The following example adds a slur from the first figure to the last figure of the first measure of the first score:

```
Spartito *psp=mainScore->GetSpartNum(0);
Battuta* pbat=psp->GetBattNum(0);
Figura * pfig1=pbat->GetPLayer(0)->GetFirstFig();
Figura * pfig2=pbat->GetPLayer(0)->GetLastF();

SymCmd cmd;
cmd.CommandId=CMD_SPARTITO_ADD_LEGATURA;
cmd.SelObjType=NOTE_SEL; // select a note
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
cmd.Path1.Array[LAYER_PATH_LEVEL]=1; //1st layer
cmd.Path1.Array[FIGURA1_PATH_LEVEL]=pfig1->GetNumericCode(); //id of 1st figure
cmd.Path1.Array[FIGURA2_PATH_LEVEL]=0 ; // not set
cmd.Path1.Array[FIGURA3_PATH_LEVEL]=0 ; // not set
cmd.Path1.Array[SIMBOLO_PATH_LEVEL]=0; // it is not a symbol

cmd.Path2.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
cmd.Path2.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
cmd.Path2.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
cmd.Path2.Array[LAYER_PATH_LEVEL]=1; //1st layer
cmd.Path2.Array[FIGURA1_PATH_LEVEL]=pfig2->GetNumericCode(); //id of 1st figure
cmd.Path2.Array[FIGURA2_PATH_LEVEL]=0 ; // not set
cmd.Path2.Array[FIGURA3_PATH_LEVEL]=0 ; // not set
cmd.Path2.Array[SIMBOLO_PATH_LEVEL]=0; // it is not a symbol

cmd.ArgV[3]=AUTOMATICO;
//adds the symbol over(SOPRA)/below(SOTTO) the note, AUTOMATICO uses MILLA.
cmd.ArgV[7]=SOLID_LINE;

mainScore->SetNetSymCmd(cmd);
mainScore->SetUsrSymCmd(cmd);
mainScore->DoNetSymCmd();
((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
```

8.9 Delete Command

To delete a figure:

```
Spartito *psp=mainScore->GetSpartNum(0);
Battuta* pbat=psp->GetBattNum(0);
Figura * pfig=pbat->GetPLayer(0)->GetFirstFig();

SymCmd cmd;
```

```
cmd.CommandId= CMD_DEL_SYMBOL;
cmd.SelObjType=ANY_SEL; // select any symbol
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
cmd.Path1.Array[LAYER_PATH_LEVEL]=1; //1st layer
cmd.Path1.Array[FIGURA1_PATH_LEVEL]=pfig->GetNumericCode(); //id of 1st figure
cmd.Path1.Array[FIGURA2_PATH_LEVEL]=0 ; // not set
cmd.Path1.Array[FIGURA3_PATH_LEVEL]=0 ; // not set
cmd.Path1.Array[SIMBOLO_PATH_LEVEL]=0; // it is not a symbol
cmd.Path1.Level=FIGURA1_PATH_LEVEL;

mainScore->SetNetSymCmd(cmd);
mainScore->SetUsrSymCmd(cmd);
mainScore->DoNetSymCmd();
((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
```

To delete a symbol (tenuto):

```
Spartito *psp=mainScore->GetSpartNum(0);
Battuta* pbat=psp->GetBattNum(0);
Figura * pfig=pbat->GetPLayer(0)->GetFirstFig();

SymCmd cmd;
cmd.CommandId= CMD_DEL_SYMBOL;
cmd.SelObjType=ANY_SEL; // select any symbol
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
cmd.Path1.Array[LAYER_PATH_LEVEL]=1; //1st layer
cmd.Path1.Array[FIGURA1_PATH_LEVEL]=pfig->GetNumericCode(); //id of 1st figure
cmd.Path1.Array[FIGURA2_PATH_LEVEL]=0 ; // not set
cmd.Path1.Array[FIGURA3_PATH_LEVEL]=0 ; // not set
cmd.Path1.Array[SIMBOLO_PATH_LEVEL]=CL_TENUTO;
cmd.Path1.Level=SIMBOLO_PATH_LEVEL;

mainScore->SetNetSymCmd(cmd);
mainScore->SetUsrSymCmd(cmd);
mainScore->DoNetSymCmd();
((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
```

To delete an horizontal symbol:

Removes the first horiz. symbol attached to the first figure of the first score:

```
mainScore->SetupIntRefs(); // builds interval references

Spartito *psp=mainScore->GetSpartNum(0);
Battuta* pbat=psp->GetBattNum(0);
Figura * pfig=pbat->GetPLayer(0)->GetFirstFig();
IntRef *pint=pfig->GetIntRef(0);
if(pint!=NULL)
{
      SymCmd cmd;
      cmd.CommandId= CMD_DEL_SYMBOL;
      cmd.SelObjType=ANY_SEL; // select any symbol
      cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
      cmd.Path1.Array[LISTA_PATH_LEVEL]=2; //1=measures list, 2=interval list
      cmd.Path1.Array[SIMBOLO_PATH_LEVEL]=pint->intRef->GetNumericCode();
        // id of the horizontal symbol
      cmd.Path1.Level=SIMBOLO_PATH_LEVEL;

      mainScore->SetNetSymCmd(cmd);
      mainScore->SetUsrSymCmd(cmd);
      mainScore->DoNetSymCmd();
      ((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
}
```

8.10 Delete a measure

To delete the first measure of all the scores:

```
Spartito *psp=mainScore->GetSpartNum(0);
while(psp!=NULL)
{
       Battuta* pbat=psp->GetBattNum(0);

       SymCmd cmd;
       cmd.CommandId=CMD_PARTITURA_DEL_BATTUTA; // the command
       cmd.SelObjType=BATTUTA_SEL; // select a measure
```

```
        cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
        cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
        cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
        cmd.Path1.Level=BATTUTA_PATH_LEVEL;

        mainScore->SetNetSymCmd(cmd);
        //sets the command as given form the network (the other lecterns)
        mainScore->SetUsrSymCmd(cmd);
        //sets the command as given from the user interface
        mainScore->DoNetSymCmd();  // executes the command set in NetSymCmd
        ((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
        // adds the command to the log of commands but it uses also UsrSymCmd...
        psp=mainScore->GetNextSpa(psp);
}
```

8.11 Delete a Score

```
Spartito *psp=mainScore->GetSpartNum(0); //the first score

SymCmd cmd;
cmd.CommandId=CMD_PARTITURA_DEL_SPARTITO; // the command
cmd.SelObjType=SPARTITO_SEL; // select a measure
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode();
cmd.Path1.Level=SPARTITO_PATH_LEVEL;
//cmd.Path1 identifies the score to be deleted

mainScore->SetNetSymCmd(cmd);
//sets the command as given form the network (the other lecterns)
mainScore->SetUsrSymCmd(cmd);
//sets the command as given from the user interface
mainScore->DoNetSymCmd();  // executes the command set in NetSymCmd
((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
// adds the command to the log of commands but it uses also UsrSymCmd...
```

8.12 Consistency Check

The function CheckLogDlg can be used to check the consistency of the score.

```
CheckLogDlg(mainScore);
```

8.13 Cut a Score

The following piece of code can be used to cut a Score.

```
Spartito *psp=mainScore->GetSpartNum(0); //the first score

SymCmd cmd;
cmd.SelObjType=SPARTITO_SEL; // select a measure
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode();
cmd.Path1.Level=SPARTITO_PATH_LEVEL;
//cmd.Path1 identifies the reference score

mainScore->CopyToClipboard(cmd);
mainScore->DelSpartito(cmd);
```

8.14 Copy a Score

```
Spartito *psp=mainScore->GetSpartNum(0); //the first score

SymCmd cmd;
cmd.SelObjType=SPARTITO_SEL; // select a measure
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode();
cmd.Path1.Level=SPARTITO_PATH_LEVEL;
//cmd.Path1 identifies the reference score

mainScore->CopyToClipboard(cmd);
```

8.15 Paste a Score before/after

```
Spartito *psp=mainScore->GetSpartNum(0); //the first score

// adding the copied score before (TRUE) or after (FALSE) psp
mainScore->PasteFromClipboard(psp, TRUE);
```

8.16 Insert label to all scores

```
Spartito *psp=mainScore->GetSpartNum(0);
Battuta* pbat=psp->GetBattNum(0);
```

```
SymCmd cmd;
cmd.CommandId=CMD_BATTUTA_LETTERA; // the command
cmd.SelObjType=BATTUTA_SEL; // select a measure
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
cmd.Path1.Level=BATTUTA_PATH_LEVEL;
strcpy(cmd.Text,"A"); //only one char label.

mainScore->SetNetSymCmd(cmd);
mainScore->SetUsrSymCmd(cmd);
mainScore->DoNetSymCmd();
((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);

//note: this command is applied to all the scores not only to the first one.
```

8.17 Add Metronome

```
Spartito *psp=mainScore->GetSpartNum(0);
while(psp!=NULL)
{
	Battuta* pbat=psp->GetBattNum(0);

	SymCmd cmd;
	cmd.CommandId=CMD_BATTUTA_MOVIMENTO; // the command
	cmd.SelObjType=BATTUTA_SEL; // select a measure
	cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
	cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
	cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
	cmd.Path1.Level=BATTUTA_PATH_LEVEL;
	cmd.ArgV[0]=CL_NCROMA; // note type
	cmd.ArgV[1]=TRUE; // TRUE=with dot
	cmd.ArgV[2]=123; // metronome value
	strcpy(cmd.Text,"Allegro"); //text

	pbat->SymCommand(cmd);

	psp=mainScore->GetNextSpa(psp);
}
```

8.18 Add a dynamic symbol

```
Spartito *psp=mainScore->GetSpartNum(0);
Battuta* pbat=psp->GetBattNum(0);
Figura* pfig=pbat->GetPLayer(0)->GetFirstFig();
if(pfig!=NULL)
{
	Figura* pparent=pfig->GetParentFig();

	SymCmd cmd;
	cmd.CommandId=CMD_FIGURA_FF;
	cmd.SelObjType=FIGURA_SEL; // select a measure
	cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
	cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
	cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
	cmd.Path1.Array[LAYER_PATH_LEVEL]=1; //1st layer
	if(pparent!=NULL) // beam of notes/chords
	{
		cmd.Path1.Array[FIGURA1_PATH_LEVEL]=pparent->GetNumericCode(); //id of beam
		cmd.Path1.Array[FIGURA2_PATH_LEVEL]=pfig->GetNumericCode();  // id of note/chord
		cmd.Path1.Array[FIGURA3_PATH_LEVEL]=0; // not set
	}
	else //note/chord
	{
		cmd.Path1.Array[FIGURA1_PATH_LEVEL]=pfig->GetNumericCode(); //id of note/chord
		cmd.Path1.Array[FIGURA2_PATH_LEVEL]=0;
		cmd.Path1.Array[FIGURA3_PATH_LEVEL]=0; // not set
	}
	cmd.Path1.Array[SIMBOLO_PATH_LEVEL]=0; // it is not a symbol
	cmd.ArgV[0]=AUTOMATICO;    // over(SOPRA)/below(SOTTO) the note, AUTOMATICO uses milla.
	cmd.ArgV[1]=TD_FF;  // the type of dynamics (see "txt/def_txt.hpp" for the list)

	mainScore->SetNetSymCmd(cmd);
	mainScore->SetUsrSymCmd(cmd);
	mainScore->DoNetSymCmd();
	((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
}
```

8.19 Delete a dynamic symbol

```
Spartito *psp=mainScore->GetSpartNum(0);
Battuta* pbat=psp->GetBattNum(0);
Figura* pfig=pbat->GetPLayer(0)->GetFirstFig();
Figura* pparent=pfig->GetParentFig();
if(pfig!=NULL && pfig->GetPTDinamico()!=NULL) // there is a dynamic?
```

```
{
	SymCmd cmd;
	cmd.CommandId=CMD_DEL_SYMBOL;
	cmd.SelObjType=FIGURA_SEL; // select a measure
	cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp->GetNumericCode(); //the score
	cmd.Path1.Array[LISTA_PATH_LEVEL]=1; //1=measures list, 2=interval list
	cmd.Path1.Array[BATTUTA_PATH_LEVEL]=pbat->GetNumericCode(); //id 1st measure
	cmd.Path1.Array[LAYER_PATH_LEVEL]=1; //1st layer
	if(pparent!=NULL) // beam of notes/chords
	{
		cmd.Path1.Array[FIGURA1_PATH_LEVEL]=pparent->GetNumericCode(); //id of beam
		cmd.Path1.Array[FIGURA2_PATH_LEVEL]=pfig->GetNumericCode();  // id of note/chord
		cmd.Path1.Array[FIGURA3_PATH_LEVEL]=0; // not set
	}
	else //note/chord
	{
		cmd.Path1.Array[FIGURA1_PATH_LEVEL]=pfig->GetNumericCode(); //id of note/chord
		cmd.Path1.Array[FIGURA2_PATH_LEVEL]=0;
		cmd.Path1.Array[FIGURA3_PATH_LEVEL]=0; // not set
	}
	cmd.Path1.Array[SIMBOLO_PATH_LEVEL]=pfig->GetPTDinamico()->GetID(); // id of symbol

	mainScore->SetNetSymCmd(cmd);
	mainScore->SetUsrSymCmd(cmd);
	mainScore->DoNetSymCmd();
	((MaMass*)mainScore)->InserisciComandoACLMaMass(&cmd);
}
```

8.20 Split layers

```
//extract a layer to a new score
// extracting a layer of part 1 to a new score.
Spartito *psp=mainScore->GetSpartNum(0);
mainScore->SplittingPart(psp);
```

8.21 Merge score layers

```
// it merges the layers of 2 parts, it works only if are single staff parts
// and if only two layers are used in each score.

int part[3]={-1,-1,-1}; // this initialization is very important!!!!
if(MutipleChoiceDial(part, mainScore, 2))
{
	if(!mainScore->MergingParts(part[0],part[1]))
		Message("parts can't be merged");
}
```

8.22 Join 2/3 score in a multi-staff score

```
// joining two/three single staff scores in one multi staff score

int part[3]={-1,-1,-1}; // this initialization is very important!!!!
if(MutipleChoiceDial(part, mainScore, 3))
{
	if(!mainScore->JoiningParts(part[0],part[1],part[2]))
		Message("parts can't be joined");
}
```

8.23 Disjoin a multi-staff score

```
// disjoining one multi staff score in two/three single staff scores
Spartito *psp=mainScore->GetSpartNum(0);
if(psp->GetNumberOfStaff()>1)
	mainScore->DisjoiningPart(psp);
```

8.24 Move a part

```
Spartito *psp1=mainScore->GetSpartNum(0);
Spartito *psp2=mainScore->GetSpartNum(1);

SymCmd cmd;
cmd.CommandId=CMD_PARTITURA_MOVE_PART; // the command
cmd.SelObjType=SPARTITO_SEL; // select a score
cmd.Path1.Array[SPARTITO_PATH_LEVEL]=psp1->GetNumericCode(); //the score to be moved
cmd.Path1.Level=SPARTITO_PATH_LEVEL;
cmd.Path2.Array[SPARTITO_PATH_LEVEL]=psp2->GetNumericCode(); //the score after which score1 is moved
cmd.Path2.Level=SPARTITO_PATH_LEVEL;

mainScore->SymCommand(cmd);
```

8.25 Move a layer

```
Spartito *psp=mainScore->GetSpartNum(0);
```

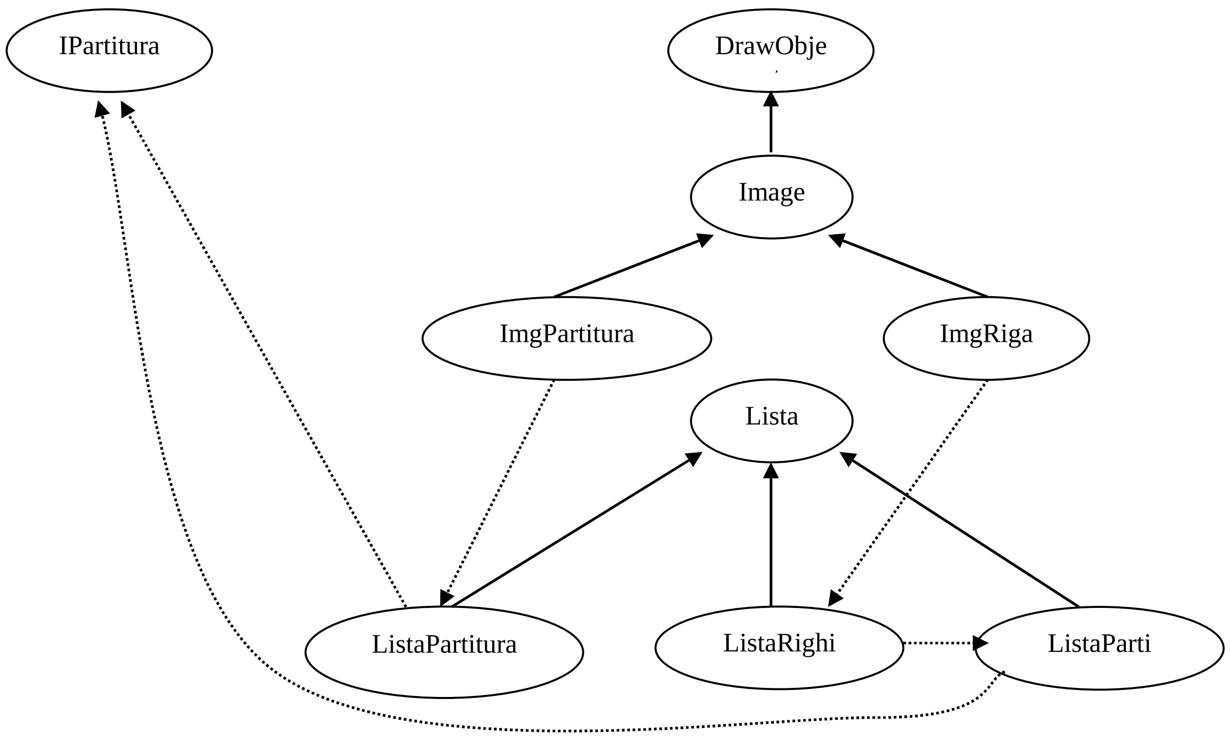
```
int numLay=psp->GetNumberOfStaff()*4;
int from, to;
if(MoveLayerDial(from, to, numLay))
       psp->SwitchLayer(from,to);
```

8.26 Transposition

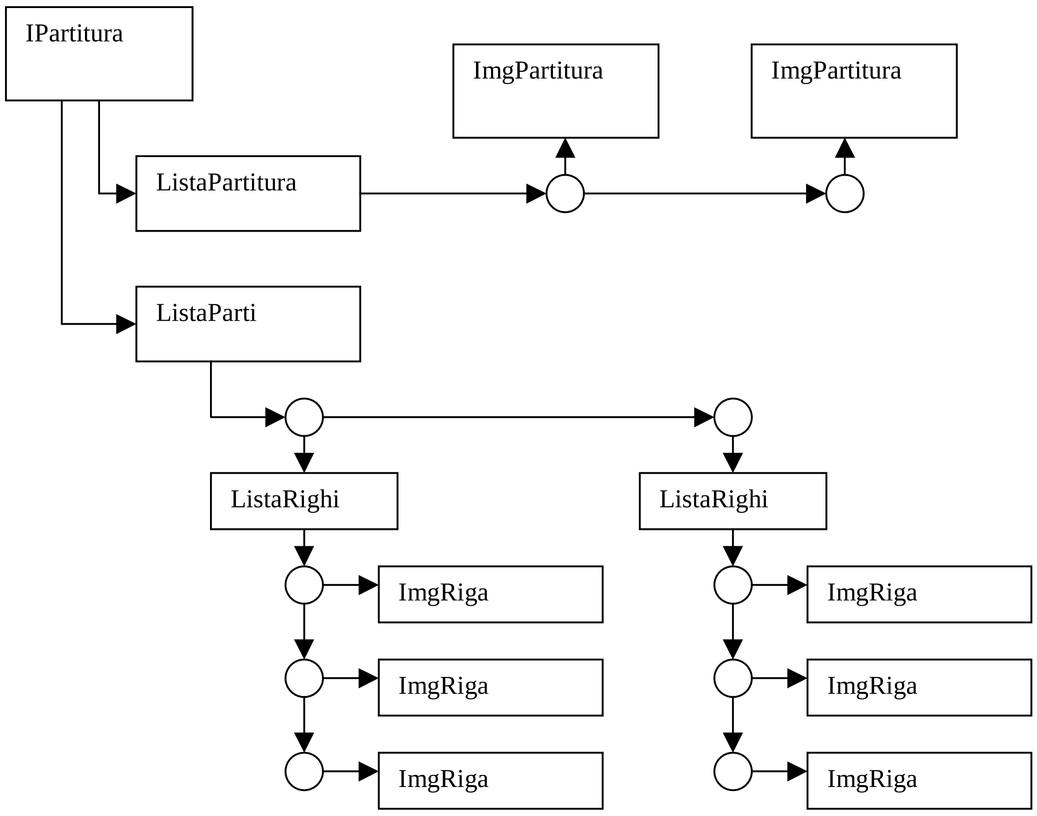
```
int clef=-1;
int partnumber=0;
wxString measures;
int    translation=0;
int interval=0;
int up=0;
int current_Score;
int numofstaff=0;
Bool sharps=FALSE;
Bool adjust=FALSE;

ListaSpartiti *listaSpa=mainScore->GetListaSpa();

current_Score=listaSpa->GetCurrentScore();
if(TranspositionsDlg(&clef, &measures, &translation, &interval, &up, &current_Score, &numofstaff,
mainScore, &sharps, &adjust))
{
       int daBat, aBat;
       if(measures.Find('-')==-1)
       {
              daBat=aBat=atoi(measures.c_str());
              if(daBat==0)
                     daBat=1;
       }
       else
       {
              daBat=atoi(measures.BeforeFirst('-').c_str());
              aBat=atoi(measures.AfterFirst('-').c_str());
       }
       listaSpa->Transpose(daBat,aBat,current_Score,clef,translation,interval,up,numofstaff,sharps,
adjust);
}
```

9 WEDEL-OOMM Score Images

The management of score images is done with the following classes:
Class *Image* that is derived from *DrawObject* that have the funcionalities to load a GIF image, and methods to draw the image in a canvas.
Class *ImgPartitura* represents the image of a Main Score and have as attributes the starting and ending measure present in the image. Class ImgRiga represents a row of the score for a single part, and have the attributes of the strating and the ending measures of the image.
Class *ListaPartitura* is the list collecting all the images of the main score. While *ListaRighi* contains all the rows of a single part. Class *ListaParti* is used to collect the *ListaRighi* objects of all the parts of a main score. Moreover class *IPartitura* collects the main score images and the parts images.



The following is an object diagram:



10 Relationships among WEDEL Editor Classes and those of the WEDEL-OOMM

Class name	English Name	Managed Symbols or Music Structures
Abbellimento	Ornament	Generic Ornament
Accento	Accent	Accent Expression
Accordo	Chord	Chords singles and in beams
Alterazione	Accidental	A generic Accidental
AlterazioneComposta	ComposedAccidental	A Composite Accidental
Ancoraggio	Anchorage	To fix an horizontal symbol withou having the needs of a figure: note, rest. The anchorage has not time/space duration for the justification and drawing.
Annotazione	Annotation	Conductor annotation
Arco	Bow	Generic Bow direction: Up, Down
ArcoGiu	Bow Down	Bow Down
ArcoGiuFinoA	BowDownTillTo	Bow down for violin with an arrow
ArcoSu	Bow Up	Bow Up
ArcoSuFinoA	BowUpTillTo	Bow up for violin with an arrow
ArmaturaChiave	KeySignature	Key Signature
Armonici	Harmonics	Both the harmonics
ArpaPedal	HarpPedal	Harp Pedal
Arpeggio	Arpeggio	Arping for strings
Barra	Barline	Generic Barline of the measure
Battuta	Measure	The measure
BDashed	DashedBarline	Dashed Barline
BDoppia	DoubleBarline	Double Barline
Bemolle	Flat	Flat
Bemolle 1Q	Flat1Q	A quarter flat
Bemolle 3Q	Flat3Q	Three quarter flat
Bend	Bend	Bend for guitar, a square slur
Bequadro	Natural	Natural
Bequadro 1Q	Natural1Q	A quarter natural
Bequadro 3 Q	Natural3Q	Three quarter natural
BFinale	FinalBarline	FinalBarline
Bfine Rit	FinalRepeatSign	End of Refrain
BInizioFine	StartFinalRepeatSign	Start and end refrain
BInizioRit	Start Repeat Sign	Start Refrain
BInvisibile	Invisible Barline	Invisible Barline
BSingola	Single Barline	Single Barline
CambioRit	ChangeRefrain	Change Refrain
CBaritono	BaritoneClef	Baritone Clef
CBasso	BassClef	Bass Clef
CBasso8	BassClef8VB	Bass Clef 8 VB
CBasso8Sopra	BassClef8VA	Bass Clef 8 VA
CBassoOld	Old CClef	Old C Clef
CContralto	AltoClef	Alto Clef
Chiave	Clef	A Generic Clef
CMezzoSoprano	MezzoSopranoClef	Mezzo Soprano Clef
Coda	Coda	The symbol of Coda
ConSord	WithMute	WithMute
Corda	String	Numbers for Strings
Corona	Fermata	Woith several differen shapes
CPercus2lines	Percussion2Lines	Percussion 2 Lines Clef
CPercusBox	PercussionBox	Percussion Box Clef
CSoprano	SopranoClef	Soprano Clef
CTab	TablatureClef	Tablature Clef

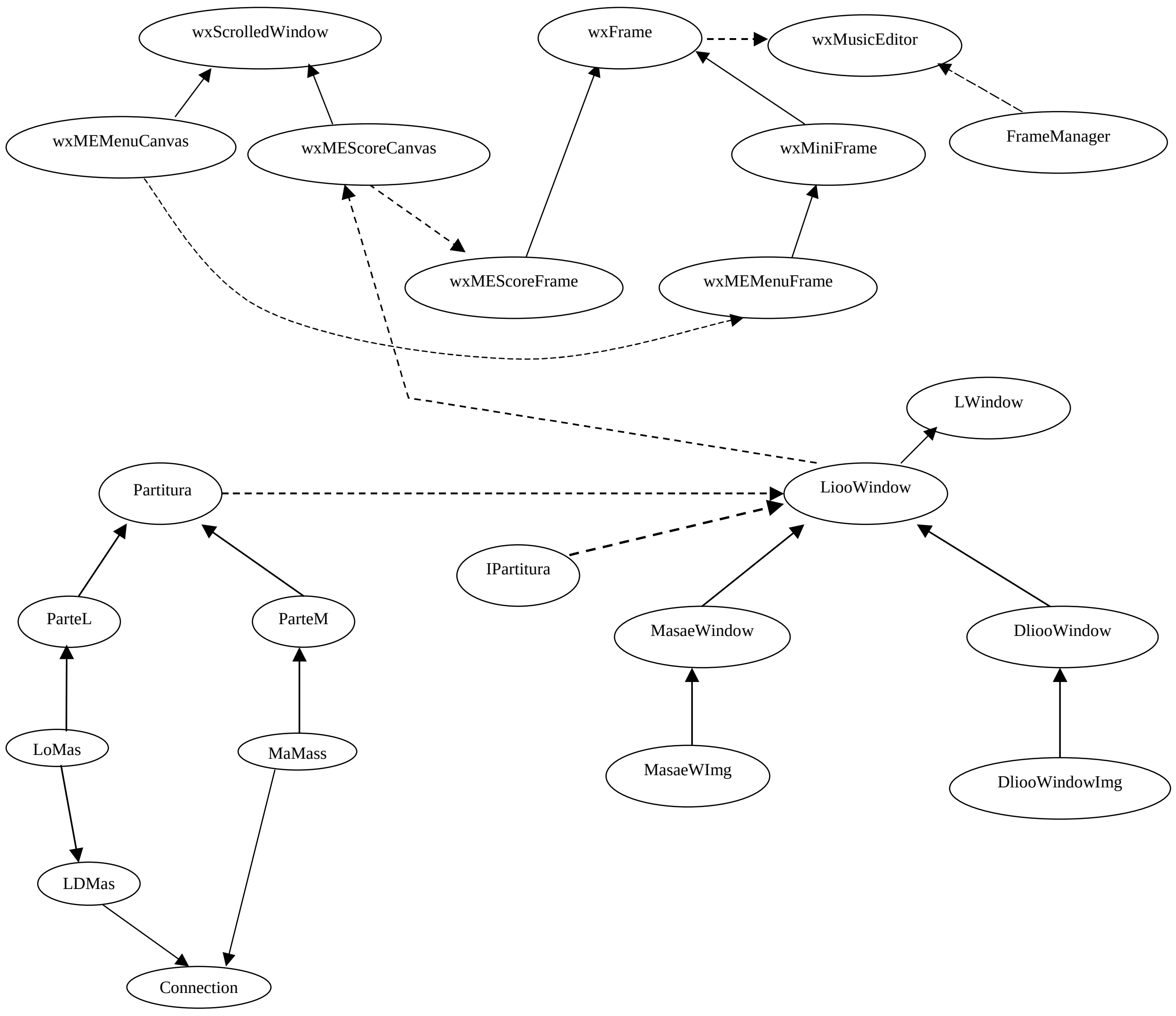
CTenore	TenorClef	Tenor Clef
CTenore8	TenorClef8VB	Tenor Clef 8 VB
CViolino	TrebleClef	Treble Clef
CViolino8	TrebleClef8VB	Treble Clef 8VB
CViolino8Sopra	TrebleClef8VA	Treble Clef 8 VA
CVuota	EmptyClef	Empty Clef
DaCapo	DaCapo	D. C.
Dal Segno	DalSegno	D. S.
Diesis	Sharp	Sharp
Diesis 3Q	Sharp3Q	Three quarter sharp
Diesis1Q	Sharp1Q	A quarter Sharp
DirezArco	BowDirection	BowDirection
DitaCorde	FretBoard	Freatboard
DitaFile	FretBoardFile	Freadborad file database
Diteggiato	Fingering	Fingering numbers
Doppio Bemolle	DoubleFlat	Double flat
Doppio Diesis	DoubleSharp	Double sharp
EspGenerica	GenericExpression	Generic Symbol/Expression
EspressComposta	ComposedExpression	Composite expression
Espressione	Expression	Generic Expression
Figura	Figure	Superclass of figures
Forcella	Fork	Crescendo, decrescendo
ForcellaEstesa	ExtendedFork	Extended crescendo and descrescendo
Freccia	Arrow	Arrow
GInferiore	InferiorGroup	Inferior Turn
Glissato	Glissando	Glissando
Gruppetto	SmallGroup	Turn
GruppoNote	Beam	Beam of figures
GSlash	GroupSlash	Slash Turn
GSuperiore	SuperiorGroup	Superior Turn
GUp	GroupUp	Up Turn
Indicazione	Indication	Indications
Intervallo	Interval	A generic Horizontal symbol
Intestazione	Heading	Generic Container of Measure Header
IntEsteso	ExtendedInterval	Extended Horizontal/Interval Symbol
Layer	Layer	The model of Layer
LegatQuadra	SquareSlur	A bend, square slur with a number (terzine)
Legatura	Slur	Slur
LegatValore	Tie	Tie
Lettera	Letter	A letter
Lista	List	List generic
ListaBattute	MeasuresList	List of Measues as seen by the Part (spartito)
ListaFigure	FigureList	List of Figures
ListaIntEst	ExtendedIntervalList	List of Extended Interval for horizontal symbols
ListaParentesi	BracketsList	List of Brackets for grouping staffs
ListaSillabe	SyllableList	List of Syllables for Lyric
ListaSpartiti	ScoresList	The list of Parts as seen by the Main Score (Partitura)
MartDolce	SweetMartellato	Sweet hammered Expression
Martellato	Martellato	Hammered Expression
Metronomo	Metronome	The metronomic indication
MInferiore	InferiorMordent	Inferior Mordent
ModifOttava	ModifyOctave	Change of Octave in several forms
Mordente	Mordent	Mordent
Movimento	Movement	Movement indication
MSuperiore	SuperiorMordent	Superior Mordent
NBiscroma	32thN	32th Note

NBreve	BreveN	Breve Note
NCroma	8thN	8th Note
NFusa	128thN	128th Note
NMinima	HalfN	Half Note
Nota	Note	Superclass of notes
NSemibiscroma	64thN	64th Note
NSemibreve	WholeN	Whole Note
NSemicroma	16thN	16th Note
NSemiminima	QuarterN	Quarter Note
NumBattuta	MeasureNumber	Number of Measure
Numgrande	BigNumber	A number written with a big font
NumPausa	RestNumber	The number on the generic rest
NumUguale	EqualNumber	The number of repeated measures
Occhiali	Glasses	To get attention
Onda	Wave	The symbol of Wave
OrganPedal	OrganPedal	Horgan Pedal
Parentesi	Bracket	Generic singe bracket for grouping staffs
ParGraffa	Brace	Brace bracket for grouping staffs
ParQuadra	SquareBracket	Square bracket for grouping staffs
ParteArco	BowPart	Generic part of the arc for violin
Partitura	Main Score	The main score
Pausa	Rest	Superclass of rets
PBiscroma	32thR	32th Rest
PCroma	8thR	8th Rest
PDueBatt	TwoMeasuresRest	Two Measures Rest
PedalDown	PedalDown	Pedal Down
Pedale	Pedal	Generic Pedal
PedaleFinoA	PedalTillTo	Pedal up to for Piano
PedalUp	PedalUp	Pedal Up
Pentagramma	Staff	The support for notes from 1 to 7 lines
Percussione	Percussion	Generic Percussion
PFusa	128thRest	128th Rest
PGenerica	GenericRest	Generic Rest
PianoPedal	PianoPedal	Piano Pedal
Pizzicato	Pizzicato	Pizz for Violin
PMinima	HalfR	Half Rest
Ponticello	Bridge	Pont. For Violin
PosizArco	BowPosition	Generic Position of Arc for Violin
PQuattroBatt	FourMeasR	Four Measure Rest
PrFiato	Breathe	To get a breath
PSemibiscroma	64thR	64th Rest
PSemibreve	WholeR	Whole Rest
PSemicroma	16thR	16th Rest
PSemiminima	QuarterR	Quarter Rest
Punta	Punta	Punta for violin
PuntoAllungato	PuntoAllungato	Punto Allungato Expression
PuntoValore	AugmentationDot	Augmentation dot
Riferimento	Reference	A reference symbol (D.C.,D.S., Label, etc.)
RipBatt	RepeatMeasure	Repeat Measure
RipBattN	RepeatMeasureNTimes	Repeat Measure N Times, with a specified N
Ripetizione	Repetition	Generic Repeat Symbol
RipmezzBatt	RepeatHalfMeasure	Repeat Half Measure
RipTempo	RepeatTime	Repeat Time
Salto	Jump	A Jump point
Scansione	Scansion	|||| symbol
Segno	Sign	Marker used for repetitions (Dal Segno)

Sforzato	Sforzato	Sforzato Expression
Sillaba	Syllable	For Lyric
Sordina	Mute	Mute
Spartito	Part Score	The single Part
Staccato	Staccato	Staccato Expression
Strumento	Instrument	Generic Instrument
Tallone	Tallone (Heel)	Tallone for Violin
Tastiera	Keyboard	Keyboard for violin
TDinamico	DynamicText	A text associated to a figure
Tempo	Time	Time Signature
Tenuto	Tenuto	Tenuto Expression
TGenerico	GenericText	A text associated to a figure
Timpano	Timpani	Percussion regulation for changing tone
TMovimento	MovementText	Textual indication of movement
TNumerico	NumericText	text
Tremolo	Tremolo	Tremolo
Trillo	Trill	Trill
TrilloOnda	TrillWave	Trill with a following wave
VariazioneValore	ChangeofValue	Generic change of figure duration
ViaSordina	WithoutMute	WithoutMute
Violino	Violin	Generic Violin Symbols

11 WEDEL Music Editor Object Oriented Model

The classes used for user interface are shown in the following class diagram:



The music editor will open many window frames, frames for music visualization/editing and some mini frames for tool palettes. Class wxMEScoreFrame derived from wxFrame has been introduced to represent a music frame while wxMEMenuFrame derived from wxMiniFrame represents a tool palette. Since the music editor can open many views of the same score (the main score view, and one view for each part) class FrameManager has been introduced to manage the views opened and also to manage the frames of the tool palettes opened. The client area of the frames are wxMEScoreCanvas for the score frames and wxMEMenuCanvas for the menu frames. Class wxMusicEditor represents the music editor and has an instance of the FrameManager to handle all the frames of the editor.
Class *LiooWindow* represents a virtual window containng a symbolic score (Partitura) or an image score (IPartitura). *MasaeWindow* represents a window with a symbolic main score while *DliooWindow* represents a window with a single part score. Moreover *MasaeWImg* and *DliooWindowImg* have been derived to handle also image scores. Class wxMEScoreCanvas has a pointer to a object derived from LiooWindow, in this way each score frame can have, a symbolic main score, a symbolic part, an image main score or an image part.

12 ABB module

The ABB module is composed from a set of classes; each one implements a musical symbol of ornament. The Ornaments considered are:

- Tremolo;
- Trill;
- Inferior mordent;
- Superior mordent
- Inferior small group;
- Superior small group.

The ornaments refers to notes or chords and thus in the classes that implement such musical figures we will refer to them. All the ornaments are children classes of a generic class Abbellimento that descends from DrawObject. In fact each ornament is an object that can be drawn. Therefore it inherits from DrawObject, through Abbellimento, all the necessary methods, redefining each time the Draw method. The classes that belong to this module are:

- Abbellimento;
- Tremolo;
- Trillo;
- Mordente;
- MInferiore;
- MSuperiore;
- Gruppetto
- GInferiore;
- GSuperiore.

12.1 Class Abbellimento

12.1.1 Description

This is an abstract class that has the purpose of representing all the ornaments that can appear on a musical score. The ornaments are sounds or groups of auxiliary sounds whose function is adorning the main sound, giving him a particular expression or importance in the musical speech.

12.1.2 Father class

DrawObject

12.1.3 Children classes

Tremolo, Trillo, Mordente, Gruppetto

12.1.4 Protected attributes

aboveNota

Boolean that defines if the ornament will be drawn above or below the figure that it refers to.

12.1.5 Public methods

Bool GetAboveNota()

It returns the Boolean that defines the position of the ornament above or below the figure it refers to.

void SetAboveNota(Bool above)

It sets the Boolean attribute aboveNota TRUE if the ornament goes above or FALSE if it goes below the figure it refers to.

void Draw()

It draws the ornament.

char *Describe(Context)

12.2 Class Tremolo

12.2.1 Description

This class allows designing the tremolo symbol, that consists of the alternation of a certain number of dashes with a variable gradient posed between two adjacent notes. This musical symbol shows the rapid alternating, rhythmically uniform and prolonged, of the sound. Differently from the other ornaments, it is managed by the class Battuta: this entails that it would be impossible the presence of a tremolo symbol between two notes of two different measures even if they were adjacent.

12.2.2 Father class

Abbellimento

12.2.3 Children classes

NONE.

12.2.4 Protected attributes

short NBarre

Integer that represents the number of dashes of the tremolo.

Point PointTo

Final point of the dash.

VUnit VU2Up, VU2Dwn, VU2Lft, VU2Rgt

Distances from AbsPos of the superior, inferior, left and right points of the rectangle containing the figure.

12.2.5 Public methods

Tremolo()

Initialiser: it defines the identifier of the class initialising the number of dashes to zero.

void SetNumTrem(short nb)

It sets NBarre at nb.

short GetNumBarre()

It returns NBarre.

void SetPos(const Point& Plft,const Point& Rgt)

It sets the extremities of the dash according to the points PLft and PRgt.

VUnit GetVU2Up()

It returns the distance from AbsPos to the superior point.

VUnit GetVU2Dwn()

It returns the distance from AbsPos to the inferior point.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the right point.

VUnit GetVU2Lft()

It returns the distance from AbsPos to the left point.

void Draw()

It draws the Tremolo.

char *Describe(Context)

12.3 Class Trillo

12.3.1 Description

This class allows drawing the Trillo symbol. The Trillo consists of the rapid alternating, rhythmically uniform and prolonged, of a sound with the immediately superior and, rarely, inferior one. The symbol of the trill is *tr* eventually followed by a certain number of waves. A limit has been imposed to the number of waves (4).

12.3.2 Father class

Abbellimento

12.3.3 Children classes

NONE.

12.3.4 Private attributes

numtrilli

Integer that represents the number of waves of the trill.

12.3.5 Public Methods

Trillo()

Initialiser: it sets the Trillo above the figure to which it refers with a number of waves equal to zero and defines the identifier of the class.

void SetNumTrilli(int ntrilli)

It sets the number of waves of the trill.

VUnit GetVU2Up()

It returns the distance from the centre of the Trillo symbol to the extreme superior point of the symbol itself.

VUnit GetVU2Dwn()

It returns the distance from the centre of the Trillo symbol to the extreme inferior point of the symbol itself.

VUnit GetVU2Rgt()

It returns the distance from the centre of the Trillo symbol to the extreme right point of the symbol itself.

VUnit GetVU2Lft()

It returns the distance from the centre of the Trillo symbol to the extreme left point of the symbol itself.

void Draw()

It draws the Trillo.

char *Describe(Context)

12.4 Class Gruppetto

12.4.1 Description

This class is a generic class for designing the Gruppetto from which the classes GInferiore and GSuperiore has been specialised. The names of these two classes do not mean that one has to be drawn below and the other above the note. They simply refer to two different kind of Gruppetto.

12.4.2 Father class

Abbellimento

12.4.3 Children classes

GInferiore, GSuperiore

12.4.4 Public methods

void Draw()

It draws the Gruppetto.

12.5 Class GInferiore

12.5.1 Description

This class allows designing the Gruppetto Inferiore symbol that on the score appears as an horizontal S; musically consists in a brief melodic inflexion due to the alternating of the main sound with the sound immediately nearby, superior and inferior, starting from the inferior one.

12.5.2 Father class

Gruppetto

12.5.3 Children classes

NONE.

12.5.4 Public methods

GInferiore()

Initialiser: it sets the Gruppetto Inferiore above the figure it refers to and defines the identifier of the class.

VUnit GetVU2Up()

It returns the distance from the centre of the Gruppetto Inferiore symbol to the extreme superior point of the symbol itself.

VUnit GetVU2Dwn()

It returns the distance from the centre of the Gruppetto Inferiore symbol to the extreme inferior point of the symbol itself.

VUnit GetVU2Rgt()

It returns the distance from the centre of the Gruppetto Inferiore symbol to the extreme right point of the symbol itself.

VUnit GetVU2Lft()

It returns the distance from the centre of the Gruppetto Inferiore symbol to the extreme left point of the symbol itself.

void Draw()

It draws the Gruppetto Inferiore.

char *Describe(Context)

12.6 Class GSuperiore

12.6.1 Description

This class allows designing the Gruppetto Superiore symbol that on the score appears as an horizontal S in a mirror-like position with respect to Gruppetto Inferiore; musically consists in a brief melodic inflexion due to the alternating of the main sound with the sound immediately nearby, superior and inferior, starting from the superior one.

12.6.2 Father class

Gruppetto

12.6.3 Children classes

NONE.

12.6.4 Public methods

GSuperiore()

Initialiser: it sets the Gruppetto Superiore above the figure it refers to and defines the identifier of the class.

VUnit GetVU2Up()

It returns the distance from the centre of the Gruppetto Superiore symbol to the extreme superior point of the symbol itself.

VUnit GetVU2Dwn()

It returns the distance from the centre of the Gruppetto Superiore symbol to the extreme inferior point of the symbol itself.

VUnit GetVU2Rgt()

It returns the distance from the centre of the Gruppetto Superiore symbol to the extreme right point of the symbol itself.

VUnit GetVU2Lft()

It returns the distance from the centre of the Gruppetto Superiore symbol to the extreme left point of the symbol itself.

void Draw()

It draws the Gruppetto Superiore.

char *Describe(Context)

12.7 Class Mordente

12.7.1 Description

This class is a generic class for designing the Mordente from which the classes MInferiore and MSuperiore has been specialised. Also in this case the names of these two classes do not mean that one has to be drawn below and the other above the note. They simply refer to two different kind of Mordente.

12.7.2 Father class

Abbellimento

12.7.3 Children classes

MInferiore, MSuperiore

12.7.4 Public methods

void Draw()

It draws the Mordente.

12.8 Class MInferiore

12.8.1 Description

This class allows designing the Mordente Inferiore symbol. It consists in an instantaneous alternating of a main sound with the sound immediately inferior; it is indicated with a characteristic winding dash vertically cut by means of a bar line.

12.8.2 Father class

Mordente

12.8.3 Children classes

NONE.

12.8.4 Public methods

MInferiore()

Initialiser: it sets the Mordente Inferiore above the figure it refers to and defines the identifier of the class.

VUnit GetVU2Up()

It returns the distance from the centre of the Mordente Inferiore symbol to the extreme superior point of the symbol itself.

VUnit GetVU2Dwn()

It returns the distance from the centre of the Mordente Inferiore symbol to the extreme inferior point of the symbol itself.

VUnit GetVU2Rgt()

It returns the distance from the centre of the Mordente Inferiore symbol to the extreme right point of the symbol itself.

VUnit GetVU2Lft()

It returns the distance from the centre of the Mordente Inferiore symbol to the extreme left point of the symbol itself.

void Draw()

It draws the Mordente Inferiore.

char *Describe(Context)

12.9 Class MSuperiore

12.9.1 Description

This class allows designing the Mordente Superiore symbol. It consists in an instantaneous alternating of a main sound with the sound immediately superior; it is indicated with a characteristic winding dash.

12.9.2 Father class

Mordente

12.9.3 Children classes

NONE.

12.9.4 Public methods

MSuperiore()

Initialiser: it sets the Mordente Superiore above the figure it refers to and defines the identifier of the class.

VUnit GetVU2Up()

It returns the distance from the centre of the Mordente Superiore symbol to the extreme superior point of the symbol itself.

VUnit GetVU2Dwn()

It returns the distance from the centre of the Mordente Superiore symbol to the extreme inferior point of the symbol itself.

VUnit GetVU2Rgt()

It returns the distance from the centre of the Mordente Superiore symbol to the extreme right point of the symbol itself.

VUnit GetVU2Lft()

It returns the distance from the centre of the Mordente Superiore symbol to the extreme left point of the symbol itself.

void Draw()

It draws the Mordente Superiore.

char *Describe(Context)

13 ALT module

In this module all the accidentals to the musical notes are implemented. An accidental can be positioned on the left of a note (and in this case it has an effect on this one only) or at the beginning of the measure in the key signature (and has en effect on all the notes of the measure). This module is thus in connection with the FIG module (by means of a IRB relationship between the class Alterazione Estesa and the class Nota) and with the BAT module (by means of a IRB relationship between the class Alterazione and the class ArmaturaChiave), besides, obviously with the DRW module (through a ISA relationship between the class AltrazioneEstesa and the class DrawObject).
The accidentals that can be placed on the left of the musical notes can be both simple (sharp, flat, natural) and double (double sharp, double flat). In the key signature, on the other hand, only the simple accidentals are used. This different use of the accidentals has made us differentiate the simple and double accidentals in the implementation. The class AlterazioneEstesa is the most general and is father of the double accidentals and of an intermediate class (Alterazione) that collects the simple accidentals; thus AlterazioneEstesa will be connected to the class Nota, whereas Alterazione to ArmaturaChiave.

13.1 Class Alterzione Estesa

13.1.1 Description

This is an abstract class and is the principal one of the module, which has the purpose to represent all the accidentals of the musical notes. This class connects (through a IRB relationship with Nota class) the ALT module to the FIG module.

13.1.2 Father class

DrawObject

13.1.3 Children classes

Alterazione, DoppioDiesis, DoppioBemolle

13.1.4 Protected attributes

int Altezza

Musical height of the accidental with respect to the note that is placed on the lower line of the staff (Mi in the tremble clef).

Bool Notina

Size of the accidental in the two types of note and small note. It is TRUE if the accidental refers to a small note.

13.1.5 Public methods

AlterazioneEstesa()

Constructor of the class the initialises the attribute Notina=FALSE and the attribute Altezza=0.

void SetAltezza(int)

It sets the musical height.

void SetNotina(Bool)

It sets the size.

Bool GetNotina()

It returns the size.

void SetPos(DrawObject* d,const Point& p)

It sets the absolute position relatively to the drawobject d. If d=NULL it sets the absolute position as ‘p’.

void Draw()

It draws the object in AbsPos.

virtual char *Describe(Context)

It returns a MusicTeX description, he wants the offset in the entry to be imposed to the height according to the clef of the measure.

13.2 Class Alterazione

13.2.1 Description

This is an abstract class that has the purpose to represent all the simple accidentals of the musical notes and acquires nearly all the functionalities of the AlterazioneEstesa class. This class connects (by means of a IRB relationship with the ArmaturaChiave class) the ALT module with the BAT module.

13.2.2 Father class

AlterazioneEstesa

13.2.3 Children classes

Diesis, Bemolle, Bequadro

13.2.4 Public methods

void Draw()

It draws the object in AbsPos.

13.3 Class DoppioDiesis

13.3.1 Description

This is a symbol class that represents the double sharp accidental symbol (that increases the note's height of a tone).

13.3.2 Father class

AlterazioneEstesa

13.3.3 Public method

DoppioDiesis()

Constructor of the class that initialises the number ID that identifies the type of graphic object.

VUnit GetVU2Up()

It returns the distance from AbsPos to the superior point differentiating the cases in which the accidentals refers to a note or to a small note.

VUnit GetVU2Dwn()

It returns the distance from AbsPos to the inferior point differentiating the cases in which the accidentals refers to a note or to a small note.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the point on the extreme right differentiating the cases in which the accidentals refers to a note or to a small note.

VUnit GetVU2Lft()

It returns the distance from AbsPos to the point on the extreme left differentiating the cases in which the accidentals refers to a note or to a small note.

void Draw()

It draws the object in AbsPos differentiating the cases in which the accidentals refers to a note or to a small note..

char *Describe(Context)

13.4 Class DoppioBemolle

13.4.1 Description

This is a symbol class that represents the double flat accidental symbol (that decreases the note's height of a tone).

13.4.2 Father class

AlterazioneEstesa

13.4.3 Public method

DoppioBemolle()

Constructor of the class that initialises the number ID that identifies the type of graphic object.

VUnit GetVU2Up()

It returns the distance from AbsPos to the superior point differentiating the cases in which the accidentals refers to a note or to a small note.

VUnit GetVU2Dwn()

It returns the distance from AbsPos to the inferior point differentiating the cases in which the accidentals refers to a note or to a small note.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the point on the extreme right differentiating the cases in which the accidentals refers to a note or to a small note.

VUnit GetVU2Lft()

It returns the distance from AbsPos to the point on the extreme left differentiating the cases in which the accidentals refers to a note or to a small note.

void Draw()

It draws the object in AbsPos differentiating the cases in which the accidentals refers to a note or to a small note..

char *Describe(Context)

13.5 Class Diesis

13.5.1 Description

This is a symbol class that represents the sharp accidental symbol (that increases the note's height of an half-tone).

13.5.2 Father class

Alterazione

13.5.3 Public method

Diesis()

Constructor of the class that initialises the number ID that identifies the type of graphic object.

VUnit GetVU2Up()

It returns the distance from AbsPos to the superior point differentiating the cases in which the accidentals refers to a note or to a small note.

VUnit GetVU2Dwn()

It returns the distance from AbsPos to the inferior point differentiating the cases in which the accidentals refers to a note or to a small note.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the point on the extreme right differentiating the cases in which the accidentals refers to a note or to a small note.

VUnit GetVU2Lft()

It returns the distance from AbsPos to the point on the extreme left differentiating the cases in which the accidentals refers to a note or to a small note.

void Draw()

It draws the object in AbsPos differentiating the cases in which the accidentals refers to a note or to a small note..

char *Describe(Context)

13.6 Class Bemolle

13.6.1 Description

This is a symbol class that represents the flat accidental symbol (that decreases the note's height of an half-tone).

13.6.2 Father class

Alterazione

13.6.3 Public method

Bemolle()

Constructor of the class that initialises the number ID that identifies the type of graphic object.

VUnit GetVU2Up()

It returns the distance from AbsPos to the superior point differentiating the cases in which the accidentals refers to a note or to a small note.

VUnit GetVU2Dwn()

It returns the distance from AbsPos to the inferior point differentiating the cases in which the accidentals refers to a note or to a small note.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the point on the extreme right differentiating the cases in which the accidentals refers to a note or to a small note.

VUnit GetVU2Lft()

It returns the distance from AbsPos to the point on the extreme left differentiating the cases in which the accidentals refers to a note or to a small note.

void Draw()

It draws the object in AbsPos differentiating the cases in which the accidentals refers to a note or to a small note..

char *Describe(Context)

13.7 Class Bequadro

13.7.1 Description

This is a symbol class that represents the natural accidental symbol (that deletes the effect of the simple accidentals sharp and flat bringing the note back to its natural status).

13.7.2 Father class

Alterazione

13.7.3 Public method

Bequadro()

Constructor of the class that initialises the number ID that identifies the type of graphic object.

VUnit GetVU2Up()

It returns the distance from AbsPos to the superior point differentiating the cases in which the accidentals refers to a note or to a small note.

VUnit GetVU2Dwn()

It returns the distance from AbsPos to the inferior point differentiating the cases in which the accidentals refers to a note or to a small note.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the point on the extreme right differentiating the cases in which the accidentals refers to a note or to a small note.

VUnit GetVU2Lft()

It returns the distance from AbsPos to the point on the extreme left differentiating the cases in which the accidentals refers to a note or to a small note.

void Draw()

It draws the object in AbsPos differentiating the cases in which the accidentals refers to a note or to a small note..

char *Describe(Context)

14 AUDIO module

Audio module is the part of the project which manages and plays audio files.

14.1 Class AudioThread

14.1.1 Description

AudioThread is a class you can use to play wave files within an application. It is derived from wxThread which is the main wx class made to manage threads.

Since AudioThread uses some A.P.I. functions, you can’t use it for Linux applications.

14.1.2 Father class

wxThread

14.1.3 Protected attributes

WaveMultiRatePlayerOnThread *player
Player attribute points at the wave file player and processor.
wxString audioFile
It’s the name of the wave file to be processed and played.
int beginBlock
The thread starts playing the file from this block. Each file is subdivided into fragments of samples (blocks), if beginBlock is 0, the execution starts from the beginning of the track. See also …

14.1.4 Public methods

AudioThread(wxString UaudioFile, int UbeginBlock)
Class constructor. It takes two parameters: the wave file name and the beginning block.
wxThreadError Run()
It starts the execution of the thread, hence it starts the playing. Call this method from the main thread context after creating the audio thread by its constructor.
virtual void *Entry( )
It is the body of the thread. You don’t have to call this method, Run calls it automatically.
void Stop()
It stops playing the file, hence it terminates the thread.
void Pause()
It pauses the playing. During the pause state of the player, the audio thread sleeps and only the main application is under execution.
void Play()
If the player is in pause state, it restores the playing state, hence the file execution starts again.
int SetRate(int rate)
Sets the playing rate (the “speed” of the track execution). If the rate is 1 the PC execution goes 1% slower than the original one, is the rate is –1 it goes 1% faster than the original one.
It the returns TRUE if the rate change has been effectively applied, FALSE otherwise; in fact there’s a stretching/compression threshold for the speed alteration which cannot be overtaken.
void DeclickOFF( )
It disables the execution of the processing function which applies some filtering to the stretched/compressed audio signal.
void DeclickON( )
Restore the filtering.
int GetCurrentBlock()
Returns the progressive number which identifies the current block (audio samples fragment) under execution. It is useful to put some marks into the audio file, hence for its synchronization with the external application (the main thread).

14.2 WaveMultiRatePlayer

14.2.1 Description

This class is the wave file player. The PLAY-PAUSE-STOP-JUMPTO functions are implemented. The main character of this player is the capability to change the playback rate during the track execution. The audio file is subdivided into 10000 samples blocks, each block is processed (stretched or compressed in its time duration) and played singularly.

15 BAT module

It is the fundamental module of the project and all the classes necessary for the management of the measure in all its parts and of the figures contained in the measure are part of it. It is connected with nearly all the other modules: on one hand it is connected with the SPA and LST modules for the management of the Battuta class (fundamental element of high level of the musical score), on the other hand it is connected with the modules ALT, FIG, TXT, DRW for the representation of the graphic elements of the measure and of the symbols that it contains.
In the musical theory it is Tempo that regulates the rational subdivision of the staffs in measures or Battute that comprehend each an equal sum of values shown by the alphanumeric values of Tempo; each measure is delimited through a bar line (of different kind) that crosses vertically the staff and indicates that the sum of values of the figures from the last bar line till that point has reached the value imposed by the signs of Tempo. This setting that considers the measure as the fundamental element of the music scanning but not as the container of an isolated group of figures, reflects the natural way of manually written music; it is not, however, the best approach for a software implementation. We decided it would be more convenient (both for the organisation and for the management and thus the quickness) and more in accord with a object-oriented programming regarding the measure as an independent object containing a list of figures.

15.1 Class Battuta

15.1.1 Description

This is an abstract class central in the management of all the musical symbols and it is the first management level of the musical score. In fact the measure is that part of score where the musical figures are inserted as well as their related symbols; thus it is graphically consistent (a big rectangle) but depends on what is contained in it and on the settings.
Each measure has a Intestazione, Heading (placed at the beginning and consisting in the symbols of Chiave, Tempo and Armatura) and the Barra, bar line (placed at the end). The presence or not of the heading or of some of its parts depends on the heading of the measure that comes before in the musical score; thus methods of copy and the passage of information from a measure to another are developed.
In the Battuta class the management of Polifonia has been foreseen (that is to say the contemporary development of more than one layer) through separated lists of figures (typically 2) managed in a coordinated and parallel way. However the management of the polyphony has been developed referring to the main layer of the musical piece and by subordinating the second layer to the behaviour of the former (see insertion and deletion).
The measures are numbered; the measure number is represented (over the staff) in the measure that begins the staff with the exclusion of number 1. Therefore the NumBattuta class is instanced only in correspondence of these measures. The graphic object NumBattuta can be neither selected, nor modified, nor deleted.

15.1.2 Father class

DrawObject

15.1.3 Connected Types

enum tipoBattuta

Four different types of measures have been necessarily introduced because of their different behaviour in specific situations, specially regarding the management of Intestazione and Barra.
INIZIO_BRANO Measure with a complete heading and uncertain bar line.
INIZIO_PENTAGRAMMA Measure with a complete heading or a heading without Tempo (if it is not modified by the previous one) and uncertain bar line.
GENERICA Measure with a heading completely dependent on the previous measure and a uncertain bar line.

enum tipoBarra

The bar line at the end of the measure can be of different kinds:
SINGOLA Generic bar line

DOPPIA
FINALE Bar line at the end of a musical piece
INIZIORIT Bar line that shows the beginning of a part of staff (Refrain) that can be recalled by referring to the conventional symbols.
FINERIT Bar line that shows the end of the refrain (Ritornello).

enum tipoArmaturaChiave
The Key Signature, that is a part of the heading of the measure, introduces the tonality of the musical piece and can have the following values:
DO_maggiore
SOL_maggiore
RE_maggiore
LA_maggiore
MI_maggiore
SI_maggiore
FAd_maggiore
DOd_maggiore
FA_maggiore
SIb_maggiore
MIb_maggiore
LAb_maggiore
REb_maggiore
SOLb_maggiore
DOb_maggiore

15.1.4 Protected attributes

ListaFigure listaFig1,listaFig2
Figures' lists of the two layers of the measure for the management of the polyphony.
Int NumProgress
Progressive number of the score measure.
Intestazione intest
Heading of the measure.
Barra *ptrBarra
Bar line of the measure.
NumBattuta *ptrNumBattuta
Pointer to the progressive number of the measure (intended as graphic object).
NumGrande *ptrNumGrande
Pointer to NumUguale or NumPausa.
Scansione *ptrScansione
Pointer to Scansione.
Lettera *ptrLettera
Pointer to Lettera.
TSalto *ptrTSalto
Pointer to TSalto.
Movimento *ptrMovimento
Pointer to Movimento.
VUnit VU2Figure
Distance from AbsPos of the measure to the x-axis of the figure at the extreme left.
VUnit distanzaFig
Distance between two successive figures in the two layers.
VUnit BATTUTA2Lft,BATTUTA2Rgt
Distances from AbsPos of the right and left points of the rectangle containing the measure.
tipoBattuta tipoBat
Type of measure.
int numeroFig1, numeroFig2
Number of figures in the two layers of the measure.
VUnit xBarra

Additional distance of the bar line from the last figure for eventual adjustments at the end of the staff.

VUnit VU2Up, VU2Dwn

Distances of the superior and inferior limit of the measure from the inferior line of the staff.

NumCode NumericCode

Numeric code of the measure (NumCode is defined as short).

NumCode NumPagDLIOO

Tell if a measure of the DLIOO is at the beginning of the page (value !=0) or not (null value). If different from zero return the page number.

NumCode NumPagMASAE

Tell if a measure of the MASAE or MASE is at the beginning of the page (value !=0) or not (null value). If different from zero return the page number.

NumCode NumPagMASE

Tell if a measure of the MASAE or MASE is at the beginning of the page (value !=0) or not (null value). If different from zero return the page number.

Bool FirstColonna

Tell if a measure is at the beginning of the column of measures.

Barra *ptrBarraPrec

Barline of the previous measure.

tipoBarra tpBarra

Type of barline of the current measure.

int NumMultiRest

Return the number of measures “contained” in the multirest.

Bool IsMultiRest

If TRUE the measure belongs to a multirest.

Battuta *pMultiRest

Pointer to the first pause measure of a multirest. Pointer to the multirest measure.

Public attributes

TipoGiust tipogiust

Type of justification (linear = G_LIN, logarithmic = G_LIN, measure never justified = G_NONE)

double kgiustificazione

Tuning parameter of the justification task

UL Dir1Sp2FirstFig

Spaces to the first figure of the measure in the directorial view for the layer 1 (1 UL = 100 VU)

UL Dir2Sp2FirstFig

Spaces to the first figure of the measure in the directorial view for the layer 2

UL Orc1Sp2FirstFig

Spaces to the first figure of the measure in the musician view for the layer 1

UL Orc2Sp2FirstFig

Spaces to the first figure of the measure in the musician view for the layer 2

UL Dir1Sp2FirstFigLine

Spaces to the first figure of the measure in the directorial view used in auto line breaking (not saved) for the layer 1

UL Dir2Sp2FirstFigLine

Spaces to the first figure of the measure in the directorial view used in auto line breaking (not saved) for the layer 2

UL Orc1Sp2FirstFigLine

Spaces to the first figure of the measure in the musician view used in auto line breaking (not saved) for the layer 1

UL Orc2Sp2FirstFigLine

Spaces to the first figure of the measure in the musician view used in auto line breaking (not saved) for the layer 2

UL Dir1Sp2FirstFigUt

Spaces to the first figure of the measure in the directorial view manually introduced by the user in the layer 1

UL Dir2Sp2FirstFigUt

Spaces to the first figure of the measure in the directorial view manually introduced by the user in the layer 2

UL Orc1Sp2FirstFigUt

Spaces to the first figure of the measure in the musician view manually introduced by the user in the layer 1

UL Orc2Sp2FirstFigUt

Spaces to the first figure of the measure in the directorial view manually introduced by the user in the layer 2

Private methods

NumCode Note2GruppoNote(short layer, TipoInserimento tpIns, NumCode figCode1, NumCode figCode2)

It creates a beam on the indicated layer starting from the figure with code figCode1 until that with code figCode2. It foresees also the insertion of spaces in the beam. Please pay attention: the method does not perform a preliminary control on the feasibility of the grouping, but groups until it can do it. If even the first figure cannot be inserted in the beam, the method does nothing and returns FALSE.

Bool GruppoNote2Note(GruppoNote*)

It convert the beamed group in single notes and chords inserting them again in the list

NumCode InsNota(int Altezza, Bool SegnoAltezza, ClassID figID, short layer, TipoInserimento GamboUp, Bool inAccordo, NumCode figCode, Bool InsInTesta)

The method inserts a note in the measure according to the following parameters:
Altezza: the height of the note.
figID: type of note.
layer (1 or 2): indicator of the list of figures in which the note is to be inserted.
GamboUp: it is to be put TRUE in order to insert notes with a upwards directed stem.
InAccordo: it is to be put FALSE to insert among already existing figures, TRUE to add the note to already existing chords.
figCode. If inAccordo is FALSE, the method inserts the new note after the figure of the layer with code figCode. If inAccordo is TRUE, the method searches for the figure with code figCode. If this is a note or a chord, the new note is added; otherwise, if it is a space, the new note replaces it; finally if it is a figure of another kind an error message is sent and the return value is FALSE.

InsInTesta: if it is TRUE the note is placed at the top of the list, otherwise it has no effect.

When inAccordo is FALSE and the note is inserted in a layer, a space is inserted in the corresponding position of the other layer; the spaces of layer 1 are placed on the highest line of the staff, whereas those of layer 2 are placed in the lowest one.

Bool InsInAccordo(Nota *pNota, Figura *pFig, ListaFigure *pLayer, TipoInserimento GamboUp, short layer)

If pFig points to a chord (or a single note) and the note pointed by pNota is compatible with the chord, the note is inserted. If pFig points to a space, the note will substitute the space. It is **indispensable** to specify the pointer to the layer in which pFig is to be found.

NumCode InsPausa(int Altezza, Bool SegnoAltezza, ClassID figID, short layer, TipoInserimento AltVariabile, Bool inAccordo, NumCode figCode, Bool InsInTesta)

The method inserts a rest in the measure according to the following parameters:
Altezza: the height of the rest.
figID: type of rest.
layer (1 or 2): indicator of the list of figures in which the rest is to be inserted.
AltVariabile: it is to be put TRUE in order to insert a rest at the set height. If it is FALSE, each rest (both of layer 1 and 2) is placed at standard height.
InAccordo: it is to be put FALSE to insert among already existing figures, TRUE to make the rest substituting existing spaces.
figCode. If inAccordo is FALSE, the method inserts the new rest after the figure of the layer with code figCode. If inAccordo is TRUE, the method searches for the figure with code figCode. If this is a space the new rest replaces it; if it is a figure of another kind an error message is sent and the return value is FALSE.
InsInTesta: if it is TRUE the rest is placed at the top of the list, otherwise it has no effect.

When inAccordo is FALSE and the rest is inserted in a layer, a space is inserted in the corresponding position of the other layer; the spaces of layer 1 are placed on the highest line of the staff, whereas those of layer 2 are placed in the lowest one.

NumCode InsSpazio(short layer, NumCode figCode, Bool InsInTesta, short CodeCommand)

The method adds a space in the indicated layer, on the right of the figure with code figCode, or at the beginning of the list if InsInTesta is TRUE. A space is inserted in the corresponding position in the other layer according to the rule: layer 1 => space on the fifth line of the staff (height 8), layer 2 => space on the first line (height 0).

NumCode InsRipetizione(ClassID figID, NumCode figCode, Bool InsInTesta, short layer)

The method adds a sign of repeat on the right of the figure with code figCode, or at the beginning of the list if InsInTesta is TRUE. The type of repeat is indicated by figID; the symbol is inserted in the first layer. A space is inserted in the corresponding position in the other layer according to the rule: layer 1 => space on the fifth line of the staff (height 8), layer 2 => space on the first line (height 0).

Bool InsNumPausa(unsigned char* s)

The method adds a text in which the number associated to the generic rest that constitutes the measure is represented. It returns FALSE if it fails.

Bool InsNumUguale(unsigned char* s)

The method adds a text in which the number used for the scanning of measures in a set of alike measures is represented. It returns FALSE if it fails.

Bool InsScansione(short n)

The method adds a scanning symbol, consisting of a set of n vertical bar lines one next the other. It returns FALSE if it fails.

Bool InsTSalto(unsigned char* s)

The method adds a symbol of TSalto, a text that shows from which point the execution of the musical piece will start again. It returns FALSE if it fails.

Bool InsLettera(unsigned char* s)

The method adds a symbol of Lettera, a text of one character that represents a sign of re-start of the piece. It returns FALSE if it fails.

Bool InsMovimento(unsigned char *p, ClassID ident, Bool punt, int s)

The method adds a Movimento symbol that shows the cadence and the execution times of a musical piece. The Movimento is represented by a text (TMovimeto) and a symbol of Metronomo. The former shows the cadence to be imposed to the music and is set through p. The latter regulates the execution times and is expressed by a note, an eventual dot and a number (ident, punt and n). It returns FALSE if it fails.

Void AdjustTremolo (Figura *fig, short layer);
prepare to draw symbol of tremolo

Void AdjustGlissato (Figura *fig, short layer);
prepare to draw symbol of glissato

15.1.5 Public methods

Battuta ()
Constructor that initialises the empty measure and *VU2Figure* at 0.
~ **Battuta ()**
Destroyer that recalls the Free method.
void Free ()
It deallocates all the pointers connected to the measure. It recalls the homonymous method of the figure lists of measure.

void SetNumberOfStaff(short nStaffs);
short GetNumberOfStaff();
The number of staffs used by measure (the same as the Spartito object).

Intestazione* GetIntest(short staff=0);
Each measure has up to 3 headers (*Intestazione*) one for each staff, with 3 staffs staff 0 is the upper, staff 1 is the middle and staff 2 is the lower (for 3 staffs). With 2 staffs staff 0 is the upper and staff 1 is the lower. This method gets the header 0 (the default) 1 or 2
Intestazione GetAllIntest();**
Gets all the headers as an array of pointers to Intestazione object.

Intestazione* GetIntestEnd(short staff=0);
Each measure now has also headers at the end of the measure (are not displayed) considering also eventual modification of clef or keysignature. This method gets the header at the end of the measure (for a certain staff)
Intestazione GetAllIntestEnd();**
The same as GetIntestEnd but for all the staffs.

void SetChiave(short staff, Chiave *chv);
Sets the clef for a certain staff (0, 1 or 2).
void SetArmatura(short staff, tipoArmaturaChiave arm);
Sets the keysignature for a certain staff (0, 1 or 2).
Chiave *GetChiave(short staff);
Sets the clef for a certain staff (0, 1 or 2).
tipoArmaturaChiave GetArmatura(short staff);
Gets the keysignature for a certain staff (0, 1 or 2).
v**oid SetTempo(char tmp[])**
It sets the type of time for all the staffs.
Tempo GetTempo(void)
Return tempo of the measure.
Layer *GetPLayer(short layer);
Gets a specific layer (from 0 to 11), it returns NULL if the layer is never used.
The following restrictions are present:

- with 1 staff no more than 4 layers can be used (layers 0 – 3)
- with 2 staffs no more than 8 layers can be used (layers 0 – 7)
- with 3 staff all the 12 layers can be used.

For compatibility layers 0 and 1 (ex layers 1 and 2) are always present and can be empty, for the others if a non NULL pointer is returned by GetPLayer at least one figure is present.

void SetPNumBattuta(NumBattuta *p)
It sets *ptrNumBattuta* at *p*.
void SetPScansione(Scansione *p)
It sets *ptrScansione* at *p*.
void SetPLettera(Lettera *p)
It sets *ptrLettera* at *p*.
void SetPTSalto(TSalto *p)
It sets *ptrTSalto* at *p*.
void SetNumProgress(int num)
It sets *NumProgress* at *n*.
int GetNumProgress(void)
It returns *NumProgress*.
void SetNumPagDLIOO(int num)
It sets *NumPaginaDLIOO* at *num*.
NumCode GetNumPagDLIOO (void)
It returns *NumPaginaDLIOO*.
void SetNumPagMASAE(int num)
It sets *NumPagina***MASAE** at *num*.
NumCode GetNumPagMASAE (void)
It returns *NumPagina***MASAE**.
void SetNumPagMASE(int num)
It sets *NumPagina***MASE** at *num*.
NumCode GetNumPagMASE (void)
It returns *NumPagina***MASE**.

void SetCodeCounterFig1(NumCode nc)
It sets at nc the numeric code of the list listaFig1.
NumCode GetCodeCounterFig1 ()
It returns the numeric code of the list listaFig1.
void SetCodeCounterFig2(NumCode nc)
It sets at nc the numeric code of the list listaFig2.
NumCode GetCodeCounterFig2 ()
It returns the numeric code of the list listaFig2.
VUnit GetVU2Up()
It returns the distance from AbsPos to the superior point.
VUnit GetVU2Dwn()
It returns the distance from AbsPos to the inferior point.
VUnit GetVU2Lft()
It returns the distance from AbsPos to the left point.
VUnit GetVU2Rgt()
It returns the distance from AbsPos to the right point.
void SetNumericCode(NumCode nc)
It sets at nc the numeric code.
NumCode GetNumericCode ()
It returns the numeric code of the measure.
v**oid Setup(VUnit vf, VUnit MaxWdt=0)**
It calculates the number of figures that the measure contains. It sets VU2Figure at *vf* and performs SetVU().
void SetVU (VUnit MaxWdt=0)
It calculates the dimensions of the measure. For disposing the first figure, it considers *VU2Figure* or, if it is 0, considers the width of the heading.
int GetNumeroFig()
It returns the maximal number of figures contained in the measure (that corresponds to the number of figures of the layer that contains more of them)
tipoBattuta GetBattuta()

It returns the type of measure.

void CopyAllIntest(Intestazione intes)**

It copies all the heading endowed on the measure heading.

Figura *GetFig(int lay, NumCode nc)

It returns the pointer to the figure in the lay layer with nc NumericCode.

void Draw()

It draws the measure.

void Select()

It selects the object by setting the colour of selection and re-designing the object.

void Deselect()

It deselects the object by setting the normal colour of selection and re-designing the object.

Bool Hit(const Point&, SelObj,DrawObject*&, SymPath&)

See the general documentation in section 1.18.2.

It is the method to be performed in order to have the position in the figures' list where notes, rests or spaces are to be inserted. The method can be recalled if the measure has positively answered to BATTUTA_SEL or, at least, to BATPART_SEL selection.

If the point posiz is inside the measure, the exit parameter questaBat assumes TRUE value, otherwise FALSE. In TRUE case, the exit parameter Altezza represents the height of the note corresponding to posiz.y, while in the FALSE case it is meaningless. According to the value set for inAccordo (FALSE if we want insert among existing figures, TRUE if we want insert in a chord or at the place of a space), the method scans the indicated layer (1 or 2) and sets FigCode:

- If we insert among existing figures, FigCode represents the numeric code of the figure of the measure **after** that in which we want to perform the insertion. If the point posiz is on the left with respect to all the figures of the layer, InsInTesta assumes TRUE value as insertion indication on the top of the layer.
- If we insert in a chord, FigCode represents the numeric code of the figure to which the note is to be inserted.

At the method exit, FigCode never assumes a null value (non-selection indicator).

Bool DeleteHit(const Point& posiz, NumCode& FigCode, NumCode& InsInTesta)

FigCode is the code of the figure of layer 1 which has been "struck" from the point *posiz*. Since the method is useful to delete also columns of figures, the figure is considered "struck" only if it is struck in the x-axis.

Bool Hit(const Rectangle& r, SelObj selObj, short layer, SymPath&, SymPath&)

See the general documentation in section 1.18.2.

Return a list of objects type SelObj inside the selection rectangle. Return NULL if the rectangle does not contain anythingor the objects inside the rectangle are not of SelObj type.

Bool SymbolicHit(SymPath&,SelObj,DrawObject*&)

See the general documentation in section 1.18.2.

Used to decode a path.

Bool FigPtr2SymPath(SymPath&,Figura*)

It reconstructs the path of a figure starting from the pointer.

Barra* GetBarra()

It returns the pointer to the bar line of the measure.

Void EliminaGruppi()

It looses all the beams of the measure by converting them in single notes.

CmdResult SymCommand(SymCmd&, Battuta *pb=NULL)

It performs the indicated command. It returns CMD_OK if the command has been successful, CMD_FAIL if the command has not been executed, CMD_UNKNOWN if the command doesn't regard it. The commands implemented in this class are all direct and concern insertions of notes, rests and spaces, beams and the bar line.

Bool DeleteSym(Drawobject*)

It deletes the specified symbol (it returns TRUE if it has been deleted). If it deals with a figure, it replaces it with the corresponding spaces.

void EraseBattuta()

It converts in spaces all the figures of the measure.

void AddFig(Figura* fig, int lay)

It adds in sequence the figures to the list.

void Suona(MoodsMidiWrite *)

Generates midi file.

DrawObject* GetObject (SymCmd *MyCmd)

Get the pointer to an hit object by a symbolic command. Works for commands regarding the measure as armature, chiavi, etc.

void SetNumMultiRest(int nb)

Set the number of measures of pauses in a multirest measure.

void GetNumMultiRest ()

Get the number of measures of pauses in a multirest measure

void SetPMultiRest (Battuta *pb)

Set the pointer to the multirest measure

Battuta* GetPMultiRest ()

Get the pointer to the multirest measure

void SetIsMultiRest (Bool ismrest=TRUE)

Set the flag indicating that the measure is a multirest measure

Bool GetIsMultiRest()

Set the pointer to the multirest measure

void SetPNumGrande(NumGrande *p)

Regarding the generic pause, set the pointer to NumGrande

NumGrande *GetPNumGrande()

Get the pointer to the NumGrande

Lettera *GetPLettera()

Regarding the Lettera (label), return the pointer to Lettera

void SetPMovimento(Movimento *)

Regarding the composite object movimento (allegro etc.), set the pointer to Movimento

Movimento *GetPMovimento()

Regarding the object Movimento, get the pointer to Movimento

float GetMetronomoTimeExec()

Regarding the object Movimento, return 0 if pointer to Movimento is NULL, otherwise the metronomic time of execution as a float number

float GetTempoTimeExec()

Regarding the object Movimento, return key time of execution as a float number: numerator/denominator

void DrawBall()

Drawing methods used in execution mode, actually deprecated

void SetVU2Lft(Vunit vu)

Set Visual Unit used to setup parameters for the drawing of the measure, this sets BATTUTA2Lft

void SetVU2Rgt(Vunit vu)

Set Visual Unit used to setup parameters for the drawing of the measure, this sets BATTUTA2Rgt

void SetVU2Up(Vunit vu)

Set Visual Unit used to setup parameters for the drawing of the measure, this sets VU2Up

void SetVU2Dwn(Vunit vu)

Set Visual Unit used to setup parameters for the drawing of the measure, this sets VU2Dwn

void SetxBarra(Vunit xpb=0)

Set the additional distance for xPosBarra

void SetupNumeroFig()

Set parameters numeroFigfor all the layers

void SetBattuta(tipoBattuta tpbat=GENERICA, Intestazione *intst=NULL,tipoSpartito tsp=SPART_GENERICO)

Set the type of measure (GENERICA is the default type)

void SetPos(DrawObject*, const Point&)

Set the position of the measure and of all its elements

Bool InsertHit(const Point& posiz, short layer, Bool inAccordo, Bool& questaBat, NumCode& Altezza, Bool& SegnoAltezza, NumCode& FigCode, NumCode& InsInTesta)

To be used for the measures selected with BATPART_SEL.

void SetBarra(tipoBarra = SINGOLA)

Set parameter tpBarra and allocates new Barra

void SetptrBarra(tipoBarra tbar)

Depending on the type (SINGOLA, DOPPIA, FINALE, INIZIORIT, FINERIT, INIZIOFINERIT) allocates new Barra

void SetptrBarraPrec(tipoBarra tbar, Bool IsNull)

Depending on the type (SINGOLA, DOPPIA, FINALE, INIZIORIT, FINERIT, INIZIOFINERIT) allocates new Barra for preceding measure Barra.

int Save(FILE*, Context)

Used for the saving

char *Describe()

Used for the transmission on the network for cooperative editing

void SetFirsColonna(Bool)

Set parameter FirstColonna

Bool GetMultivoice()

Return TRUE if more than one layer is present in the measure

int GetVoice(Figura* pFig)

Return the number of layer where the figure is, actually 1 or 2.

void SetupSimboli()

Run adjust on the lists of figures for layer 1 and 2

void SetupGambi()

Setup the stems, up and down for the notes of the measures according to the MILLA formatter

NumCode GetPos(int lay, Figura *ptr)
Return the NumCode for the figure in the layer specified

Figura* GetFigTotal(int lay, NumCode nc)
Return pointer to figure having NumCode=nc

Figura* GetFigAll(int lay, NumCode nc)
Return pointer to figure having NumCode=nc, search also in chords (Accordo)

Bool GetChiaveFig(Figura *pfig)
Return TRUE if all figures in the list have an ID (id >= CL_CVIOLINO) && (id <= CL_CSOPRANO), otherwise return FALSE

void CheckBattuta()
Deprecated

Bool CheckBatt()
Check time consistency of the measure, if it right return TRUE, otherwise FALSE

v**oid SetupDurata()**
Setup the duration for the figures of the measure if the figure is a semibreve pause (set duration = key tempo of the measure) or a non regular duration figure (terzina).

v**oid BeamingAuto()**
When the measure in editing mode becomes time consistent, this method try to beam figures in groups depending on the time key of the measure.

int CheckBattPause(Battuta *pbat)

v**oid MakeBattMultiRest(Battuta* pbat1, Battuta *pbat2, int nb)**
Scans the measure and verify that each figure that belongs to it is a pause and that the measure is time consistent.
Return 1 if the measure is composed of pauses only,
return 0 if in the measure at least one note is present,
return 2 if the measure is composed of pauses only, but with a time key different from the first or with a different barline.
v**oid print()**
Print method of object measure.
v**oid GiustificaBattuta(TipoGiust, double kGiust)**
Tell the system to justify the measure with the passed parameters
void SetupIntRefs();
It sets in the figures of the score the references to the horiz. symbols starting/ending/over each figure.

15.2 Class ListaFigure

15.2.1 Description
This class is necessary in order to allow other methods that need to return a Figura to use in a direct way methods already present for the management of lists (in Lista class).
Each figure that is added to the list has been endowed of a unequivocal numeric code. To this end for each insertion the counter *CodeCounter* is updated, and it is not decreased when a deletion occurs. The type **NumCode** is defined as short.

15.2.2 Father Class
Lista

15.2.3 Protected Attributes

NumCode CodeCounter

Counter to assign the numeric code to each object of the list.

Node* lastNode

Last node visited with methods GetFirstFig or GetNextFig.

GruppoNote* lastGruppoNote

Last group visited with methods GetFirstFig or GetNextFig.

15.2.4 Public methods

ListaFigure()

Constructor that puts CodeCounter=1.

~ ListaFigure()

Destroyer: it performs *Free*.

void Free()

Deallocates the list.

void SetCodeCounter(NumCode nc)

It sets the value of the CodeCounter at nc.

NumCode GetCodeCounter()

It returns the value of the CodeCounter.

void InsertEnd(DrawObject *newobj)

It inserts an element in the bottom of the list, giving the code to it.

void InsertEndSimple(DrawObject *newobj)

It inserts a figure without changing the numeric code.

void InsertTop(DrawObject *newobj)

It inserts an element on the top of the list, giving the code to it.

Figura *GetFirstF(void)

It returns the first figure of the list.

Figura *GetLastF(void)

It returns the last figure of the list.

Figura *GetNextF(Figura *pf)

It returns the figure successive to the figure pointed by pf.

Figura *GetPrevF(Figura *pf)

It returns the figure that precedes the figure pointed by pf.

void InsertAfterF(Figura *pf1,Figura *pf2)

Inserts the figura of pf2 after the figure of pf1 and gives the code to the inserted figure.

void InsertAfterFSimple(Figura *pf1,Figura *pf2)

Inserts the figura as above, but does not reenumerate numeric codes.

Figura *DelFirstF(void)

It deletes the first figure of the list.

Figura *DelLastF(void)

It deletes the last figure of the list.

Figura *AtPosF(unsigned long i)

It returns the i figure of the list (i>=0 && i < numobject).

Figura *ChangeF(Figura *p1, Figura *p2)

It changes the figure p1 with the figure p2, by setting the numeric code of p2.

Figura *ChangeFSimple(Figura *p1, Figura *p2)

It changes the figure p1 with the figure p2.

Figura *GetFig(NumCode nc)

It returns the figure with NumericCode nc.

Figura *GetFigTotal(NumCode nc)

It returns the figure with NumericCode nc, checking also the groupes of notes.

Figura *GetFigAll(NumCode nc)

It returns the figure with NumericCode nc, checking also the chords.

Figura *GetNextFigura(Figura *pf)

It returns the next element to pf, seeking also in the groupes of notes and in chords.

Bool FigPtr2SymPath(SymPath&, Figura*)

It reconstructs the path of a figure starting from the pointer.

Bool FindFig(Figura *pf)

It returns TRUE if pf is in the list of figures, seeking also in the groupes of notes and in chords.

void Suona(MoodsMidiWrite *)

This method is implemented in the child classes Nota, Figura, Pausa etc. It generates the Midi file to be played.

int Save(FILE*, Context)

Save method.

char *Describe(Context context)

Return description of the class.

Figura *GetFirstFig(void)

It returns the first figure of the list, entering also in the groupes of notes.

Figura *GetNextFig(void)

It returns the successive element of the list.

GruppoNote *GetLastGruppoNote(void)

It returns a pointer to the group of notes that eventually contains the last figure extracted with GetFirstFig/GetNextFig. Returns NULL is such group does not exist.

void InsertAfterFig(Figura *pf1, Figura *pf2, Battuta *pbat, short layer)

It inserts figure pf2 after figure pf1 in the measure if it does not exist a group of notes.

void DelPtrFig(Figura *pf1, Battuta *pbat, short layer)

It delete the figure in the measure or in the last group of notes.

int GetPosFig(DrawObject* pf)

Finds the position of the figures in the list of figures and in the group of notes.

Figura *AtPosFig(int pos)

Finds the figure in position pos in the list of figures and in the groupes of notes.

float GetDurataTerz(Figura *pfig1, LegatQuadra *pleg, float den)

Returns the durata of irregular duration figure.

UL GetSpTotale()

It returns the sum of the spacing attributes of the list of figures and groupes of notes, depending on the view (directorial, musician).

UL GetSpTotaleLine()

It returns the sum of the spacing attributes of the line breaking of the list of figures and groupes of notes, depending on the view (directorial, musician).

void GetMinMaxStaff(short& minStaff, short& maxStaff);

Gets the minimum/maximum staff of the figures in the list.

15.3 Class Intestazione

15.3.1 Description

In this class, that is a part of measure (relation IPO), the heading of the measure is implemented and managed in its three parts: the musical clef (relation IPO with the Chiave class), key signature (relation IPO with the ArmaturaChiave class), the time (relation IPO with the Tempo class).

15.3.2 Father Class

DrawObject

15.3.3 Connected types

Enum tipoArmaturaChiave

The key signature, that is part of the measure heading, determines the tonality to the musical piece and can assume the following values:

DO_maggiore
SOL_maggiore

RE_maggiore
LA_maggiore
MI_maggiore
SI_maggiore
FAd_maggiore
DOd_maggiore
FA_maggiore
SIb_maggiore
MIb_maggiore
LAb_maggiore
REb_maggiore
SOLb_maggiore
DOb_maggiore
LA_minore
MI_minore
SI_minore
FAd_minore
DOd_minore
SOLd_minore
REd_minore
LAd_minore
RE_minore
SOL_minore
DO_minore
FA_minore
SIb_minore
MIb_minore
LAb_minore

15.3.4 Protected

ArmaturaChiave armChiave, armChiaveAttuale
Key signature of the heading of the present measure.

Tempo tempo
Time of the heading of the present measure.
Chiave *ptrChiave
Pointer to the clef of the heading of the present measure.
Bool drwChiave
Boolean that indicates if designing (TRUE) or not the clef of the heading.
Bool drwArmChv
Boolean that indicates if designing (TRUE) or not the key signature of the heading.
Bool drwTempo
Boolean that indicates if designing (TRUE) or not the time of the heading.
VUnit INTESTAZIONE2Up,INTESTAZIONE2Dwn, INTESTAZIONE2Lft,INTESTAZIONE2Rgt
Distances from AbsPos of the superior, inferior, left and right points of the rectangle containing the heading.

15.3.5 Public Methods

Intestazione()
It initialises the pointer to the clef at NULL.
~ Intestazione ()
Destroyer that deallocates the clef.
VUnit GetVU2Up()
It returns the distance from AbsPos to the superior point.
VUnit GetVU2Dwn()
It returns the distance from AbsPos to the inferior point.
VUnit GetVU2Lft()
It returns the distance from AbsPos to the left point.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the right point.

VUnit SetVU2Up(Vunit vu)

Set the distance from AbsPos to the superior point.

VUnit SetVU2Dwn(Vunit vu)

Set the distance from AbsPos to the inferior point.

VUnit SetVU2Lft(Vunit vu)

Set returns the distance from AbsPos to the left point.

VUnit SetVU2Rgt(Vunit vu)

Set returns the distance from AbsPos to the right point.

void SetChiave(Chiave *chv)

It sets the clef of the heading.

Chiave *GetChiave()

It returns the clef of the heading.

void SetArmatura(tipoArmaturaChiave arm)

It sets the key signature of the heading.

tipoArmaturaChiave GetTipoArmatura()

It returns the type of key signature of the heading.

ArmaturaChiave GetArmatura()

It returns the key signature of the heading.

ArmaturaChiave *GetPtrArmatura()

It returns the pointer to the key signature of the heading.

void SetTempo(char tmp[])

It sets the Tempo of clef of the heading.

Tempo GetTempo()

It returns the Tempo of the heading.

Tempo *GetPtrTempo()

It returns the pointer to the Tempo of the heading.

void Copy(Intestazione*)

It copies the present heading.

void Draw()

It designs the heading in its active parts.

Bool Hit(const Point&, SelObj, DrawObject*&, SymPath&)

See the general documentation of the method in section 1.18.2.

CmdResult SymCommand(SymCmd&, Battuta *pb=NULL)

It performs the indicated command. It returns CMD_OK if the command has been successful, CMD_FAIL if it has not been performed, CMD_UNKNOWN if the command does not concern it. The commands of this class are all direct and concern the insertion of the musical clef in the measure.

void Free()

Deallocates all the pointers connected to the heading.

void SetVU(tipoBattuta, Intestazione*)

Set the dimensions of Intestazione depending on the type of actual measure and the Intestazione of the preceding measure.

void SetPos(DrawObject *d, const Point& p)

Set the position of the present parts of Intestazione depending on the type of actual measure.

int Save(FILE *fp, Context)

Save method.

char *Describe (Context context)

Return description of the class.

void print()

Print method.

15.4 Class ArmaturaChiave

15.4.1 Description

In this class the part of the measure heading called key signature is developed and managed; from the musical point of view this class determines, through the musical accidentals that compose it, the *tonality* of the musical piece. The accidentals of the key signature are valid for the whole measure and, if not differently specified, for the following measures as well. It is important to highlight the peculiarities in the link among signatures of two consecutive measures: if in a measure there is no tonality variation with respect to the previous measure, the key signature is not designed again; whereas if, for example, there is a passage from a sharp key signature to a flat one it is necessary to design, besides the modified key signature, a signature of naturals that cancels the previous tonality.

15.4.2 Father Class

DrawObject

15.4.3 Connected types

Enum tipoArmaturaChiave

The key signature, that is part of the measure heading, gives the tonality to the musical piece and can assume the following values:

DO_maggiore
SOL_maggiore
RE_maggiore
LA_maggiore
MI_maggiore
SI_maggiore
FAd_maggiore
DOd_maggiore
FA_maggiore
SIb_maggiore
MIb_maggiore
LAb_maggiore
REb_maggiore
SOLb_maggiore
DOb_maggiore
LA_minore
MI_minore
SI_minore
FAd_minore
DOd_minore
SOLd_minore
REd_minore
LAd_minore
RE_minore
SOL_minore
DO_minore
FA_minore
SIb_minore
MIb_minore
LAb_minore

15.4.4 Protected Attributes

VUnit ARMATURA2Up,ARMATURA2Dwn,ARMATURA2Lft,ARMATURA2Rgt

Distances from AbsPos of the superior, inferior, left and right points of the rectangle containing the key signature.

Diesis dss[7]

Sharp of the key signature.

Bemolle bmll[7]

Flat of the key signature.

Bequadro bqdr[7]

Natural of the key signature.

int numeroDiesis

Number of sharp of the key signature.

int numeroBemolle

Number of flat of the key signature.

int numeroBequadroD

Number of natural of the key signature.

int numeroBequadroB

Number of natural of the key signature.

TipoArmaturaChiave tipoArm

Type of key signature.

15.4.5 Public Attributes

int shiftChiave

Distance of the key signature from the clef.

int altezzaChiave

Height of the clef.

Bool tipoAlt

Kind of key signature (sharp or flat) to which, if existing, the key signature of naturals must refer.

tipoArmaturaChiave tipoArmPrec

Kind of key signature of the previous measure.

15.4.6 Public Methods

ArmaturaChiave()

Initialiser.

VUnit GetVU2Up()

It returns the distance from AbsPos to the superior point.

VUnit GetVU2Dwn()

It returns the distance from AbsPos to the inferior point.

VUnit GetVU2Lft()

It returns the distance from AbsPos to the left point.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the right point.

VUnit SetVU2Up(Vunit vu)

Set the distance from AbsPos to the superior point.

VUnit SetVU2Dwn(Vunit vu)

Set the distance from AbsPos to the inferior point.

VUnit SetVU2Lft(Vunit vu)

Set the distance from AbsPos to the left point.

VUnit SetVU2Rgt(Vunit vu)

Set the distance from AbsPos to the right point.

void SetArmatura(tipoArmaturaChiave tparm)

It sets the type of key signature of the heading.

TipoArmaturaChiave GetArmatura()

It returns the type of key signature of the heading.

void Draw()

It draws the key signature.

void SetVU(int)

It set the dimensions of the key signature.

void SetPos(DrawObject *drwobj, const Point& p)

It sets the position of the key signature.

char *Describe(Context)

Return description of the class.

Bool HasSameDesc(ArmaturaChiave*)

Returns TRUE if the object has the same description of that passed as parameter.

UL GetSp2NextFig()

Returns space in logic unit for the current view until next figure.

15.5 Class Chiave

15.5.1 Description

It is an abstract class that is used to represent the musical clefs.

15.5.2 Father Class

DrawObject

15.5.3 Protected Methods

Bool Chiavina

It returns TRUE if it's a chiavina, FALSE if it's a chiave.

15.5.4 Public Methods

Chiave()

Constructor

Bool Getchiavina()

Return parameter chiavina

void Setchiavina(Bool)

Set parameter chiavina

VUnit GetVU2Up()

It returns the distance from AbsPos to the superior point.

VUnit GetVU2Dwn()

It returns the distance from AbsPos to the inferior point.

VUnit GetVU2Lft()

It returns the distance from AbsPos to the left point.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the right point.

virtual int GetShift()

It returns for each clef the shift of the key signature with respect to the tremble clef.

virtual int GetAltezza()

It returns for each clef the height of the key signature with respect to the tremble clef.

void Draw()

It is necessary for the VIRTUAL definition of draw() in DrawObject.

UL GetSp2NextFig()

Returns spaces in UL in the current view until the next figure of the voice.

v**irtual char *Describe(Context)**

Returns the description of the object in MusicTex.

UL GetSp2NextFig()

Returns spaces in UL in the current view until the next figure of the voice.

Bool HasSameDesc (Chiave*)

Returns TRUE if object has the same description of the one passed as parameter.

virtual void Select ()

Call corresponding function of DrawObject , draw.

virtual void Deselect ()

Call corresponding function of DrawObject , draw with old color.

virtual void print ()

Print method.

15.6 Class CBaritono

15.6.1 Description

This is a symbol class that represents the symbol of the Baritone clef.

15.6.2 Father Class

Chiave

15.6.3 Public Methods

CBaritono()
Initialises the object.
VUnit GetVU2Up()
It returns the distance from AbsPos to the superior point.
VUnit GetVU2Dwn()
It returns the distance from AbsPos to the inferior point.
VUnit GetVU2Lft()
It returns the distance from AbsPos to the left point.
VUnit GetVU2Rgt()
It returns the distance from AbsPos to the right point.
int GetShift()
It returns for each clef the shift of the key signature with respect to the tremble clef.
int GetAltezza()
It returns for each clef the height of the key signature with respect to the tremble clef.

void SetPos(DrawObject *d, const Point& p)
Set the position of the key.
void Draw()
Draw method.
char *Describe(Context)
Return description of the class.
void print()
Printing method.

15.7 Class CBasso

15.7.1 Description

This is a symbol class that represents the symbol of the Bass clef.

15.7.2 Father Class

Chiave

15.7.3 Public Methods

CBasso()
Initialises the object.
VUnit GetVU2Up()
It returns the distance from AbsPos to the superior point.
VUnit GetVU2Dwn()
It returns the distance from AbsPos to the inferior point.
VUnit GetVU2Lft()
It returns the distance from AbsPos to the left point.
VUnit GetVU2Rgt()
It returns the distance from AbsPos to the right point.
int GetShift()
It returns for each clef the shift of the key signature with respect to the tremble clef.
int GetAltezza()
It returns for each clef the height of the key signature with respect to the tremble clef.
void SetPos(DrawObject *d, const Point& p)
Set the position of the key.
void Draw()
Draw method.
char *Describe(Context)
Return description of the class.
void print()

Printing method.

15.8 Class CContralto

15.8.1 Description

This is a symbol class that represents the symbol of the Alto clef.

15.8.2 Father Class

Chiave

15.8.3 Public Methods

CContralto()
Initialises the object.

VUnit GetVU2Up()
It returns the distance from AbsPos to the superior point.

VUnit GetVU2Dwn()
It returns the distance from AbsPos to the inferior point.

VUnit GetVU2Lft()
It returns the distance from AbsPos to the left point.

VUnit GetVU2Rgt()
It returns the distance from AbsPos to the right point.

int GetShift()
It returns for each clef the shift of the key signature with respect to the tremble clef.

int GetAltezza()
It returns for each clef the height of the key signature with respect to the tremble clef.

void SetPos(DrawObject *d, const Point& p)
Set the position of the key.

void Draw()
Draw method.

char *Describe(Context)
Return description of the class.

void print()
Printing method.

15.9 Class CMezzosoprano

15.9.1 Description

This is a symbol class that represents the symbol of the Mezzosoprano clef.

15.9.2 Father Class

Chiave

15.9.3 Public Methods

CMezzosoprano()
Initialises the object.

VUnit GetVU2Up()
It returns the distance from AbsPos to the superior point.

VUnit GetVU2Dwn()
It returns the distance from AbsPos to the inferior point.

VUnit GetVU2Lft()
It returns the distance from AbsPos to the left point.

VUnit GetVU2Rgt()
It returns the distance from AbsPos to the right point.

int GetShift()
It returns for each clef the shift of the key signature with respect to the tremble clef.

int GetAltezza()
It returns for each clef the height of the key signature with respect to the tremble clef.

void SetPos(DrawObject *d, const Point& p)
Set the position of the key.

void Draw()

Draw method.

char *Describe(Context)

Return description of the class.

void print()

Printing method.

15.10 Class Soprano

15.10.1 Description

This is a symbol class that represents the symbol of the Soprano clef.

15.10.2 Father Class

Chiave

15.10.3 Public Methods

CSoprano()

Initialises the object.

VUnit GetVU2Up()

It returns the distance from AbsPos to the superior point.

VUnit GetVU2Dwn()

It returns the distance from AbsPos to the inferior point.

VUnit GetVU2Lft()

It returns the distance from AbsPos to the left point.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the right point.

int GetShift()

It returns for each clef the shift of the key signature with respect to the tremble clef.

int GetAltezza()

It returns for each clef the height of the key signature with respect to the tremble clef.

void SetPos(DrawObject *d, const Point& p)

Set the position of the key.

void Draw()

Draw method.

char *Describe(Context)

Return description of the class.

void print()

Printing method.

15.11 Class CTenore

15.11.1 Description

This is a symbol class that represents the symbol of the Tenor clef.

15.11.2 Father Class

Chiave

15.11.3 Public Methods

CTenore()

Initialises the object.

VUnit GetVU2Up()

It returns the distance from AbsPos to the superior point.

VUnit GetVU2Dwn()

It returns the distance from AbsPos to the inferior point.

VUnit GetVU2Lft()

It returns the distance from AbsPos to the left point.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the right point.

int GetShift()

It returns for each clef the shift of the key signature with respect to the tremble clef.

int GetAltezza()

It returns for each clef the height of the key signature with respect to the tremble clef.

void SetPos(DrawObject *d, const Point& p)

Set the position of the key.

void Draw()

Draw method.

char *Describe(Context)

Return description of the class.

void print()

Printing method.

15.12 Class CViolino

15.12.1 Description

This is a symbol class that represents the symbol of the Tremble clef.

15.12.2 Father Class

Chiave

15.12.3 Public Methods

CViolino()

Initialises the object.

VUnit GetVU2Up()

It returns the distance from AbsPos to the superior point.

VUnit GetVU2Dwn()

It returns the distance from AbsPos to the inferior point.

VUnit GetVU2Lft()

It returns the distance from AbsPos to the left point.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the right point.

int GetShift()

It returns for each clef the shift of the key signature with respect to the tremble clef.

int GetAltezza()

It returns for each clef the height of the key signature with respect to the tremble clef.

void SetPos(DrawObject *d, const Point& p)

Set the position of the key.

void Draw()

Draw method.

char *Describe(Context)

Return description of the class.

void print()

Printing method.

15.13 Class Tempo

15.13.1 Description

This is the class that represents the time in the musical measure. The time is represented by a fraction in which both at the numerator and at the denominator numbers with maximum two figures can appear. For the insertion of these figures we preferred an input procedure of the kind *Numeratore/Denominatore* by means of the keyboard. In this class there is a set of controls for avoiding the insertion of mistaken characters or in incorrect positions. We have also implemented the two exceptions in the time representation: 4/4 that is equivalent to a capital c and 2/2 that is equivalent to a dashed c.

The Tempo class contains two TNumerico classes (relation IS_PART_OF), by means of which the textual information inserted by the user is changed in two whole numbers: numerator and denominator.

To be noticed: the time is not necessarily always the same for a whole column of measures.

15.13.2 Father Class

DrawObject

15.13.3 Protected Attributes

TNumerico Numeratore, Denominatore

Numerical texts that represent the numerator and denominator.

VUnit TEMPO2Rgt

Distance from AbsPos of the right side of the rectangle containing the time.

Bool LetteraC

It assumes TRUE value if the time is to be represented with the *c* letter.

15.13.4 Public Methods

Tempo()

It initialises the object.

VUnit GetVU2Up()

It returns the distance from AbsPos to the superior point, that corresponds to the staff height.

VUnit GetVU2Dwn()

It returns the distance from AbsPos to the inferior point, that always corresponds to 0.

VUnit GetVU2Lft()

It returns the distance from AbsPos to the left point, that always corresponds to 0.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the right point.

Bool SetTempo(char tmp[])

It sets the time on the basis of the string pointed by tmp. The following strings are accepted:

- "c": it sets 4/4 designed with letter.
- "C": it sets 2/2 designed with letter.
- "*num/den*", where *num* and *den* are two whole numbers with a maximum of two figures.

int operator!=(Tempo tempo)

Redefinition of the *different* operator in order to confront the actual time with that of the previous measure.

void operator=(Tempo tempo)

Assignment operator.

void SetVU2Rgt(Vunit vu)

Set the distance from AbsPos to the right point.**virtual void SetColor(int col)**

Set the color.

int GetNumeratore()

Returns parameter Numeratore.

int GetDenominatore()

Returns parameter Denominatore.

void SetVU()

Set the object in the measure.

void SetPos(DrawObject *d, const Point& p)

Set the position of the object.

void Draw()

Draw time of the measure.

char *Describe(Context)

Returns the description of time in MusicTex.

Bool HasSameDesc(char tmp[])

Returns TRUE if the object has the same description of the passed parameter.

Bool HasSameDesc(Tempo*)

Returns TRUE if the object has the same description of the passed parameter.

void print()

Printing method.

15.14 Class Barra

15.14.1 Description

It is an abstract class that is used for representing the five possible bar lines of the measure. It is connected to Battuta by means of a relation of IRB. The length of the bar line can be found executing *GetVU2Rgt* (the method *GetVU2Lft* returns 0 for each type of bar line).

15.14.2 Father Class

DrawObject

15.14.2.1 Protected Attributes

Vunit ProlUp, ProlDwn

The lengths of the extensions of the bar lines over and under the staff.

15.14.3 Public Methods

Barra()

It initialises at o the extensions of the bar lines.

VUnit GetVU2Up()

It returns the distance from AbsPos to the superior point, that corresponds to the staff height.

VUnit GetVU2Dwn()

It returns the distance from AbsPos to the inferior point, that always corresponds to 0.

VUnit GetVU2Lft()

It always returns 0.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the right point.

virtual tipoBarra GetTipoBarra()

It returns the kind of bar line that has been set.

void Setup (Vunit prUp, Vunit prDwn)

It sets the length of Barra.

void SetPos (DrawObject*, const Point&)

It sets the position of Barra.

void Draw ()

Necessary for the “virtual” definition of Draw() in DrawObject.

virtual char *Describe(Context)

It returns the description MusicTex of the object.

virtual void print()

Printing method.

15.15 Class BDoppia

15.15.1 Description

This is a symbol class that represents the symbol of the double bar line.

15.15.2 Father Class

Barra

15.15.3 Public Methods

BDoppia()

It initialises the object.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the right point.

tipoBarra GetTipoBarra()

It returns the kind of bar line that has been set.

void Draw()

Draw method.

char *Describe(Context)

Return description of the class.

void print()

Printing method.

15.16 Class BFinale

15.16.1 Description

This is a symbol class that represents the symbol of the final bar line.

15.16.2 Father Class

Barra

15.16.3 Public Methods

BFinale()

It initialises the object.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the right point.

tipoBarra GetTipoBarra()

It returns the kind of bar line that has been set.

void Draw()

Draw method.

char *Describe(Context)

Return description of the class.

void print()

Printing method.

15.17 Class BFineRit

15.17.1 Description

This is a symbol class that represents the symbol of the bar line that determines the end of the refrain.

15.17.2 Father Class

Barra

15.17.3 Public Methods

BFineRit()

It initialises the object.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the right point.

tipoBarra GetTipoBarra()

It returns the kind of bar line that has been set.

void Draw()

Draw method.

char *Describe(Context)

Return description of the class.

void print()

Printing method.

15.18 Class BInizioRit

15.18.1 Description

This is a symbol class that represents the symbol of the bar line that determines the beginning of the refrain.

15.18.2 Father Class

Barra

15.18.3 Public Methods

BInizioRit()

It initialises the object.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the right point.

tipoBarra GetTipoBarra()

It returns the kind of bar line that has been set.

void Draw()

Draw method.

char *Describe(Context)

Return description of the class.

void print()

Printing method.

15.19 Class BSingola

15.19.1 Description

This is a symbol class that represents the symbol of the single bar line.

15.19.2 Father Class

Barra

15.19.3 Public Methods

BSingola()

It initialises the object.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the right point.

tipoBarra GetTipoBarra()

It returns the kind of bar line that has been set.

void Draw()

Draw method.

char *Describe(Context)

Return description of the class.

void print()

Printing method.

15.20 Class BInizioFine

15.20.1 Description

This is a symbol class that represents the symbol of the beginning-final bar line .

15.20.2 Father Class

Barra

15.20.3 Protected Attributes

BfineRit bf

Class BfineRit.

BInizioRit bi

Class BInizioRit.

15.20.4 Public Methods

BInizioFine()

It initialises the object.

VUnit GetVU2Rgt()

It returns the distance from AbsPos to the right point.

tipoBarra GetTipoBarra()

It returns the kind of bar line that has been set.

void SetPos (DrawObject*, const Point&)

It sets the position of Barra.

v**irtual void SetColor (int col)**

It sets the color.

void Draw()

Draw method.

char *Describe(Context)

Return description of the class.

void print()

Printing method.

15.21 Class BInizioRit

15.21.1 Description

This is a symbol class that represents the symbol of the beginning chorus bar line .

15.21.2 Father Class

Barra

15.21.3 Public Methods

BInizioRit()
It initialises the object.
VUnit GetVU2Rgt()
It returns the distance from AbsPos to the right point.
tipoBarra GetTipoBarra()
It returns the kind of bar line that has been set.
void Draw()
Draw method.
char *Describe(Context)
Return description of the class.
void print()
Printing method.

15.22 Class Cbasso8

15.22.1 Description

This is a symbol class that represents the symbol of the Bass-8 clef.

15.22.2 Father Class

Chiave

15.22.3 Public Methods

Cbasso8()
Initialises the object.
VUnit GetVU2Up()
It returns the distance from AbsPos to the superior point.
VUnit GetVU2Dwn()
It returns the distance from AbsPos to the inferior point.
VUnit GetVU2Lft()
It returns the distance from AbsPos to the left point.
VUnit GetVU2Rgt()
It returns the distance from AbsPos to the right point.
int GetShift()
It returns for each clef the shift of the key signature with respect to the treble clef.
int GetAltezza()
It returns for each clef the height of the key signature with respect to the tremble clef.
void SetPos(DrawObject *d, const Point& p)
Set the position of the key.
void Draw()
Draw method.
char *Describe(Context)
Return description of the class.

15.23 Class Cviolino8

15.23.1 Description

This is a symbol class that represents the symbol of the Treble-8 clef.

15.23.2 Father Class

Chiave

15.23.3 Public Methods

Cviolino8()
Initialises the object.
VUnit GetVU2Up()
It returns the distance from AbsPos to the superior point.
VUnit GetVU2Dwn()
It returns the distance from AbsPos to the inferior point.
VUnit GetVU2Lft()
It returns the distance from AbsPos to the left point.
VUnit GetVU2Rgt()
It returns the distance from AbsPos to the right point.

int GetShift()

It returns for each clef the shift of the key signature with respect to the tremble clef.

int GetAltezza()

It returns for each clef the height of the key signature with respect to the tremble clef.

void SetPos(DrawObject *d, const Point& p)

Set the position of the key.

void Draw()

Draw method.

char *Describe(Context)

Return description of the class.

15.24 Class Giustificazione

15.24.1 Description

This is a symbol class that represents the methods and structures used in justification and line breaking task.

15.24.2 Father Class

None

15.24.3 Protected Attributes

Gelemento **Sims

Matrix of elements to represent the simultaneities

Gelemento is a struct declared as follows:

```
struct Gelemento  {
  public:
     double beat;
     Figura* fig;
     UL space;
     double fraz;
}
```

Battuta **Bats

Array of pointers to measures. Each column of Sims represents a voice. The index corresponding

pointer of Bats array poits to the belonging measure.

int ncolonne

Columns of matrix Sims.

int nrighe

Rows of matrix Sims.

int nmaxcolonne

Maximum number of columns of matrix Sims.

float den

Denominator of time key.

15.24.4 Public Methods

Giustificazione()

Initialises the object.

~**Giustificazione()**

Destroyer.

void Costruisci (int nr, int nc)

It allocates memory for the matrix and initialises it.

void Inserimento (Battuta*)

It inserts the figures of a new measure in the last columns of the matrix, one column for voice.

void Sincronizza (Battuta*)

It synchronizes the structure inserting NoOperation in the beatlines (column of the matrix).

void dump ()

It prints on file the matrix Sims: for debugging session.

void GiustificaLog (double kGiust)

It sets the space fields of the matrix for a logarithmic justification.

void GiustificaLinog (double kGiust)

It sets the space fields of the matrix for a linear justification.

void SetUpSpG ()

It sets the space fields of the figures of the measure for the directorial view.

void SetUpSpOr ()

It sets the space fields of the figures of the measure for the musician view.

void SetUpSpDLine ()

It sets the space fields of the figures of the measure for the line breaking.

16 BRA Module

16.1 Class ParGraffa

16.1.1 Description

The present class provides the representation of a brace bracket with free height and fixed width. The design is obtained through straight sliding and scale change of the function $x = y^2 - y^3$, $-1/3 \leq y \leq 1$, which is accompanied for a certain while by a function proportional to it in order to make thicker the line of the brace. In order not to calculate the black pixel by each redraw, each time that the height of the brace bracket is changed, a bitmap description of the superior half of the brace is saved.

16.1.2 Father Class

DrawObject

16.1.3 Protected Attributes

VUnit Width, Height

Width and height of the brace bracket.

unsigned char *BitMap

Pointer to an array of byte that represents a bitmap description of the superior half of the bracket. The array is long as half height of the bracket; each byte describes an horizontal line from left to right: bit 1 ≡ black, bit 0 ≡ white.

16.1.4 Public Methods

ParGraffa()

It initialises the height at 0 and the width at the fixed value.

ParGraffa(VUnit h)

It initialises the width at the fixed value and the height at *h*, computing the array pointed by BitMap.

~**ParGraffa()**

Deallocates the array of byte.

void Init(VUnit h)

It sets the height at *h*; if this is different from the previous height, it computes the array of byte pointed by BitMap.

VUnit GetVU2Up()

It returns *Height*.

VUnit GetVU2Dwn()

It returns 0.

VUnit GetVU2Rgt()

It returns *Width.*

VUnit GetVU2Lft()

It returns 0.

Draw()

It draws the brace brackets on the basis of the bitmap description.

void print();

int Save(FILE*,Context);

for the saving

char *Describe();

17 DRW Module

In this module there are classes for the low level management of the graphic objects, and classes for the GUI.

17.1 Class DrawObject

17.1.1 Description

This is the root class for all the symbols that have to be displayed on the screen or printed on sheets.

17.1.2 Father class

NONE

17.1.3 Connected Types

17.1.4 Protected attributes:

ClassID ID

ID for graphic object identification

short Color

The color to be used for drawing the object.

short OldColor;

The old color, is used for temporary highlight.

Point AbsPos;

Absolute position of the centre of the object

17.1.5 Public methods

DrawObject()

Constructor

virtual ~DrawObject()

Destructor

operator Rectangle()

Convert the Draw Object in a rectangle

ClassID GetID()

Return the class identifier

void SetID(ClassID id)

Set the class identifier

virtual void SetColor(int col)

Set the colour

int GetColor();

Return the colour

virtual void SetOldColor(int col)

virtual int GetOldColor(void)

virtual VUnit GetVU2Up() = 0

Return the distance between AbsPos and the upper margin

virtual VUnit GetVU2Dwn() = 0
Return the distance between AbsPos and the bottom margin

virtual VUnit GetVU2Lft() = 0
Return the distance between AbsPos and the left margin

virtual VUnit GetVU2Rgt() = 0
Return the distance between AbsPos and the right margin

VUnit GetWidth()
Return the object width

VUnit GetHeight()
Return the object height

virtual void SetPos(DrawObject *d,const Point& p)
Set the object position relative at drawobject ' d '. If d == NULL the position is set to ' p '

virtual void Move(const Point& p)
Change the object position on the display

Point GetAbsPos()
Return the object absolute position

Bool is(ClassID)
TRUE if the object is a DrawObject

virtual void Draw() = 0
Draw the object in AbsPos

virtual Bool Hit(const Point& apos)
Return TRUE if the object is selected

virtual Bool Hit(const Point&, SelObj, DrawObject*&, SymPath&)
Other way to select an object

virtual Bool Hit(const Rectangle&)
Return TRUE if the object is inside the rectangle

virtual Bool Hit(const Rectangle&, SelObj, SymPath&, SymPath&)
Other way to select an object (using a rectangle)

virtual Bool SymbolicHit(SymPath&, SelObj, DrawObject*&)
Decode a path

virtual void Select()
Select an object, set the selection colour and redraw it

virtual void Deselect()
Deselect an object, set the colour to the default and redraw it

virtual CmdResult Command(CmdID,CmdArgV)
Execute the CmdID command with the specified arguments
Return CMD_OK if the command was successfully executed, CMD_FAIL if the execution fail and CMD_UNKNOWN if the command was not recognised

virtual CmdResult SymCommand(SymCmd&, Battuta* pBattuta=NULL)

virtual Bool DeleteSym(DrawObject*)
Delete the specified symbol. Return TRUE if the operation is performed correctly.

17.2 Class GDEVICE

17.2.1 Description

This is an abstract class central in the management of the display. It defines some tools needed for the graphica manipulation of the objects. It presents the generic interface for drawing on screen and for printing.

17.2.2 Father class

NONE

17.2.3 Connected Types

enum ColorIcon
Define the colors for the graphics objects and can have the following values:

BIANCO	(white)
NERO	(black)
VERDE	(green)
BLU	(blue)
ROSSO	(red)
GRIGIO	(gray)
GRIGIOS	(dark gray)
GIALLO	(yellow)

enum DrwMode

```
DefMode = 3,            // Standard
XORMode = 4,            // XOR tra il codice colore e quello del punto
ANDMode = 1,            // Come sopra ma con AND
ORMode = 0    // Come sopra ma con OR
```

struct Carattere
A structure that define any character from the font

int code	Char code
int height	Char height
int width	Char width
int realwidth	Real char width
int nbytes	Bytes needed for storage
int ascend	Char ascent
unsigned char *image;	Binary immage of the char coded from top

int VU2UP, VU2DWN, VU2RG, VU2LF, DX, DY;
definition of the selection rectangle and the char positioning inside it

17.2.4 Protected attributes

LiooFont *lista_font[4]
Available font for the setted display
int activeFont
The ID of the active font
Point currentPos;
Current position

Point activeInitTextPos
String to be printed position

17.2.5 Public methods

GDevice()
Constructor

~GDevice();
Destructor

virtual void Clear()
Clear the display

virtual int GetMaxPixX()
Return the maximum number of pixel in x-axis

virtual int GetMaxPixY()
Return the maximum number of pixel in y-axis

virtual int GetNumColors()
Return the maximum number of colors

virtual void SetColor(int)
Set the color

virtual int GetColor()
Get the current color

virtual void SetDrwMode(DrwMode)
Set the Draw mode

virtual void PutPixel(const Point &p);
Draw a pixel in the point p and return the point color before drawing

void MoveTo(const Point &p)
void MoveTo(VUnit x,VUnit y)
Set the current position using a point or the co-ordinates.

Point CurrentPos()
Return the current position.

virtual void LineTo(const Point& p)
virtual void LineTo(VUnit x,VUnit y)
Draw a line from the current position to the given point.
Use the line style defined with SetLineStyle and the color defined with SetColor.

virtual void Line(const Point& p1, const Point& p2)
Draw a line between two points

virtual void BigLine(const Point& p1, const Point& p2, VUnit width)
Draw a line between two points using the given width

virtual void Rect(const Rectangle& r,Bool isfilled=false)
Draw a rectangle (filled or not)

virtual void ShadowRect(const Rectangle& r,Bool isOn)

Draw a rectangle with shadow effect
isOn=TRUE the rectangle is raised.
isOn=FALSE the rectangle is embossed.

virtual void Arc(const Rectangle& r, short Angle1, short Angle2, Bool isfilled=false)

Draw an arc for the ellipse inserted in the reactangle 'r' from the angle Angle1 until Angle2 (in degrees)
Optionally is should be filled

virtual void Ellipse(const Rectangle& r,Bool isFilled=false)

Draw an arc for the ellipse inserted in the reactangle 'r'
Optionally is should be filled

FontID SetFont(FontID)

Set the font to FontID

FontID GetActiveFont()

Return the FontID of the active font

int GetFontAscent()

Return the font ascent.

int GetFontHeight()

Return the font height

void TextTo(const Point &p)
void TextTo(VUnit x,VUnit y)

Set the current text position in order to have the font baseline in the given position

virtual void OutText(unsigned char *)

Print a string with the current font

int GetTextWidth(unsigned char *)

Return the width of a string (pixels)

int GetTextHeight(unsigned char *txt)

Return the string height (pixels).

long ImageSize(const Rectangle&)

Return the bytes needed to store the image from the indicated

virtual void PutImage(const Point&,void *buf,int w,int h,DrwMode)

Redraw, starting from the point 'p', the image stored in the buffer 'buf' using the DrwMode

int Get_VU2UP(int carcode)
int Get_VU2DWN(int carcode)
int Get_VU2RG(int carcode)
int Get_VU2LF(int carcode)
int Get_VU2DX(int carcode)
int Get_VU2DY(int carcode)

Return the 6 fields of the "Carattere" struct. It seems to be neccessary only for the music font

17.3 Class GSCREEN

17.3.1 Description

This is an implementation of the GDevice interface for the management of the display. It defines some tools needed for the graphical representation of the objects.

17.3.2 Father class

GDevice

17.3.3 Connected Types

enum ColorIcon

Define the colors for the graphics objects and can have the following values:

BIANCO	(white)
NERO	(black)
VERDE	(green)
BLU	(blue)
ROSSO	(red)
GRIGIO	(gray)
GRIGIOS	(dark gray)
GIALLO	(yellow)

enum DrwMode

```
DefMode = 3,          // Standard
XORMode = 4,          // XOR
ANDMode = 1,          // AND
ORMode = 0    // OR
```

struct Carattere

A structure that define any character from the font

int code	Char code
int height	Char height
int width	Char width
int realwidth	Real char width
int nbytes	Bytes needed for storage
int ascend	Char ascent
unsigned char *image;	Binary image of the char coded from top

int VU2UP, VU2DWN, VU2RG, VU2LF, DX, DY;

definition of the selection rectangle and the char positioning inside it

17.3.4 Private attributes:

Mouse *mouse;

Mouse manager

int activeFont;

FontID of the active font.

Rectangle activeClipRegion;

Active clipping region

Bool KeyPressed;

True if key pressed

int tmpKey;

temp key code, communicated by Call_KPress()

int lastKey;

last key code obtained with WaitKey()

Point currentPos;
Current position

Point activeInitTextPos
The position of the string to be printed

Point MousePoint;
Mouse position given by Call_MMoved()

int MouseAction;
Last mouse action

Bool ActionDetected;
Is TRUE if the last mouse action was read with MousePos()

CursorTypes CurType;

LPixmap *Copia;
LPixmap *Disegno;
memory bitmaps used during execution

17.3.5 Public methods

GScreen();
Constructor
~GScreen()
Destructor

void GMoodsOpenMainScore(const char *title, int reader);
It opens a new frame for .

void GMoodsOpenPart(const char *title, int reader);

Bool GOpen(int argc, char **argv)
Open the graphic display and register the standard font

void GClose()
Close the graphic display

Widget GNewWindow(const char *title = NULL)
Open a new window and return the Widget, if errors return NULL

Widget GNewDrawArea(int x, int y, VUnit height, VUnit width, Widget widget,char *title=NULL)
Open a new draw area and return the Widget, if errors return NULL

void GSetMainDrawArea()
Set the main window as active

void GSetDrawArea(Widget)
Set the specified draw area as active

void GSetFocusWindow(Widget)
Set the focus on the specified window

void GCloseWindow(Widget)
Close the specified window and reactivate the main one

void Clear()
Clear the display

virtual int GetMaxPixX()
Return the maximum number of pixel in X axis

virtual int GetMaxPixY()
Return the maximum number of pixel in Y axis

virtual int GetNumColors()
Return the maximum number of colours

virtual void SetColor(int)
Set the colour

virtual int GetColor()
Get the current colour

virtual void SetDrwMode(DrwMode)
Set the Draw mode

Rectangle SetClipRegion(const Rectangle&)
Set the draw area that can be modified by the graphic functions. Return the previous active area

void PutPixel(const Point &p);
Draw a pixel in the point ' p '

void PutPixel0Up(const Point &p);
Draw a pixel in the point ' p ' considering the axis origin in the top left corner
Normally the axis origin is placed in the bottom left corner

int GetPixel(const Point &);
Return the point colour

void MoveTo(const Point &p)
void MoveTo(VUnit x,VUnit y)
Set the current position using a point or the co-ordinates.

Point CurrentPos()
Return the current position

virtual void LineTo(const Point& p)
virtual void LineTo(VUnit x,VUnit y)
Draw a line from the current position to the given point.
Use the line style defined with SetLineStyle and the color defined with SetColor.

virtual void Line(const Point& p1, const Point& p2)
Draw a line between two points

virtual void BigLine(const Point& p1, const Point& p2, VUnit width)
Draw a line between two points using the given width

virtual void Rect(const Rectangle& r,Bool isfilled=false)
Draw a rectangle (filled or not)

virtual void ShadowRect(const Rectangle& r,Bool isOn)
Draw a rectangle with shadow effect

isOn=TRUE the rectangle has to be raised.
isOn=FALSE the rectangle has to be embossed.

virtual void Arc(const Rectangle& r, short Angle1, short Angle2, Bool isfilled=false)
Draw an arc for the ellipse inserted in the rectangle 'r' from the angle Angle1 until Angle2 (in degrees)
Optionally is should be filled

virtual void Ellipse(const Rectangle& r,Bool isFilled=false)
Draw an arc for the ellipse inserted in the rectangle 'r'
Optionally is should be filled

FontID SetFont(FontID)
Set the font to FontID

FontID GetActiveFont()
Return the FontID of the active font

int GetFontAscent()
Return the font ascent.

int GetFontHeight()
Return the font height

void TextTo(const Point &p)
void TextTo(VUnit x,VUnit y)
Set the current text position in order to have the font baseline in the given position

virtual void OutText(unsigned char *)
Print a string with the current font

int GetTextWidth(unsigned char *)
Return the width of a string (pixels)
int GetTextHeight(unsigned char *txt)
Return the string height (pixels).

int WaitKey()
Wait a key to be pressed and return it's value

void SetTmpKey(int key);
Set the tmpKey to key and the KeyPressed to TRUE

int KeyHit()
Return TRUE if was pressed a key that can be read with WaitKey method.
Deprecated.

int LastKey();
Return the last key read with WaitKey().
Deprecated.

long ImageSize(const Rectangle&)
Return the bytes needed to store the image from the indicated
Deprecated.

void GetImage(const Rectangle&,void *buf);
Fill the buffer with the data about the image from the rectangle ' r '
Deprecated.

virtual void PutImage(const Point&,void *buf,int w,int h,DrwMode)

Draw, starting from the point 'p', the image stored in the buffer 'buf' using the DrwMode
Deprecated.

void GSetCursor (CursorTypes ct, int fgc = NERO, int bgc = BIANCO);

CursorTypes GGetCursor();

void SetMouse(Mouse* m)
Set the mouse manager

Mouse *GetMouse()
Return the mouse manager

void SetMousePoint(const Point &p)
Set the mouse point to ' p '

void SetMouseAction(int act)
Set the mouse action to ' act '

int MousePos(Point& pm)
Put the mouse position in ' pm ' and return the mouse buttons status

int Get_VU2UP(int carcode)
int Get_VU2DWN(int carcode)
int Get_VU2RG(int carcode)
int Get_VU2LF(int carcode)
int Get_VU2DX(int carcode)
int Get_VU2DY(int carcode)
Return the 6 fields of the "Carattere" struct. It seems to be necessary only for the music font

void CopyPixmap (Widget sorg,int x,int y,int width,int height)
copy the ' sorg ' window into the current one

LPixmap* GetLPixmapCopia();
get the “Copia” bitmap.

LPixmap* GetLPixmapDisegno();

void CopyPixmap (LPixmap *pixmap,int x,int y,int width,int height);

void SwapPixmap();

void GResetPixmap();

void GSetPixmap (LPixmap *lpixmap);

void GNewDC(Widget)

void GDeleteDC()

17.4 Class GPRINT

17.4.1 Description

This is a class derived from GDEVICE that control the generation of the printed sheet.
It implements the interface of GDevice for postscript generation.

Warning: by now only the *Write* method has been implemented.

17.4.2 Father class

GDevice

17.4.3 Connected Types

enum ColorIcon

Define the colours for the graphics objects and can have the following values:

BIANCO	(white)
NERO	(black)
VERDE	(green)
BLU	(blue)
ROSSO	(red)
GRIGIO	(grey)
GRIGIOS	(dark grey)
GIALLO	(yellow)

enum DrwMode

```
DefMode = 3,        // Standard
XORMode = 4,        // XOR tra il codice colore e quello del punto
ANDMode = 1,        // Come sopra ma con AND
ORMode = 0          // Come sopra ma con OR
```

struct Carattere

A structure that define any character from the font

int code	Char code
int height	Char height
int width	Char width
int realwidth	Real char width
int nbytes	Bytes needed for storage
int ascend	Char ascent
unsigned char *image;	Binary immage of the char coded from top

int VU2UP, VU2DWN, VU2RG, VU2LF, DX, DY;
definition of the selection rectangle and the char positioning inside it

17.4.4 Private attributtes

FILE *PSfile, *pfafile_musicfont

Two files needed as PS output and a pfa file with the music font definition

double PrintUnit

during the printing there are a conversion between PostScript unit and inch

int PentNo

define the number of staves in the page

17.4.5 Protected attributes

LiooFont *lista_font[4]

Available font for the setted display

int activeFont

The ID of the active font

Point currentPos;
Current position

Point activeInitTextPos
String to be printed position

17.4.6 Public methods

GPrint()
Constructor

~GPrint();
Destructor

void PrnOpen (char *fname)
open the moodsPS.ps file in the " LIOO_MDS_DIR " directory and create the PostScript header
LIOO_MDS_DIR must be set as an environment variable

void PrnClose ()
close the moodsPS.ps file

void Write(char *,...)
insert a string in the PS file with a printf style format.

void SetPrintUnit(double)
set the print unit

double GetPrintUnit()
Return the print unit

void SetPentNo(int)
Set the staves number on a single page

int GetPentNo()
Return the staves number on a single page

virtual void Clear()
Clear the display

virtual int GetMaxPixX()
Return the maximum number of pixel in X axis

virtual int GetMaxPixY()
Return the maximum number of pixel in Y axis

void MoveTo(const Point &p)
void MoveTo(VUnit x,VUnit y)
Set the current position using a point or the co-ordinates.

Point CurrentPos()
Return the current position.

virtual void LineTo(const Point& p)
virtual void LineTo(VUnit x,VUnit y)
Draw a line from the current position to the given point.
Use the line style defined with SetLineStyle and the colour defined with SetColor.

virtual void Line(const Point& p1, const Point& p2)
Draw a line between two points

void BigLine(const Point& p1, const Point& p2, VUnit width)
Draw a line between two points with the given width

virtual void Rect(const Rectangle& r,Bool isfilled=false)
Draw a rectangle (filled or not)

virtual void Arc(const Rectangle& r, short Angle1, short Angle2, Bool isfilled=false)
Draw an arc for the ellipse inserted in the rectangle ' r ' from the angle Angle1 until Angle2 (in degrees)
Optionally is should be filled

virtual void Ellipse(const Rectangle& r,Bool isFilled=false)
Draw an arc for the ellipse inserted in the rectangle ' r '
Optionally is should be filled

virtual void OutText(unsigned char *)
Print a string with the current font

int GetTextWidth(unsigned char *)
Return the width of a string (pixels)

int GetTextHeight(unsigned char *txt)
Return the string height (pixels).

virtual void PutImage(const Point&,void *buf,int w,int h,DrwMode)
Draw, starting from the point 'p', the image stored in the buffer 'buf' using the DrwMode

18 FIG module

The FIG module contains the classes that manage the musical figures of note and rest. According to the setting of the LIOO analysis, there is a class for each musical symbol, therefore in this module we have a class for each kind of note and one for each kind of rest. In order to collect the methods and attributes common to the classes related to the notes, the abstract class Nota has been introduced, as well as the class Pausa concerning rests. Nota and Pausa are sons of Figura, an abstract class from which all the other classes of the module derive. Among these there are Accordo and GruppoNote, that manage these particular kind of multiple figures (chord and beam), Ripetizione, with the corresponding sons, and Spazio, that contributes to the correct placement of the figure in the staff.

18.1 Class Figura

18.1.1 Description

The abstract class Figura contains the methods and the most general attributes to represent notes and rests on the screen and to recall the representation of the symbols connected to notes and rests.

In the musical vocabulary the definition of **figure** is *each notation sign corresponding to different duration values of notes and rests.* The figures of the modern notation are nine: breve note, whole note, half note, quarter note, 8th note, 16th note, 32nd note, 64th note and 128th note. For each figure, two classes have been created: one for representing the note (e.g. NCroma) and one for representing the corresponding rest (e.g. PCroma). The exception, among the rest classes, are the classes PausaDueBattute, PausaQuattroBattute and PausaGenerica that have no correspondence among the notes. All the classes of this kind are descended from figure but not directly. In effect in order to manage the classes related to the notes the class Nota has been introduced, subclass of Figura, and the same has been made for the rests with the class Pausa, subclass of Figura. Two classes, Accordo and GruppoNote, have been introduced as figures. Even if in the musical theory chords and beams do not represent figures, they have been inserted as descended from Figura because they follow representation principles that are the same of those of the single figures and because in this way the list of the objects composing a measure contains objects of one type. For the same reason we found as figure the class Ripetizione specialised in RipetizioneTempo, RipetizioneMezzaBattuta and RipetizioneBattuta.

By means of Figura all classes of Nota, Pausa, Accordo, GruppoNote e Ripetizione become DrawObject. Obviously these classes do not cover all the range of musical symbols that can be represented by our lectern. The other symbols, that are DrawObject too, are each created by means of their own class. The class of the symbols referred to a unique figure (and they are the majority) is connected to Figura or Nota by means of a relation IS_REFERRED_BY, that is the direct translation of the link that, in the musical theory, exists between symbol and figure that it refers to. This implies that Figura and its descended must organise the disposition of the symbols around themselves (it is the task of method Adjust): the only exception is represented by the symbol Tremolo whose disposition is performed by the Battuta class.

The symbols connected to Figura are:
Corona (Fermata) It is referred to Figura because can act both on a note or on a rest.
PuntoValore (Augmentation Dot) It is referred to Figura because can act both on a note or on a rest.
Occhiali (Glasses) It is referred to Figura because can be associated to both a note or a rest.
Strumento (Instrument) It is referred to Figura because can be associated to both a note or a rest.
TGenerico (Generic Text) It has a lasting effect, thus in the practice it influences many figures. It has a relation IS_REFERRED_BY with figure since it has to be placed below or above the figure from which the effect must begin.
Annotazione (Annotation) It has an effect lasting in time, thus in the practice it influences many figures. It has a relation IS_REFERRED_BY with figure since it has to be placed below or above the figure from which the effect must begin.
TDinamico The relation with Figura permits that, for example, a mezzoforte (dynamic indication) is placed on a rest. This feature can be surprising but it allows a more “flexible” management of the dynamic signs. In effect, if a dynamic sign is on a rest it is intended that it will be an effect starting from

the notes that follow the rest. If these notes have already plenty of symbols, e.g. ornaments and fingering, it results useful to move backward the dynamic sign without changing its meaning.

The most usual procedure consists in designing these symbols and those referred to Nota in a precise position with respect to the figure. If the augmentation dot must be placed immediately on the right of the figure and the accidentals (see the Nota class) immediately on the left, for the remaining symbols the usual habit is followed: we put, for example, the ornaments and the fingering (see Nota class) above the note and the dynamic signs and agogic (generic texts and annotations) below the note. For the sake of completeness we have decided that each symbol that can be designed on a note can be designed under it too. This possibility is indispensable when we write polyphonic music, that is to say, when we use the layers (*voci* in musical vocabulary) offered by LIOO. In this case the user will write above the staff all the symbols referred to the upper layer and below the staff all those referred to the inferior layer. Among all the symbols connected to Figura and Nota only the augmentation dot and the accidental can be written inside the staff: the others must be placed above or below.

Figura uses the attributes VU2Up (which means *Visual Unit to Up* and is measured, like the others, starting from AbsPos), VU2Dwn, VU2Lft and VU2Rgt in order to describe the rectangle (that as a rule will not be designed) that surrounds the figure with its symbols. To organise the symbols around it, Figura uses another rectangle, smaller than the first, that delimits the space occupied by the figure without symbols. This second rectangle is defined by the attributes VU2UpF, VU2DwnF, VU2LftF, VU2RgtF, always referred to AbsPos of the figure. If no symbol is connected to the figure in question these attributes coincide to those previously introduced.

A possible exploitation of the methods of this class is described in the sequence of the calls to methods (to be remembered is that Figura is an abstract class, therefore it will be never instanced: the following example is useful to understand the classes derived from figure):

1. new Figura (Init if we don’t use the constructor).
2. SetAltezza (set height: the default height is 0).
3. SetTDinamico.
4. SetPPuntoValore.
5. SetPos (it works according to the height).
6. Draw.
7. deleteSym.
8. SetPos.
9. Draw.
10. ~ Figura (eventually).

After setting a figure and modifying it, it is mandatory to call Adjust before designing it. The only exception is when the setting or the modification are mad with SetPos, because SetPos performs Adjust. It is therefore recommended to perform all the changes needed and execute SetPos immediately before Draw. This last indication is valid for all the classes of this module. The only difference with Figura is that the other classes are richer in methods.

In general is very useful to know the horizontal symbols that are starting, ending or over a figure.
For this reason in the Figura object has an array of pointers to the horizantal symbols (sons of IntEsteso) starting, ending or over the figure, for this task the struct IntRef is present:

Struct IntRef

IntRefType type

the type of reference, if can be:

INT_START	the horiz. symbol is starting from the figure
INT_OVER	the horiz. symbol is "over" the figure
INT_END	the horiz. symbol is ending to the figure

IntEsteso *intRef

the pointer to the horiz. symbol

18.1.2 Father class

DrawObject

18.1.3 Children classes

Nota, Pausa, GruppoNote, Ripetizione

18.1.4 Protected attributes

int Altezza

Musical height of the figure with respect to the note that occupies the inferior line of a staff (Mi in the tremble clef). For example in the tremble clef the Sol of the second line has Altezza=2. Further explanation on the use of this attribute can be found in the documentation of the Nota and Pausa classes.

VUnit VU2Up, VU2DwnF, VU2LftF, VU2RgtF

Distances from AbsPos of the upper, inferior, right and left points of the rectangle containing the figure and all the symbols connected to it.

Corona *PCorona

Pointer to the fermata associated to the figure.

Occhiali *POcchiali

Pointer to the sign of Occhiali (Glasses) associated to the figure.

PuntoValore *PPuntoValore

Pointer to the augmentation dot associated to the figure.

TDinamico *PTDinamico

Pointer to the dynamic sign associated to the figure.

TGenerico *PTGenerico

Pointer to the generic text associated to the figure.

Annotazione *Pannotazione

Pointer to the sign of annotation associated to the figure.

Strumento *PStrumento

Pointer to the instrument associated to the figure.

NumCode NumericCode

Numeric code of the figure (the type corresponds to short).

LegatQuadra *terz

Symbol to group together an irregular group.

float durata

Useful to compute duration of figures in irregular groups.

int NumSpaces

Deprecated.

Bool Figurina

Returns TRUE if figure is a grace note or grace pause.

18.1.5 Private attributes

UL SpG, SpOr, SpDirUt, SpOrcUt

Spacing attributes used for justification and formatting:

1) SpG – Spacing Attributes Directorial
   These are the spacing attributes for the view of the director, be calculated with logarithmic
   justification as well as with linear justification.
2) SpOr – Spacing Attributes Musician
   These are the spacing attributes for the view of the single musician, be calculated with logarithmic
   justification as well as with linear justification.
3) SpDirUt – Spacing Attributes manually introduced by the user in the view of the director.
4) SpOrcUt – Spacing Attributes manually introduced by the user in the view of the director.

The director the visualises SpG+SpDirUt, the musician visualises SpOr+SpOrcUt.

Spacing attributes are expressed in Logic Unit (UL) defined as follow:

1 UL = 1/600 staff space = 1/100 pixel

Represent the spaces until the next figure

UL SpDLine, SpOLine
Spacing attributes used for line breaking. Used only in runtime, not serialised.

18.1.6 Private methods

CmdResult CmdFiguraCorona (Bool AboveF)
It sets the fermata to the figure. If it is successful, it returns CMD_OK, otherwise CMD_FAIL. It is necessary to specify AboveF=TRUE for the fermata above the figure or FALSE for the fermata below the figure.
CmdResult CmdFiguraOcchiali (Bool AboveF)
It sets the glasses to the figure. If it is successful, it returns CMD_OK, otherwise CMD_FAIL. It is necessary to specify AboveF=TRUE for the glasses above the figure or FALSE for the glasses below the figure.
CmdResult CmdFiguraTDinamico (Bool AboveF, TDinamicType td)
It assigns to the figure the dynamic sign specified by the td parameter. The available dynamic texts are: *pppp, ppp, pp, p, ffff, fff, ff, f, mp, mf, sp, sf, sfz, fz, fp*. If it is successful, it returns CMD_OK, otherwise CMD_FAIL. It is necessary to specify AboveF=TRUE for designing this sign above the figure or FALSE for designing it below the figure.
CmdResult CmdFiguraTGenerico (Bool AboveF, char * s)
It assigns to the figure the text given as a parameter. If it is successful, it returns CMD_OK, otherwise CMD_FAIL. It is necessary to specify AboveF=TRUE for designing this sign above the figure or FALSE for designing it below the figure.
CmdResult CmdFiguraAnnotazione (Bool AboveF, char * s)
It assigns to the figure the text given as a parameter. If it is successful, it returns CMD_OK, otherwise CMD_FAIL. It is necessary to specify AboveF=TRUE for designing this sign above the figure or FALSE for designing it below the figure.
void AdjustPuntoValore ()
It sets the augmentation dot on the right of the figure. If the augmentation dot falls on a line of the staff or
additional line, it is moved in the space just above. The bounding box of the figure is enlarged enough to
contain this symbol too.
void AdjustCorona (Vunit Ylinea1, Vunit Ylinea5)
It sets the corona symbol above the note or below, depending on the value of the symbol AboveNota (TRUE =
above, FALSE = below). Input parameters Ylinea1 and Ylinea5 are the effectikve heights (in Vunit), of the
lowest and highest line of the staff. The bounding box of the figure is enlarged enough to
contain this symbol too.
void AdjustOcchiali (Vunit Ylinea1, Vunit Ylinea5)
It sets the occhiali symbol above the note or below, depending on the value of the symbol AboveNota (TRUE
= above, FALSE = below). Input parameters Ylinea1 and Ylinea5 are the effectikve heights (in Vunit), of the
lowest and highest line of the staff. The bounding box of the figure is enlarged enough to contain this symbol
too.
void AdjustTDinamico (Vunit Ylinea1, Vunit Ylinea5)
It sets TDinamico symbol above the note or below, depending on the value of the symbol AboveNota (TRUE
= above, FALSE = below). Input parameters Ylinea1 and Ylinea5 are the effectikve heights (in Vunit), of the

lowest and highest line of the staff. The bounding box of the figure is enlarged enough to contain this symbol too.

void AdjustTGenerico (Vunit Ylinea1, Vunit Ylinea5)

It sets the TGenerico symbol above the note or below, depending on the value of the symbol AboveNota (TRUE = above, FALSE = below). Input parameters Ylinea1 and Ylinea5 are the effectikve heights (in Vunit), of the lowest and highest line of the staff. The bounding box of the figure is enlarged enough to contain this symbol too.

void AdjustAnnotazione (Vunit Ylinea1, Vunit Ylinea5)

It sets the Annotazione symbol above the note or below, depending on the value of the symbol AboveNota (TRUE = above, FALSE = below). Input parameters Ylinea1 and Ylinea5 are the effectikve heights (in Vunit), of the lowest and highest line of the staff. The bounding box of the figure is enlarged enough to contain this symbol too.

18.1.7 Public methods

Figura ()

It calls the Init method.

~ **Figura ()**

It calls the Free method.

virtual void Init ()

It initialises the figure, by setting Altezza=0 and NumSpaces=1 and by putting NULL all the pointers. It initialises also the rectangles that delimit the figure.

v**oid SetAltezza (int alt)**

It sets to alt the musical height of the figure.

int GetAltezza ()

It returns the musical height of the figure.

void SetNStaff(short staff);

Each *Figura* object has an attribute (*NStaff*) indicating the staff in which it has to be placed. It is initialized to 0 (the upper staff).
This method sets the staff of the figure (0, 1 or 2).

short GetNStaff();

Gets the staff of the figure, for multi staff chords/beams it returns –1.

virtual void GetMinMaxStaff(short& minStaff, short& maxStaff);

Gets the minimum/maximum staff of the figure. This method is redefined in Accordo and GruppoNote to return the minimum/maximum staff of the figures inside the chord or the beam.

Battuta* GetPBattuta();

Gets the measure of the figure.

void SetPBattuta(Battuta* measure);

Sets the measure of the figure.

Figura *GetParentFig();

Gets the parent of the figure, it is NULL for single notes/rest, for a note in a chord is the parent Accordo object and for a note/rest/chord in a beam is the parent GruppoNote object.

void SetVU2UpF(VUnit vu)

It sets VU2UpF to the vu value.

void SetVU2DwnF(VUnit vu)

It sets VU2DwnF to the vu value.

void SetVU2LftF(VUnit vu)

It sets VU2LftF to the vu value.

void SetVU2RgtF(VUnit vu)

It sets VU2RgtF to the vu value.

VUnit Get VU2UpF ()

It returns VU2UpF.

VUnit Get VU2DwnF ()

It returns VU2DwnF.

VUnit Get VU2LftF ()

It returns VU2LftF.

VUnit Get VU2RgtF ()

It returns VU2RgtF.

void SetVU2Up(VUnit vu)

It sets VU2Up to the vu value.

void SetVU2Dwn(VUnit vu)

It sets VU2Dwn to the vu value.

void SetVU2Lft(VUnit vu)

It sets VU2Lft to the vu value.

void SetVU2Rgt(VUnit vu)

It sets VU2Rgt to the vu value.

VUnit GetVU2Up()

It returns VU2Up.

VUnit GetVU2Dwn()

It returns VU2Dwn.

VUnit GetVU2Lft()

It returns VU2Lft.

VUnit GetVU2Rgt()

It returns VU2Rgt.

void SetNumericCode(NumCode nc)

It sets to nc the numerical code of the figure.

NumCode GetNumericCode ()

It returns the numerical code of the figure.

void SetPCorona (Corona *pc)

It sets to pc the pointer to the fermata of the figure.

Corona *GetPCorona ()

It returns the pointer to the fermata.

void SetPOcchiali(Occhiali *po)

It sets to po the pointer to the glasses of the figure.

Occhiali *GetPOcchiali ()

It returns the pointer to the glasses.

v**oid SetPPuntoValore(PuntoValore *ppv)**

It sets to ppv the pointer to the augmentation dot.

PuntoValore *GetPPuntoValore()

It returns the pointer to the augmentation dot.

v**oid SetPTDinamico(TDinamico *ptd)**

It sets to ptd the pointer to the dynamic text of the figure.

TDinamico *GetPTDinamico()

It returns the pointer to the dynamic text of the figure.

v**oid SetPTGenerico(TGenerico *ptg)**

It sets to ptg the pointer to the generic text of the figure.

TGenerico *GetPTGenerico()

It returns the pointer to the generic text of the figure.

v**oid SetPAnnotazione (Annotazione *pa)**

It sets to pa the pointer to the annotation of the figure.

Annotazione *GetPAnnotazione()

It returns the pointer to the annotation of the figure.

v**oid SetPStrumento(Strumento *pst)**

It sets to pst the pointer to the instrument of the figure.

Strumento *GetPStrumento()

It returns the pointer to the instrument of the figure.

CmdResult CmdFiguraPunto(int NPunti)

It sets NPunti augmentation dots to the figure. If it is successful, it returns CMD_OK, otherwise CMD_Fail.

Bool DeleteSym(DrawObject *sym)

It deletes the specified symbol if it is one of those connected to figure. It returns TRUE if the symbol has been deleted.

Bool DeleteStrumento(DrawObject *sym)

Used by DeleteSym, it deals with the particular cases of the instruments. In the Figura class it manages only instruments common to all the figure, concerning the other instruments it has to specialise the method in the different figures.

Virtual void Free()

Deallocates all symbols connected to the figure, by putting NULL the related pointers. It initialises the figure by calling Init.

Bool Hit(const Point& p, SelObj so, DrawObject*& d, SymPath& sp)

It returns TRUE if the figure has been struck and a selection of the FIGURA_SEL or of the ANY_SEL kind has been set, or when one of the symbols linked to the figure has been struck and a selection of the ANY_SEL kind has been set.

Bool Hit(const Rectangle&, SelObj, SymPath&, SymPath&)

See the paragraph 1.18.2.

Bool SymbolicHit(SymPath&, SelObj, DrawObject*&)

Decodes the path symPath (see the paragraph 1.18.2).

virtual Bool FigPtr2SymPath(SymPath&, Figura*)

It reconstructs the path of a figure starting from the pointer.

CmdResult SymCommand(SymCmd)

See the general documentation of the method SymCommand in the section 1.19.

CmdResult SymCommand(SymCmd&, Battuta *pBattuta=NULL)

See the general documentation of the method SymCommand in the section 1.19.

void SetSpG (UL spazi)

Sets parameter SpG.

void SetSpOr (UL spazi)

Sets parameter SpOr.

void SetSpDirUt (UL spazi)

Sets parameter SpDirUt.

void SetSpOrcUt (UL spazi)

Sets parameter SpOrcUt.

UL GetSpG (UL spazi)

Returns SpG.

UL GetSpOr (UL spazi)

Returns SpOr.

UL GetSpDirUt (UL spazi)

Returns SpDirUt.

UL GetSpOrcUt (UL spazi)

Returns SpOrcUt.

UL GetSpDLine (UL spazi)

Returns SpDLine.

UL GetSpOLine (UL spazi)

Returns SpOLine.

void SetSpDLine (UL spazi)

Sets parameter SpDLine.

void SetSpDirUt (UL spazi)

Sets parameter SpOLine.

virtual UL GetSp2NextFig ()

Returns space in UL until the next figure of voice:
1) for director SpG+SpDirUt
2) for musician SpOr+SpOrcUt

void SetNotina (Bool b)

Sets parameter to draw a natural dimension figure or a grace figure (TRUE = grace note or pause).

Bool GetNotina ()

Returns parameter Notina.

void SetNotina (Bool b)

Sets parameter to draw a natural dimension figure or a grace figure (TRUE = grace note or pause).

Bool GetNotina ()

Returns parameter Notina.

virtual void Suona (MoodsMidiWrite*)

Method to generate Midi file from symbolic.

void SetTerzina (LegatQuadra *pleg)

Sets parameter terz = pleg if not NULL.

void DeleteTerzina (LegatQuadra *pleg)

Sets parameter terz = NULL if terz = pleg.

LegatQuadra *GetTerzina (void)

Returns terz.

virtual Bool IsANote ()

Returns FALSE.

virtual Bool IsARest ()

Returns FALSE.

virtual float GetDurata ()

Returns the effective duration of the note (take in account also irregular groups, augmentation dots etc.).

float GetDurata (float den)

Returns the effective duration of the note (take in account also irregular groups, augmentation dots

void SetDurata (float dur)

Set parameter durata

void SetNumSpaces (int ns)

Set parameter NumSpaces

int GetNumSpaces ()

Returns parameter NumSpaces

void SetPos (DrawObject *d, const Point& p)

Set the position of the figure and calls Adjust().

virtual void SetPosWA (DrawObject *d, const Point& p)

Set the position of the figure and without calling Adjust()

virtual void Adjust (Bool primaleg, Battuta* pBattuta, int layer)

Sets in the right position all the symbols related to the figure.

void Draw ()

Draws the figure.

virtual void print ()

Print method.

virtual DrawObject* GetAggiunta (SymCmd* MyCmd)

Related to mouse commands, returns pointer to symbol.

void SetupSimboli ()

deprecated.

virtual char* Describe (Context)

Returns a description in MusicTex of the figure.

char* GiustifDescribe ()

Returns a description of the justification.

void AddIntRef(IntRefType,IntEsteso*)

Adds a reference of a horiz. symbol to the figure.

void DelIntRefs()

Removes all the references to the Horizontal symbols present in the figure

int GetNIntRefs()

It returns the number of references present.

IntRef* GetIntRef(int n)

It returns the pointer to the n-th IntRef element of the figure starting from 0.

Warning: use SetupIntRefs() methods of Partitura and of Spartito to properly initialize this data

Status_ID GetStatus()

It returns the status of the figure, it can be:

NORMAL – visible and with visual duration
GRACED – visible and without visual duration
HIDDEN – invisible and with visual duration
GHOSTED – invisible and without visual duration

virtual void SetStatus(Status_ID stat)

It sets the status of the figure.

18.2 Class Nota

18.2.1 Description

The abstract class Nota collects the attributes and the methods valid for the management of simple notes, chords and beams.

The meaning of **note** is *graphic symbol that refers to a musical sound and to the related duration value.* The note, *which is placed on the musical line (staff), provided with clef and eventually with key signature, or outside it, above or below, with the addition of the so called leger lines, and, in case, with the integration of accidentals (sharp, double sharp, flat, double flat or natural), makes clear a precise height.*

The height of notes is implemented with the Altezza attribute (of the int kind), inherited by Figura. To each position of the staff and of the leger lines a number is assigned (Altezza) that has value 0 for the note placed on the first line (the inferior line of the staff), 1 for the note placed on the successive space, and so on. For example, if the staff has a tremble clef the Re note, that is placed immediately under the staff has Altezza –1, the following Mi has Altezza 0, the Sol on the second line has Altezza 2. The most spread method for indicating the height of a note is different from that above described: it uses the name of the note and the number of the octave which the note belongs to (e.g. Fa sharp 3). This method present the problem that in order to represent on the staff a note so described it is necessary to know the clef. On the other hand our implementation permits identifying immediately the position of the note on the staff without knowing the clef.

For the ornaments called **appoggiatura** and **acciaccatura (**we remember that the musical ornaments are not represented only by the classes derived from the Abbellimento class) and for the solo rhythm small notes are typically used. At the beginning of the project it had been decided to implement the small notes with a class of their own but later it has been decided to obtain the small notes through an attribute (Bool Notina) of the Nota class. In this way, by specifying Notina=TRUE, the note is designed with a smaller dimension (this is also valid for the beams but not for the chords) and it is also possible to draw small notes anywhere on the staff.

In order to obtain an appoggiatura we need to set the note with SetNotina (TRUE). An acciaccatura composed by only one note has the aspect of a little 8th note with an oblique dash, therefore it is obtained by creating a 8th note (with new NCroma) and performing SetAcciaccatura (TRUE). For the acciaccatura made of small groups of notes see the documentation of the GruppoNote class.

The symbols connected with Nota through a relation IS_REFERRED_BY are:

AlterazioneComposta (**Composta** alteration)
Abbellimento (ornament) It is a symbol that has to be designed above or below the note.
Diteggiato (fingering) It shows which finger has to execute the note. It has to be designed above or below the note.
Espressione (expression) It shows the expression to be given to the single note. It has to be designed above or below the note.
Sordina (mute) For the string instruments, it shows a technique of execution of the note.
Suddivisione (subdivision) It shows the subdivision of the value of the note.

To these symbols the seven (augmentation dot, fermata, glasses, dynamic text, generic text, annotation and instrument) that Nota inherits from Figura are to be added.

18.2.2 Father Class

Figura

18.2.3 Children Classes

NBreve, NSemibreve, NMinima, NSemiminima, NCroma, NSemicroma, NBiscroma, NSemibiscroma, NFusa, Accordo.

18.2.4 Protected Attributes

VUnit LGambo

Length of the stem of a note. *Stem* is the vertical segment that starts from the ellipsis of the note and goes upwards or downwards. Only the whole note has no stem: all the other kind of notes have one. Nevertheless in chords with stem only a note has a stem and the others are deprived of it. If LGambo>0, the stem is directed upwards; if LGambo<0, the stem is directed downwards. If LGambo=0, the note has no stem.

Bool Tie

Added by Suona method.

Bool Coda

It indicates if the note has to be designed with the coda. It is TRUE for the note for which the coda has to be designed. It is initialised at FALSE.

Abbellimento *PAbbellimento

Pointer to the ornament to the note.

Diteggiato *PDiteggiato

Pointer to the fingering to the note.

EspressComposta *PEspressComposta

Pointer to the composite expression sign (espressione composta).

Sordina *PSordina

Pointer to the mute sign.

Suddivisione *PSuddivisione

Pointer to the subdivision sign.

18.2.5 Protected Methods

void AdjustEspressComposta(Vunit Ylinea1, Ylinea5, Bool onspace, Bool onstaff, Point point, Espressione *Pespressione=NULL)

Positions the espressione composta above or below the note. Ylinea1 and Ylinea5 are the effective heights (in VUnit) of the lowest and highest line of the staff. Adapt the bounding box of the figure plus its symbols as to contain all the alterations of espressione composta.

void AdjustArmonici(Vunit Ylinea1, Ylinea5, Point point)

Positions the armonici above or below the note. Ylinea1 and Ylinea5 are the effective heights (in VUnit) of the lowest and highest line of the staff. Adapt the bounding box of the figure plus its symbols as to contain all the alterations of espressione composta.

void AdjustParteArco(Vunit Ylinea1, Ylinea5, Point point)

Positions the ParteArco above or below the note. Ylinea1 and Ylinea5 are the effective heights (in VUnit) of the lowest and highest line of the staff. Adapt the bounding box of the figure plus its symbols as to contain all the alterations of espressione composta.

void AdjustPosizArco(Vunit Ylinea1, Ylinea5, Point point)

Positions the PosizArco above or below the note. Ylinea1 and Ylinea5 are the effective heights (in VUnit) of the lowest and highest line of the staff. Adapt the bounding box of the figure plus its symbols as to contain all the alterations of espressione composta.

void AdjustDirezArco(Vunit Ylinea1, Ylinea5, Point point)

Positions the DirezArco above or below the note. Ylinea1 and Ylinea5 are the effective heights (in VUnit) of the lowest and highest line of the staff. Adapt the bounding box of the figure plus its symbols as to contain all the alterations of espressione composta.

void AdjustCorda(Vunit Ylinea1, Ylinea5, Point point)

Positions the Corda above or below the note. Ylinea1 and Ylinea5 are the effective heights (in VUnit) of the lowest and highest line of the staff. Adapt the bounding box of the figure plus its symbols as to contain all the alterations of espressione composta.

18.2.6 Private methods

CmdResult CmdNotaDiesis()

It assigns to the note the accidental sharp. If it is successful it returns CMD_OK, otherwise CMD_FAIL.

CmdResult CmdNotaDiesis1Q()

It assigns to the note the accidental sharp1Q. If it is successful it returns CMD_OK, otherwise CMD_FAIL.

CmdResult CmdNotaDiesis3Q()

It assigns to the note the accidental sharp3Q. If it is successful it returns CMD_OK, otherwise CMD_FAIL.

CmdResult CmdNotaBemolle()

It assigns to the note the accidental flat. If it is successful it returns CMD_OK, otherwise CMD_FAIL.

CmdResult CmdNotaBemolle1Q()

It assigns to the note the accidental flat1Q. If it is successful it returns CMD_OK, otherwise CMD_FAIL.

CmdResult CmdNotaBemolle3Q()

It assigns to the note the accidental flat3Q. If it is successful it returns CMD_OK, otherwise CMD_FAIL.

CmdResult CmdNotaBequadro()

It assigns to the note the accidental natural. If it is successful it returns CMD_OK, otherwise CMD_FAIL.

CmdResult CmdNota2Diesis()

It assigns to the note the accidental double sharp. If it is successful it returns CMD_OK, otherwise CMD_FAIL.

CmdResult CmdNota2Bemolle()

It assigns to the note the accidental double flat. If it is successful it returns CMD_OK, otherwise CMD_FAIL.

CmdResult CmdNotaTrillo(TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta, int NOnde)

It assigns to the note the trill sign. If it is successful it returns CMD_OK, otherwise CMD_FAIL. In order to represent the extension of the trill effect, a wave, of different lengths, follows the *tr* symbol. The entry parameter NOnde can have the following values: 0 (no wave), 1,2,3,4 (wave with the greatest length).

CmdResult CmdNotaGruppettoSup1(TipoInserimento, Nota* , Battuta*)

It assigns to the note the turn sign. If it is successful it returns CMD_OK, otherwise CMD_FAIL.

CmdResult CmdNotaGruppettoInf1(TipoInserimento, Nota* , Battuta*)

It assigns to the note the turn back sign. If it is successful it returns CMD_OK, otherwise CMD_FAIL.

CmdResult CmdNotaMordenteSup(TipoInserimento, Nota* , Battuta*)

It assigns to the note the superior mordent sign. If it is successful it returns CMD_OK, otherwise CMD_FAIL.

CmdResult CmdNoteMordente2Sup(TipoInserimento, Nota* , Battuta*)

It assigns to the note the superior double mordent sign. If it is successful it returns CMD_OK, otherwise CMD_FAIL. To be noticed: *superior mordent* is only the name of a kind of mordent and it does not refer to the position of the mordent with respect to the note.

CmdResult CmdNoteMordente2Inf(TipoInserimento, Nota* , Battuta*)

It assigns to the note the inferior double mordent sign. If it is successful it returns CMD_OK, otherwise CMD_FAIL. To be noticed: *inferior mordent* is only the name of a kind of mordent and it does not refer to the position of the mordent with respect to the note.

CmdResult CmdNotaTenuto(TipoInserimento, Nota* , Battuta*)

It assigns to the note tenuto sign. If it is successful it returns CMD_OK, otherwise CMD_FAIL.

CmdResult CmdNotaPuntoAll(TipoInserimento, Nota* , Battuta*)

It assigns to the note the punto allungato sign. If it is successful it returns CMD_OK, otherwise CMD_FAIL.

CmdResult CmdNoteMordenteInf(TipoInserimento, Nota* , Battuta*)

It assigns to the note the inferior mordent sign. If it is successful it returns CMD_OK, otherwise CMD_FAIL. It is necessary to specify AboveN=TRUE for designing this sign above the figure, or FALSE for designing it below. To be noticed: *inferior mordent* is only the name of a kind of mordent and it does not refer to the position of the mordent with respect to the note.

CmdResult CmdNotaDito(TipoInserimento, Nota* , Battuta*, int NDito)

It assigns the fingering to the note. If it is successful, it returns CMD_OK, otherwise CMD_FAIL.

CmdResult CmdNotaCorda(SymCmd SCmd TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It recalls the SymCommand of the instrument associated to the note. If the pointer to the instrument is NULL a new Arco (Bow) is assigned to it.

CmdResult CmdNotaStrArco(SymCmd SCmd TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the bow sign. It entrusts to bow the execution of the command that consists in visualising the “Arco” string above or below the note.

CmdResult CmdNotaPizzicato(TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the pizzicato sign. If it is successful it returns CMD_OK otherwise CMD_FAIL.

CmdResult CmdNotaSforzato(TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the sforzato sign. If it is successful it returns CMD_OK otherwise CMD_FAIL.

CmdResult CmdNotaAccento(TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the accent sign. If it is successful it returns CMD_OK otherwise CMD_FAIL.

CmdResult CmdNotaMartellato(TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the martellato sign. If it is successful it returns CMD_OK otherwise CMD_FAIL.

CmdResult CmdNotaMartDolce(TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the martellato dolce sign. If it is successful it returns CMD_OK otherwise CMD_FAIL.

CmdResult CmdNotaStaccato(TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the sign of portato staccato (loure). If it is successful it returns CMD_OK otherwise CMD_FAIL.

CmdResult CmdNotaEspGenerica(int Idsym, TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the sign of espressione generica. If it is successful it returns CMD_OK otherwise CMD_FAIL.

CmdResult CmdNotaViaSordinaOttoni(TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the sign of without mute for Ottoni. If it is successful it returns CMD_OK otherwise CMD_FAIL.

CmdResult CmdNotaViaSordinaArchi(TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the sign of without mute for Archi. If it is successful it returns CMD_OK otherwise CMD_FAIL.

CmdResult CmdNotaViaSordinaTesto(TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the sign of without mute textual. If it is successful it returns CMD_OK otherwise CMD_FAIL.

CmdResult CmdNotaConSordinaOttoni(TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the sign of with mute for Ottoni. If it is successful it returns CMD_OK otherwise CMD_FAIL.

CmdResult CmdNotaConSordinaArchi(TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the sign of with mute for Archi. If it is successful it returns CMD_OK otherwise CMD_FAIL.

CmdResult CmdNotaConSordinaTesto(TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the sign of with mute textual. If it is successful it returns CMD_OK otherwise CMD_FAIL.

CmdResult CmdNotaArmoniciOttoni(TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the sign Armonici for Ottoni. If it is successful it returns CMD_OK otherwise CMD_FAIL.

CmdResult CmdNotaArmoniciArchi(TipoInserimento tpIns, Nota* pNota, Battuta* pBattuta)

It assigns to the note the sign Armonici for Archi. If it is successful it returns CMD_OK otherwise CMD_FAIL.

CmdResult CmdNotaSuddivisione(short nt)

It assigns to the note the subdivision sign with a number of bar lines equal to nt. If it is successful it returns CMD_OK otherwise CMD_FAIL. Necessary to specify the number of lines dividing the note.

CmdResult CmdNotaTremolo(short nb)

It assigns to the note the tremolo sign with a number of bar lines equal to nb. The tremolo involves the note in question and the successive figure. If this latter is not a note the method fails. If it is successful it returns CMD_OK otherwise CMD_FAIL. Necessary to specify number of bars composing the object.

CmdResult CmdNotaGlissato()

It assigns to the note the glissato sign If it is successful it returns CMD_OK otherwise CMD_FAIL.

void AdjustAlterazComposta()

Sets the alterations of alterazione composta on the left of the note. Adapt the bounding box of the figure plus its symbols as to contain all the alterations of alterazione composta.

void AdjustAbbellimento(Vunit Ylinea1, Ylinea5, Point)

Positions the abbellimento above or below the note. Ylinea1 and Ylinea5 are the effective heights (in VUnit) of the lowest and highest line of the staff. Adapt the bounding box of the figure plus its symbols as to contain all the alterations of alterazione composta.

void AdjustDiteggiato(Vunit Ylinea1, Ylinea5, Point)

Positions the diteggiato above or below the note. Ylinea1 and Ylinea5 are the effective heights (in VUnit) of the lowest and highest line of the staff. Adapt the bounding box of the figure plus its symbols as to contain all the alterations of alterazione composta.

void AdjustSordina(Vunit Ylinea1, Ylinea5, Point)

Positions the sordina above or below the note. Ylinea1 and Ylinea5 are the effective heights (in VUnit) of the lowest and highest line of the staff. Adapt the bounding box of the figure plus its symbols as to contain all the alterations of alterazione composta.

void AdjustStrumento(Vunit Ylinea1, Ylinea5, Point)

Positions the strumento above or below the note. Ylinea1 and Ylinea5 are the effective heights (in VUnit) of the lowest and highest line of the staff. Adapt the bounding box of the figure plus its symbols as to contain all the alterations of alterazione composta.

void AdjustSuddivisione()

Positions the suddivisione on the note.

18.2.7 Public methods

Nota()

It calls the Init method.

Nota(double)

It calls the Init method.

~ Nota()

It calls the Free method.

Bool IsANote()

Returns TRUE.

float GetDurata()

Returns 0 if grace note otherwise calls the method of Figura.

void AdjustStrumento(Bool primaleg, Battuta *pBattuta, int layer)

Positions at the right place all the sybols of the notet. Sets VU2Up, VU2Dwn, VU2Lft, VU2Rgt on the base of the dimension of the note and that of the symbols related to the note. If a croma, or a semicroma etc. has Coda=FALSE, it means that it belongs to a chord or to a group. The chords have their own Adjust method, while groups refer to this one. Thus to the notes of this kind is raised (with INCRYNOTESENZACODA) the rectangle bounding them to adequate to the bars of the groups.

void SetPos(DrawObject*, const Point&)

Sets the position of the object.

void Draw()

Draw the note and the symbols related to it.

void print()

Printing method.

v**irtual void DrawForAccordo()**

Draws the note. Around the note draws only the alteration, the augmentation dot and the diteggiatura. Used to draw the notes in the chords.

v**irtual void printForAccordo()**

Draws the note and alterations.

CmdResult CmdNotaNota2Notina()

Tranforms a note in a notina setting Notina=TRUE and gives back CMD_OK. Makes the stem little, with the same direction. If a note is already a grace note or belongs to a group of notes, the method does nothing and gives back CMD_FAIL.

CmdResult CmdNotaNotina2Nota(TipoInserimento tpIns, Nota *pNota, Battuta* pBattuta, int layer)

Tranforms a notina in a note setting Notina=FALSE and gives back CMD_OK. Makes the stem big, with the same direction. If a note is already a big note or belongs to a group of notes, the method does nothing and gives back CMD_FAIL.

v**irtual DrawObject* GetAggiunta(SymCmd* MyCmd)**

Returns the type of symbol related to the note selected,

v**irtual void AdjustF()**

Implemented in the child classes.

v**irtual void Adjust0()**

Deprecated.

DrawObject* UltimoDrwPriority(Bool sopra)

Returns last drawn symbol.

void InitRectF()

It initialises the rectangle that surrounds the note for the note’s default values (or for the small note if Notina=TRUE).

void Init()

It initialises the note by setting Altezza=0, NumSpaces=1, Notina=FALSE, Coda=FALSE, LGambo=0, and putting NULL all the pointers. It initialises also the rectangles that delimit the note.

virtual void SetLGambo (VUnit lg)
It sets the stem of the note at the lg length (positive, negative or null).

virtual void SetGamboUp ()
It sets the stem above the note with default length (half default length if the note is a small one).

void SetNoGambo ()
It sets the note without stem, LGambo=0.

virtual void SetGamboDwn ()
It sets the stem below the note with default length (half default length if the note is a small one).

VUnit GetLGambo ()
It returns the length of the stem of the note.

virtual void SetCoda(Bool coda)
It sets Coda=coda. Since some kind of note refuses the coda, this method is set virtual and redefined in the note classes that do not admit Coda=TRUE.

Bool GetCoda ()
It returns Coda.

void SetPAlterazioneComposta (AlterazioneComposta *palt)
It sets PAlterazioneComposta at palt.

AlterazioneComposta *GetPAlterazioneComposta()
It returns PAlterazioneComposta.

void SetPAbbellimento(Abbellimento *pabb)
It sets PAbbellimento at pabb.

Abbellimento *GetPAbbellimento()
It returns PAbbellimento.

void SetPDiteggiato (Diteggiato *pd)
It sets PDiteggiato at pd.

Diteggiato *GetPDiteggiato()
It returns PDiteggiato.

void SetPEspressComposta (EspressComposta *pe)
It sets PEspressComposta at pe.

EspressComposta *GetPEspressComposta()
It returns PEspressComposta.

void SetPSordina (Sordina *ps)
It sets PSordina at ps.

Sordina *GetPSordina()
It returns PSordina.

void SetPSuddivisione (Suddivisione *ps)
It sets PSuddivisione at ps.

Suddivisione *GetPSuddivisione()
It returns PSuddivisione.

void MoveX(VUnit dx)
It moves the note with an horizontal movement of dx VUnit.

Bool DeleteSym(DrawObject *sym)
It deletes the specified symbol if it is among those connected to note. It returns TRUE if the symbol is deleted.

Bool DeleteStrumento(DrawObject *Sym)
Used by DeleteSym, it deals with the particular cases of the instruments.

void Free()
Deallocates all the symbols connected to note, putting NULL all the related pointers. It initialises the note by calling Init.

Bool Hit(const Point& p, SelObj, selobj, DrawObject*& drwObjSel, SymPath& symPath)
See the general documentation of the Hit method in section 1.18.2.

Bool SymbolicHit(SymPath&, SelObj, DrawObject*&)
See paragraph 1.18.2.

CmdResult SymCommand(SymCmd)
See general documentation of the SymCommand method in section 1.19.

Char *Describe(Context)

See section ….

18.3 Class Accordo

18.3.1 Description

The class Accordo permits representing the *simultaneous combination of several sounds.* From the graphic point of view a chord consists of several notes of the same duration disposed vertically and attached at the same stem (only the whole notes' chords have no stem). According to our analysis we intend for "notes of the same duration" instances of a single class son of Nota (with the exception of Accordo), as for example NCroma. Since a chord has an appearance and a meaning similar to those of the single note, it has been decided to make this class derive from Nota. In this way most of the attributes and methods are inherited even if is necessary to redefine some methods. At the same time Accordo contains several notes and therefore it has been necessary let this class derive also from ListaFigure.

We have also decided that the signs of expression, ornament, violin, fingering, fermata, dynamics and agogic are unique for the chord. This implies that each note that is inserted in the chord looses all these symbols. Obviously it does not loose the accidentals, the augmentation dot and the fingering that are peculiar of the single note and thus are to be represented even if it is inserted into a chord.

Since the use of this class is very different from that of the other classes of the module, we present an example of use similar to that introduced for the Figura class:

1. Accordo
2. AddNota
3. AddNota
4. AddNota
5. SetLGambo
6. AddNota
7. SetCoda
8. AddNota
9. AddNota
10. SetPuntiValore
11. SetPos
12. Draw
13. SetPViolino
14. SetPCorona
15. SetPos
16. Draw
17. ~ Accordo

After making modifications on a chord it is mandatory to perform Adjust again before designing it, and it is possible to execute it also through the method SetPos that recalls Adjust.

Accordo inherits, among others, the attributes of Altezza and AbsPos. From the musical point of view talking about the height of a chord is totally wrong since the chord is composed by notes with different heights. This attribute and AbsPos is useful to uniform the management of this class and that of the other musical figures. Thus as **Altezza** of Accordo is to be considered the height of the upper note, if the stem is directed upwards, and that of the inferior note, if the stem is directed downwards. This convention is useful in order to let GruppoNote treat the chord like a simple note. From the height of the chord also its AbsPos is fixed (it is a task of the SetPos method).

Accordo inherits from Figura the NumericCode attribute; furthermore, as a ListaFigure, it has to assign the codes to the possessed notes.

18.3.2 Father Class

Nota, ListaFigure

18.3.3 Protected Attributes

ClassID NoteID
Identifier of the type of note that forms the chord.
Bool DiteggiatoUp
It is TRUE if the fingerings of the single notes are to be designed above the chord, FALSE if they are to be designed below.
Arpeggio *PArp
Pointer to the arpeggio associated to the chord.

18.3.4 Private Methods

CmdResult CmdAccordoArpeggio()
It assigns the arpeggio sign to the chord. If it is successful, it returns CMD_OK, otherwise CMD_FAIL.
CmdResult CmdAccordoNotina2Nota()
It sets the parameter for the grace notes.
CmdResult CmdAccordoNota2Notina()
It sets the parameter for the grace notes.
CmdResult CmdAccordoNota2Acciaccatura()
It sets the parameter for the grace notes.

void PlaceNoteGamboUp (Vunit Ylinea1)
Positions the notes of the chord in the case that the stem is upward. The chord is examined from the bottom to the top and the notes are positioned at the left of the stem. If two notes are paretially overlapped, the highest is moved on the right of the stem. Ylinea1 is the effective height of the lowest line of the staff.
void PlaceNoteGamboDown (Vunit Ylinea1)
Positions the notes of the chord in the case that the stems is downward. The chord is examined from the top to the bottom and the notes are positioned at the left of the stem. If two notes are partially overlapped, the lowest is moved on the left of the stem. Ylinea1 is the effective height of the lowest line of the staff.
void AdjustAlterazPValore ()
Sets the alterations and the augmentation dots of the notes composing the chord.
void AdjustEspressComposta (Vunit Ylinea1, Vunit Ylinea5, Bool primaFase)
Sets the the espressione composta. Ylinea1 and Ylinea5 are the effective heights of the lowest line and the highest line of the staff.
void AdjustAbbellimento (Vunit Ylinea1, Vunit Ylinea5)
Sets the the abbellimento. Ylinea1 and Ylinea5 are the effective heights of the lowest line and the highest line of the staff.
void AdjustStrumento (Vunit Ylinea1, Vunit Ylinea5)
Sets the the strumento. Ylinea1 and Ylinea5 are the effective heights of the lowest line and the highest line of the staff.
void AdjustDiteggiato (Vunit Ylinea1, Vunit Ylinea5)
Sets the the diteggiato. Ylinea1 and Ylinea5 are the effective heights of the lowest line and the highest line of the staff.
void AdjustCorona (Vunit Ylinea1, Vunit Ylinea5)
Sets the the corona. Ylinea1 and Ylinea5 are the effective heights of the lowest line and the highest line of the staff.
void AdjustOcchiali (Vunit Ylinea1, Vunit Ylinea5)
Sets the the occhiali. Ylinea1 and Ylinea5 are the effective heights of the lowest line and the highest line of the staff.
void AdjustTDinamico (Vunit Ylinea1, Vunit Ylinea5)
Sets the the TDinamico. Ylinea1 and Ylinea5 are the effective heights of the lowest line and the highest line of the staff.
void AdjustTGenerico (Vunit Ylinea1, Vunit Ylinea5)
Sets the the TGenerico. Ylinea1 and Ylinea5 are the effective heights of the lowest line and the highest line of the staff.
void AdjustSordina (Vunit Ylinea1, Vunit Ylinea5)

Sets the the abbellimento. Ylinea1 and Ylinea5 are the effective heights of the lowest line and the highest line of the staff.

void AdjustAnnotazione (Vunit Ylinea1, Vunit Ylinea5)

Sets the the annotazione. Ylinea1 and Ylinea5 are the effective heights of the lowest line and the highest line of the staff.

void AdjustArpeggio (Vunit Ylinea1, Vunit Ylinea5)

Sets the the abbellimento. Ylinea1 and Ylinea5 are the effective heights of the lowest line and the highest line of the staff.

18.3.5 Public Methods

Accordo()

It performs Init.

~ Accordo ()

It performs Free.

Init()

It initialises the chord as empty list. It sets Altezza=0, NumSpaces=1, Notina=FALSE, Coda=FALSE, LGambo=0, DiteggiaotUp=TRUE and puts NULL all the pointers. It initialises also the rectangles that delimit the note.

ClassID GetNoteID()

It returns the identifier of the type of note that forms the chord.

void SetDiteggiatoUp(Bool dup)

It sets DiteggiatoUp at dup.

Bool GetDiteggiatoUp()

It returns DiteggiaotUp.

void SetPArpeggio(Arpeggio *pa)

It sets Arpeggio at pa.

Arpeggio GetPArpeggio()

It returns PArpeggio.

Bool AddNota(Nota *pn)

It checks the note pointed by pn. If it pass the control, it is added to chord and the method returns TRUE, otherwise it is not added and the method returns FALSE. The cases in which the note *pn is rejected are:

- The note has not the same value of those present in the chord.
- In the chord exists a note with the same height of that we want to insert.
- The note is a small note.

The first note that is inserted in the chord fixes its NoteID. It is not necessary to present the notes aligned: Adjust will correctly align them on the basis of AbsPos.x. This method inserts the notes so as to maintain the list ordered according to the height of notes. Since the list is a double one, it is easy to scan the chord from the top to the bottom and from the bottom to the top. The notes added to the chord are deprived of all the symbols but the augmentation dot, the accidentals and the fingering.

void SetLGambo(VUnit lg)

It sets the length of the stem. If the chord is composed by whole notes it does not act. It is mandatory to assign a not null length to the stem of the chord.

void SetGamboUp()

It sets the stem over the note with default length. If the chord is composed by whole notes it does not act.

void SetGamboDwn()

It sets the stem under the note with default length. If the chord is composed by whole notes it does not act.

void SetCoda(Bool Coda)

It sets the coda of the chord. If the chord is composed by whole notes, half notes or quarter notes Coda is always FALSE.

void SetPuntiValore(int np)

It sets np augmentation dots to all the notes that compose the chord. Np must be included among 1 and 3, otherwise the method has no effect. If we want to add notes to the chord, it is necessary to execute this method another time after the addition.

void Free()

Deletes all the symbols and the related pointers, it deletes the list and deallocates the notes. Finally it initialises the chord.

Bool Hit(const Point& p, SelObj selObj, DrawObject*& drwObjSel, SymPath& symPath)

See the general documentation of the Hit method in section 1.18.2.

Bool SymbolicHit(SymPath&, SelObj, DrawObject*&)

See the general documentation of the Hit method in section 1.18.2.

Bool DeleteSym(DrawObject *sym)

It deletes the specified symbol if it is among those linked to the chord. It returns TRUE if the symbol has been deleted.

Bool FigPtr2SymPath (SymPath&, Figura*, Bool inGruppo=FALSE)

It rebuilds the PATH of a figure starting from the pointer.

void Select()

It designs on the screen the rectangle that surrounds the chord provided with symbols, with colour XOR_SELECTED_COLOR.

void Deselect()

It deletes from the screen the rectangle designed by Select.

void SetPos(DrawObject *d, const Point &p)

It sets the position of the chord, not depending from Altezza, and calls Adjust(). Lgambo must be already setted . Set also the Altezza of the chord, which is the height of the superior note if the stem is up or the height of the inferior note if the stem is down [this convention is used so the GruppoNote treats the chord as a simple note.]. From the height of the chord is determined also its AbsPos (the alignment will be based on this). If it is necessary to add notes to the chord, after adding is a must to call this method again.

void SetPosWA(DrawObject *d, const Point &p)

Equal to SetPos, but does not call Adjust().

void AdjustF()

It adjust the horizontal position of the notes of the chord depending on the direction of the stem and setup for the drawing of the stem. This method has to be called after setting the stem and the eventual coda of the chord. If LGambo=0 this method acts as if Lgambo was > 0.

void Adjust (Bool primaleg, Battuta *pBattuta, int layer)

Setup the notes and set the bounding box around the chord. Setup also all the symbols related to the chord.

void Draw()

Call draw for all the symbols related to the chord.

void print()

Printing method.

int GetDiffUpperLower()

Returns a different value depending on the difference between the distance of the highest note of a chord from the third line of the staff and the distance of the lowest note of the chord from same line.

int GetDiffNUpNDown()

Returns a different value depending on the difference between the number of notes of a chord up the third line of the staff and the number of notes of the same chord laying below the third line.

int GetNumUp()

Returns the number of notes which height is major than 4.

int GetNumDown()

Returns the number of notes which height is minor than 4.

Nota* GetLastNote(void)

Returns the last note of the chord.

Nota* GetFirstNote(void)

Returns the first note of the chord.

virtual DrawObject* GetAggiunta (SymCmd* MyCmd)

Gets the pointer to an added symbol.

CmdResult SynCommand(SymCmd SCmd)

See the general documentation of the SymCommand method in section 1.19.

char *Describe(Context)

See section……

int GetAltezzaMedia()

Returns the medium height between the highest and lowest note of the chord.

Note

It is absolutely not difficult to extend the chords also to the small notes: it is only necessary to change some methods, as already made for Nota.

18.4 Class GruppoNote

18.4.1 Description

The word **beam** refers to a sequence of notes that have no coda but are connected through parallel bar lines. The number of bar lines corresponds to the number of coda that the notes had had if they would not be grouped (e.g. a 16th note has two coda, thus a group of 16th notes has two bar lines). This implies that the notes that can compose beams are: 8th notes, 16th notes, 32nd notes, 64th notes and 128th notes.

Since a beam contains several notes, it is practically mandatory making the class GruppoNote derive from ListaFigure that has the task to represent groups. In addition, since the Battuta class is also a list of figures, we thought of making GruppoNote derive from Figura. In this way the beam can be placed and designed with the same commands used for the single figures. Furthermore, this class inherits from Figura the fundamental attributes and redefines its methods. Since beams are sequences of notes, it could seem better make GruppoNote derive from Nota rather than Figura, but it would have overloaded this class with useless attributes.
There are also mixed beams, composed by notes with different values. In this case the number of bar lines changes, inside the beam, from note to note. It is rather unusual to find them in the scores, and, moreover, there are some beams where appear notes and rests. LIOO does not permit inserting rests in a beam.

Since in the composition of measures can appear spaces (useful in particular to avoid overlapping among symbols), it has been considered useful to introduce spaces also inside the beams.
It is possible to obtain beams of small notes (it is enough to group notes that have Notina=TRUE), this is enough in order to obtain acciaccature with several notes. In a beam can appear at the same time single notes and chords (as already said chords of small notes are not possible), besides, of course, spaces. The total number of figures in a beam (notes, chords and spaces) is contained in the attribute NumSpaces, inherited from Figura, that shows how many units of space are occupied by the beam.
For the beam management it is necessary to find out the highest note of the beam, if the stem is over the note, or the lowest if the stem is under it. This note is called *the most meaningful note* of the beam. It could be also a chord, it cannot be, obviously, a space.

As already done for the class Accordo, we present a brief example of the use of methods of GruppoNote:

1. GruppoNote
2. AddFigura
3. AddFigura
4. AddFigura
5. AddFigura
6. SetGamboDwn
7. AddFigura
8. AddFigura
9. SetPos
10. Draw

11. ExtractFigura
12. ExtractFigura
13. AddFigura
14. SetPos
15. Draw
16. Select
17. Deselect
18. ~ GruppoNote

To be noticed SetNotina has not been used, in effect the dimension of the beam depend on the dimension of the notes that are inserted. Also to be noticed is that SetGamboDwn has been inserted only after having inserted some notes (this method works according to the dimension of the beam).

The operation of notes' insertion is performed from left to right and it is possible to start and end the insertion with a space. Anyway it is better not to insert spaces at the extremities of beams because they are not distinguishable from the spaces present in the measure.
GruppoNote inherits from Figura the attribute NumericCode; moreover, as a ListaFigure, it has to assign the codes to the notes that it possesses.

18.4.2 Father Class

Figura, ListaFigure

18.4.3 Protected Attributes

int NumNote
It is the number of notes and chords that compose the beam. This number does not take into account the spaces that are in the group.

Nota *FirstNota, *LastNota
Pointer to the first and to the last note (or chord) of the beam.

Bool Notina
It is TRUE if the beam is composed by small notes, otherwise FALSE. It is initialised at FALSE.

VUnit SpaceWidth
The width of the space unit of the staff.

VUnit LGambo
It is the length of the stem of the most meaningful note of the beam, it is calculated from the physical height of the note to the height of the first bar line. The *first bar line* is the lowest one if the beam has the stem directed upwards or the highest one if the beam has the stem directed downwards.

ClassID NoteID
Identifier of the kind of note that composes the beam.

short NBarre
Number of bar lines of the beam.

Point BPoint1,BPoint2
The extreme points of the first bar line of the beam. If BPoint1=(0,0), the bar lines are not designed.

float BSlope
Slope of the bar lines of the beam, expressed as angular coefficient.

TipoTratto Tratto[3]
Array that contains parallel parts of bar (the most external part does not appear because the most external bar is not broken).

18.4.4 Public Methods

GruppoNote()
It performs Init.

~**GruppoNote()**
Destroyer.

UL GetSp2NextFig()

Returns total spacing.

void Init()

It initialises the beam as empty list. It turns the attributes to zero and sets Notina=FALSE.

void SetNumNote(int nn)

It sets NumNote=nn.

int GetNumNote()

It returns NumNote.

void SetFirstNota(Nota *pn)

It sets FirstNota=pn.

Nota *GetFirstNota()

It returns FirstNota.

void SetLastNota(Nota *pn)

It sets LastNota=pn.

Nota *GetLastNota()

It returns LastNota.

v**oid SetNotina(Bool b)**

It sets Notina=b. It cannot be performed when in the beam there are some notes. Furthermore is the dimension of the notes that are inserted that decides the dimension of the beam.

Bool GetNotina()

It returns Notina.

void SetSpaceWidth(VUnit sw)

It sets SpaceWidth=sw.

VUnit GetSpaceWidth()

It returns SpaceWidth.

void SetLGambo(VUnit lg)

It sets at *lg* the stem of the most meaningful note. If |*lg*| < *LGAMBOMIN*, LGambo takes the value *LGAMBOMIN* se *lg* ≥ 0 or *–LGAMBOMIN* if *lg* < 0 (if the beam dimension is small the comparisons are made with *LGAMBOMIN/2*). It can be performed after inserting in the beam a note at least, so that the beam knows (through Notina) if it is composed by normal or small notes.

void SetGamboUp()

It sets the stem above the beam with default length (half default length if the note is a small one). I has to be performed after inserting in the beam a note at least, so that the beam knows (through Notina) if it is composed by normal or small notes.

void SetGamboDwn()

It sets the stem below the beam with default length (half default length if the note is a small one). I has to be performed after inserting in the beam a note at least, so that the beam knows (through Notina) if it is composed by normal or small notes.

VUnit GetLGambo()

Returns the length of LGambo.

void SetNoteID(ClassID id)

It sets NoteID at id.

ClassID GetNoteID()

It returns NoteID.

Bool AddFigura(Figura *pf)

It adds one figure to the beam and increases it of 1 NumSpaces. Only the notes (and chords) with value lower or equal to the 8th note and the spaces are admitted to form the beams. If we want to insert in the beam a figure that does not belong to these types, the method returns FALSE. The figures have to be inserted in the beam ordered in horizontal position, from left to right. The spaces can be inserted also at the beginning or at the end of the beam but it is better not to insert them in these positions in order not to create confusion with the spaces outside the beam. The first note inserted in the beam decides the direction of the beam's stem, the value and the dimension of the notes of the beam. If we want to add a note with a different value or a different dimension, such note is not added and the method returns FALSE. In the other cases the method returns TRUE. It is not necessary that the notes inserted have all the stem in the same direction.

Bool AddAfterFigura(Figura* pf1, Figura* pf2)

Adds the figure pf2 to the group after the figure pf1. Accepts the spaces and the pauses normal.

Bool DelFigura(Figura *pf)
Deletes a figure from the group.
void Adjust(Bool primaleg, Battuta *pBattuta, int layer)
Set up the figures belonging to the group.
void Draw()
Drawing method for the group of notes.
void print()
Printing method for the group of notes.
void DrawTratto(Nota *pNota, short nBarra, short Direz, Vunit Bthick, Vunit dXGambo)
Drawing method for the line grouping the notes.
void printTratto(Nota *pNota, short nBarra, short Direz, Vunit Bthick, Vunit dXGambo)
Print a line of the grouping tratto that groups the notes.
void DrawBarre()
Drawing method for the bars grouping the notes.
void printBarre()
Print method for the bars grouping the notes.
Bool DeleteSym(DrawObject *sym)
Deletes the symbol specified if it is from the ones connected to the group. It returns TRUE if the symbol has been canceled.
int GetMedia()
Returns the parameter media of a group of notes. Media= sum heights of notes/ number of notes.
int GetBeamDis()
Returns a parameter to determine the direction of the stem in the beams.
int GetDelta()
Returns the parameter delta used to determine the slope of the beam.
int GetMeanHighLow()
Returns the parameter media=Highest+Lowest/2.
int GetMean()
Returns the parameter media= sum heights/number of notes.
Bool isOne (void)
Returns TRUE if there is only one note or if it is empty.
Figura* ExtractFigura()
It extracts from the beam the figure (note or space) on the extreme left. It decreases of 1 NumSpaces. If the beam is empty, it returns NULL. After executing this method, it is necessary to perform Adjust both on the extracted note (in case of a space we do not need it), and on the beam remaining. This is not necessary if the note and the beam are set with SetPos, because SetPos calls Adjust.
void SetPos(DrawObject* d,const Point& p)
It sets the position of the beam and calls Adjust.
Bool Hit(const Point& p, SelObj selObj, DrawObject*& drwObjSel, SymPath& symPath)
See general documentation of the Hit method in section 1.18.2.
Bool Hit(const Rectangle&, SelObj, SymPath& SymPath&)
See general documentation of the Hit method in section 1.18.2.
Bool SymbolicHit(SymPath&, SelObj selObj, DrawObject*&)
See general documentation of the method in section 1.18.2.
Bool FigPtr2SymPath(SymPath&, Figura*)
It reconstructs the path in a figure starting from the pointer.
void Select()
It designs on the screen the rectangle that surrounds the beam complete with symbols, with colour XOR_SELECTED_COLOR.
void Deselect()
It deselects from the screen the rectangle designed by Select.
CmdResult SymCommand(SymCmd CSmd)
See the general documentation of the method SymCommand in section 1.19.

char *Describe(Context)

See in section….

18.5 Class Pausa

18.5.1 Description

The Pausa class is abstract and contains the attributes that unify the general functions for the rest figures. **Rest** means *a moment of silence in a piece of music intended as an execution prescription*. Therefore to the figures whose duration has been fixed by the notes correspond as many homonymous figures of rest. At class level, for each class son of note (Accordo excluded) there is a corresponding son of Pausa (e.g. NMinima and PMinima). Other classes have been introduced in addition: PDueBattute (corresponding to a duration of eight quarters, equivalent to the class Nbreve), PQuattroBattute (corresponding to a duration of sixteen quarters) and PGenerica (where the duration is expressed by a number).

The management of rests is easier with respect to that of notes. For example there are not beams of rests or chords of rests. In addition, with respect to Figura, Pausa has not a relation IS_REFERRED_BY with other classes, thus the symbols that can surround a rest are only those of Figura: augmentation dot, fermata, instrument, glasses, dynamic text, generic text and annotation. Concerning the instrumental indications only the timpani can be associated to a rest and for the moment it has been decided to permit only the rests to refer a timpani symbol.

This class inherits from Figura the Altezza attribute. From the musical point of view a rest has no height but this attribute is useful to design rests on each height of the staff. This is indispensable writing polyphonic music: if the rests of a layer cannot be moved they are likely to overlap the figures of the other layer. The positioning of a rest on the basis of Altezza is ruled by the method SetPos of Figura, that Pausa inherits: the rest is placed on the line or space specified by Altezza.

18.5.2 Father Class

Figura

18.5.3 Child Classes

PGenerica, PQuattrobattute, PDueBattute, PSemibreve, PMinima, PSemiminima, PCroma, PSemicroma, PBiscroma, PSemibiscroma, PFusa.

18.5.4 Private Methods

CmdResult CmdPausaTimpano(Bool, Bool, unsigned char *)

It sets the instrument (timpani) to the rest. If it is successful, it returns CMD_OK, otherwise CMD_FAIL. It is necessary to specify AboveF=TRUE for the instrument above the figure or FALSE for the instrument below the figure.

CmdResult CmdPausaNota2Notina()

On mouse event sets the notina parameter.

CmdResult CmdPausaNotina2Nota()

On mouse event sets the notina parameter.

void AdjustStrumento (Vunit Ylinea1, Vunit Ylinea5)

Adjust method for the strumento symbol (timpano etc.).

18.5.5 Public Methods

Pausa()

It performs Init.

Pausa(double)

It performs Init setting durata.

Void Suona(MoodsMidiWrite*)

Prepare the file to be executed via MIDI file.

Bool IsARest()

Returns TRUE.

Void Adjust (Bool primaleg, Battuta *pBattuta, int layer)

Prepares the symbol for the drawing.

Void Draw()

Drawing method for the pausa.

Void print()

Printing method.

void Init()

It initialises the rest, by setting Altezza=0 and NumSpaces=1 and putting NULL all the pointers. It also initialises the rectangles that delimits the rest.

Bool DeleteSym(DrawObject *sym)

It deletes the specified symbol if it is among those linked to the rest. It returns TRUE if the symbol has been successfully deleted.

CmdResult SymCommand(SymCmd)

See the general documentation on the SymCommand method in section 1.19.

char *Describe(Context)

Description of the symbol**.**

18.6 Class NBiscroma

18.6.1 Description

The NBiscroma class implements the symbol of 1/32 duration note.

18.6.2 Father Class

Nota

18.6.3 Public Methods

NBiscroma()

Constructor.

void Init()

It performs Init of Nota setting the appropriate ID.

v**oid AdjustF ()**

Prepares the symbol for the drawing.

void Draw()

Drawing method for the symbol.

void print()

Printing method.

char *Describe(Context)

Description of the symbol**.**

void DrawForAccordo();

Drawing method for the symbol in a chord, taking in account the setting of the flags.

void printForAccordo();

Printing method.

18.7 Class NBreve

18.7.1 Description

The NBreve class implements the symbol of breve note.

18.7.2 Father Class

Nota

18.7.3 Public Methods

NBreve()

Constructor.

void Init()

It performs Init of Nota setting the appropriate ID.

v**oid AdjustF ()**

Prepares the symbol for the drawing.

void Draw()

Drawing method for the symbol.

void print()

Printing method.

char *Describe(Context)

Description of the symbol**.**

void DrawForAccordo();

Drawing method for the symbol in a chord, taking in account the setting of the flags.

void printForAccordo();

Printing method.

void SetLGambo (Vunit lg)

Does nothing. This note does not have stem.

void SetCoda (Bool Coda)

Does nothing. This note does not have flag.

18.8 Class NCroma

18.8.1 Description

The NCroma class implements the symbol of croma note .

18.8.2 Father Class

Nota

18.8.3 Protected Attributes

Bool Acciaccatura

TRUE if the note is a grace note of the kind “acciaccatura”

18.8.4 Public Methods

NCroma()

Constructor.

void Init()

It performs Init of Nota setting the appropriate ID.

void AdjustF ()

Prepares the symbol for the drawing.

void Draw()

Drawing method for the symbol.

void print()

Printing method.

char *Describe(Context)

Description of the symbol**.**

void DrawForAccordo();

Drawing method for the symbol in a chord, taking in account the setting of the flags.

void printForAccordo();

Printing method.

void SetAcciaccatura(Bool b);

Set the Acciaccatura parameter.

Bool GetAcciaccatura()

Returns the value of Acciaccatura.

CmdResult SymCommand(SymCmd&, Battuta* pBattuta)

Returns a command related to the conversion from grace note to nota and viceversa.

18.9 Class NFusa

18.9.1 Description

The NFusa class implements the symbol of fusa note.

18.9.2 Father Class

Nota

18.9.3 Public Methods

NFusa()

Constructor.

void Init()

It performs Init of Nota setting the appropriate ID.

void AdjustF ()

Prepares the symbol for the drawing.

void Draw()

Drawing method for the symbol.

void print()

Printing method.

char *Describe(Context)

Description of the symbol**.**

void DrawForAccordo();

Drawing method for the symbol in a chord, taking in account the setting of the flags.

void printForAccordo();

Printing method.

18.10 Class NMinima

18.10.1 Description

The NMinima class implements the symbol of minima note.

18.10.2 Father Class

Nota

18.10.3 Public Methods

NMinima()

Constructor.

void Init()

It performs Init of Nota setting the appropriate ID.

v**oid AdjustF ()**

Prepares the symbol for the drawing.

void Draw()

Drawing method for the symbol.

void print()

Printing method.

char *Describe(Context)

Description of the symbol**.**

void DrawForAccordo();

Drawing method for the symbol in a chord, taking in account the setting of the flags.

void printForAccordo();

Printing method.

void SetCoda (Bool Coda)

Does nothing. This note does not have flag.

18.11 Class NSemibiscroma

18.11.1 Description

The NSemibiscroma class implements the symbol of semibiscroma note.

18.11.2 Father Class

Nota

18.11.3 Public Methods

NSemibiscroma()

Constructor.

void Init()

It performs Init of Nota setting the appropriate ID.

v**oid AdjustF ()**

Prepares the symbol for the drawing.

void Draw()

Drawing method for the symbol.

void print()

Printing method.

char *Describe(Context)

Description of the symbol**.**

void DrawForAccordo();

Drawing method for the symbol in a chord, taking in account the setting of the flags.

void printForAccordo();

Printing method.

18.12 Class NSemibreve

18.12.1 Description

The NSemibreve class implements the symbol of semibreve note.

18.12.2 Father Class

Nota

18.12.3 Public Methods

NSemibreve()

Constructor.

void Init()

It performs Init of Nota setting the appropriate ID.

v**oid AdjustF ()**

Prepares the symbol for the drawing.

void Draw()

Drawing method for the symbol.

void print()

Printing method.

char *Describe(Context)

Description of the symbol**.**

void DrawForAccordo();

Drawing method for the symbol in a chord, taking in account the setting of the flags.

void printForAccordo();

Printing method.

void SetLGambo (Vunit lg)

Does nothing. This note does not have stem.

void SetCoda (Bool Coda)

Does nothing. This note does not have flag.

18.13 Class NSemicroma

18.13.1 Description

The NSemicroma class implements the symbol of semicroma note.

18.13.2 Father Class

Nota

18.13.3 Public Methods

NSemiminima()

Constructor.

void Init()

It performs Init of Nota setting the appropriate ID.

v**oid AdjustF ()**

Prepares the symbol for the drawing.

void Draw()

Drawing method for the symbol.

void print()

Printing method.

char *Describe(Context)

Description of the symbol**.**

void DrawForAccordo();

Drawing method for the symbol in a chord, taking in account the setting of the flags.

void printForAccordo();

Printing method.

18.14 Class NSemiminima

18.14.1 Description

The NSemiminima class implements the symbol of semiminima note.

18.14.2 Father Class

Nota

18.14.3 Public Methods

NSemicroma()
Constructor.

void Init()
It performs Init of Nota setting the appropriate ID.

v**oid AdjustF ()**
Prepares the symbol for the drawing.

void Draw()
Drawing method for the symbol.

void print()
Printing method.

char *Describe(Context)
Description of the symbol**.**

void DrawForAccordo();
Drawing method for the symbol in a chord, taking in account the setting of the flags.

void printForAccordo();
Printing method.

void SetCoda (Bool Coda)
Does nothing. This note does not have flag.

18.15 Class PBiscroma

18.15.1 Description

The PBiscroma class implements the symbol of biscroma pause.

18.15.2 Father Class

Pause

18.15.3 Public Methods

PBiscroma()
Constructor.

void Init()
It performs Init of Pausa setting the appropriate ID.

v**oid Adjust (Bool primaleg, Battuta *pBattuta, int layer)**
Prepares the symbol for the drawing.

void Draw()
Drawing method for the symbol.

void print()
Printing method.

char *Describe(Context)
Description of the symbol**.**

18.16 Class PCroma

18.16.1 Description

The PCroma class implements the symbol of croma pause.

18.16.2 Father Class

Pause

18.16.3 Public Methods

PCroma()
Constructor.

void Init()
It performs Init of Pausa setting the appropriate ID.

void Adjust (Bool primaleg, Battuta *pBattuta, int layer)
Prepares the symbol for the drawing.
void Draw()
Drawing method for the symbol.
void print()
Printing method.
char *Describe(Context)
Description of the symbol.

18.17 Class PFusa

18.17.1 Description

The PFusa class implements the symbol of fusa pause.

18.17.2 Father Class

Pause

18.17.3 Public Methods

PFusa()
Constructor.
void Init()
It performs Init of Pausa setting the appropriate ID.
void Adjust (Bool primaleg, Battuta *pBattuta, int layer)
Prepares the symbol for the drawing.
void Draw()
Drawing method for the symbol.
void print()
Printing method.
char *Describe(Context)
Description of the symbol.

18.18 Class PGenerica

18.18.1 Description

The PGenerica class implements the symbol of generica pause.

18.18.2 Father Class

Pause

18.18.3 Public Methods

PGenerica()
Constructor.
void Init()
It performs Init of Pausa setting the appropriate ID.
void Adjust (Bool primaleg, Battuta *pBattuta, int layer)
Prepares the symbol for the drawing.
void Draw()
Drawing method for the symbol.
void print()
Printing method.
char *Describe(Context)
Description of the symbol.

18.19 Class PDueBattute

18.19.1 Description

The PueBatutte class implements the symbol of two measures pause.

18.19.2 Father Class

Pause

18.19.3 Public Methods

PDueBattute()
Constructor.
void Init()
It performs Init of Pausa setting the appropriate ID.
void Adjust (Bool primaleg, Battuta *pBattuta, int layer)
Prepares the symbol for the drawing.
void Draw()
Drawing method for the symbol.
void print()
Printing method.
char *Describe(Context)
Description of the symbol.

18.20 Class PMinima

18.20.1 Description

The PMinima class implements the symbol of minima pause.

18.20.2 Father Class

Pause

18.20.3 Public Methods

PMinima()
Constructor.
void Init()
It performs Init of Pausa setting the appropriate ID.
void Adjust (Bool primaleg, Battuta *pBattuta, int layer)
Prepares the symbol for the drawing.
void Draw()
Drawing method for the symbol.
void print()
Printing method.
char *Describe(Context)
Description of the symbol.

18.21 Class PQuattroBattute

18.21.1 Description

The PQuattroBattute class implements the symbol of four measures pause.

18.21.2 Father Class

Pause

18.21.3 Public Methods

PQuattroBattute()
Constructor.
void Init()
It performs Init of Pausa setting the appropriate ID.
void Adjust (Bool primaleg, Battuta *pBattuta, int layer)
Prepares the symbol for the drawing.
void Draw()
Drawing method for the symbol.
void print()
Printing method.
char *Describe(Context)
Description of the symbol.

18.22 Class PSemibiscroma

18.22.1 Description

The PSemibiscroma class implements the symbol of semibiscroma pause.

18.22.2 Father Class

Pause

18.22.3 Public Methods

PSemibiscroma()
Constructor.

void Init()
It performs Init of Pausa setting the appropriate ID.

v**oid Adjust (Bool primaleg, Battuta *pBattuta, int layer)**
Prepares the symbol for the drawing.

void Draw()
Drawing method for the symbol.

void print()
Printing method.

char *Describe(Context)
Description of the symbol**.**

18.23 Class PSemibreve

18.23.1 Description

The PSemibreve class implements the symbol of semibreve pause.

18.23.2 Father Class

Pause

18.23.3 Public Methods

PSemibreve()
Constructor.

void Init()
It performs Init of Pausa setting the appropriate ID.

v**oid Adjust (Bool primaleg, Battuta *pBattuta, int layer)**
Prepares the symbol for the drawing.

void Draw()
Drawing method for the symbol.

void print()
Printing method.

char *Describe(Context)
Description of the symbol**.**

18.24 Class PSemicroma

18.24.1 Description

The PSemicroma class implements the symbol of semicroma pause.

18.24.2 Father Class

Pause

18.24.3 Public Methods

PSemicroma()
Constructor.

void Init()
It performs Init of Pausa setting the appropriate ID.

v**oid Adjust (Bool primaleg, Battuta *pBattuta, int layer)**
Prepares the symbol for the drawing.

void Draw()
Drawing method for the symbol.

void print()
Printing method.

char *Describe(Context)

Description of the symbol**.**

18.25 Class PSemiminima

18.25.1 Description

The PSemiminima class implements the symbol of semiminima pause.

18.25.2 Father Class

Pause

18.25.3 Public Methods

PSemiminima()

Constructor.

void Init()

It performs Init of Pausa setting the appropriate ID.

v**oid Adjust (Bool primaleg, Battuta *pBattuta, int layer)**

Prepares the symbol for the drawing.

void Draw()

Drawing method for the symbol.

void print()

Printing method.

char *Describe(Context)

Description of the symbol**.**

18.26 Class RipBattuta

18.26.1 Description

The RipBattuta class implements the symbol of repetition measure.

18.26.2 Father Class

Ripetizione

18.26.3 Public Methods

RipBattuta()

Constructor.

void Init()

It performs Init setting the appropriate ID.

v**oid Adjust (Bool primaleg, Battuta *pBattuta, int layer)**

Prepares the symbol for the drawing.

void Draw()

Drawing method for the symbol.

void print()

Printing method.

char *Describe(Context)

Description of the symbol**.**

18.27 Class Ripetizione

18.27.1 Description

The Ripetizione class implements the symbol of repetition measure.

18.27.2 Father Class

Figura

18.27.3 Public Methods

Ripetizione()

Constructor.

void SetPos(class DrawObject*, const class Point&)

Set the position for the symbols to be drawn

void Init()

It performs Init setting the appropriate ID.

void Adjust (Bool primaleg, Battuta *pBattuta, int layer)
Prepares the symbol for the drawing.
void Draw()
Drawing method for the symbol.
void print()
Printing method.
CmdResult SymCommand(SymCmd&, Battuta *pb=NULL)
Returns CMD_FAIL

18.28 Class RipMezzaBattuta

18.28.1 Description

The RipMezzaBattuta class implements the symbol of repetition half measure.

18.28.2 Father Class

Ripetizione

18.28.3 Public Methods

RipMezzaBattuta()
Constructor.
void Init()
It performs Init setting the appropriate ID.
void Adjust (Bool primaleg, Battuta *pBattuta, int layer)
Prepares the symbol for the drawing.
void Draw()
Drawing method for the symbol.
void print()
Printing method.
char *Describe(Context)
Description of the symbol**.**

18.29 Class RipTempo

18.29.1 Description

The RipTempo class implements the symbol of time repetition .

18.29.2 Father Class

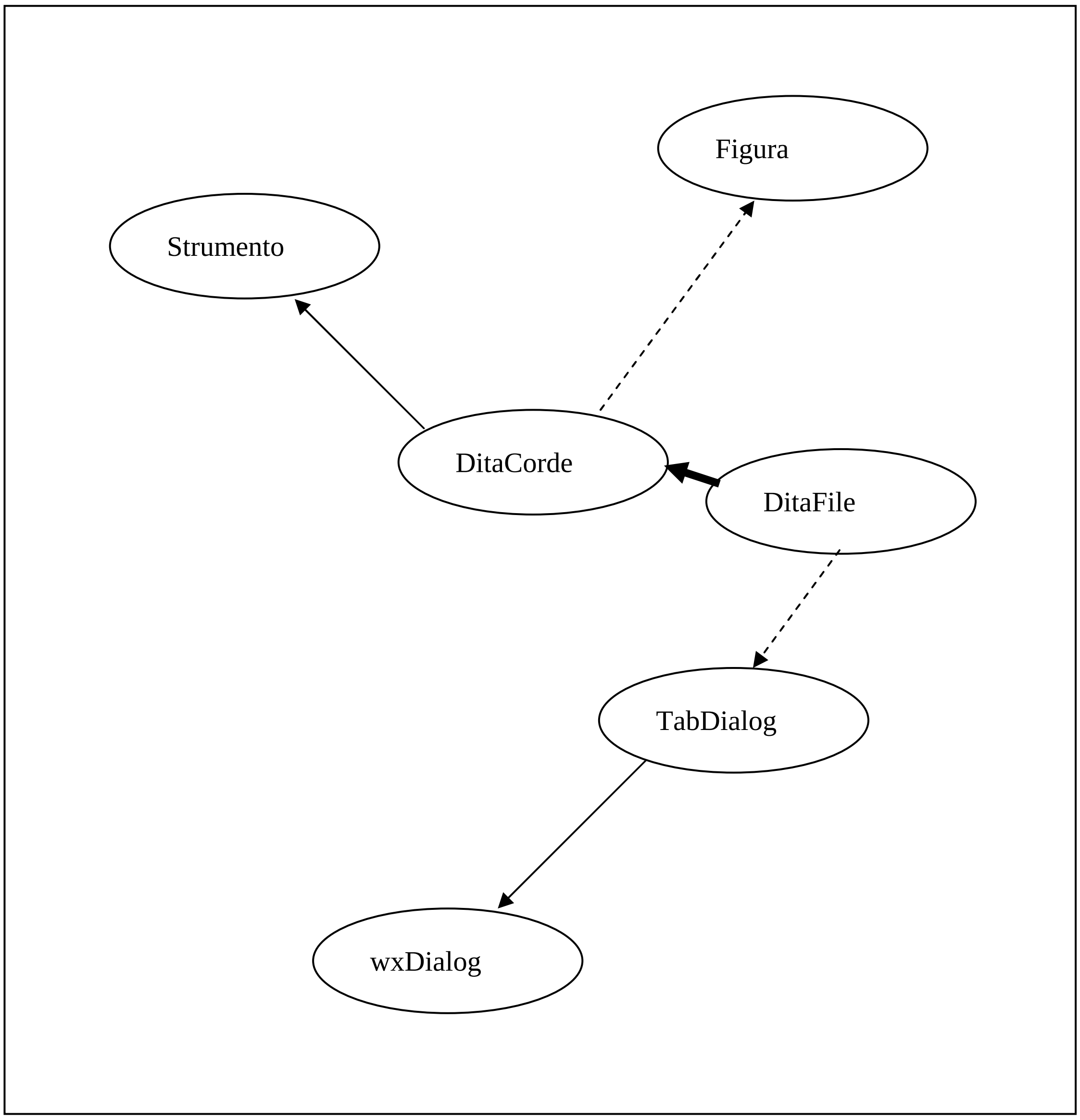
Ripetizione

18.29.3 Public Methods

RipTempo()
Constructor.
void Init()
It performs Init setting the appropriate ID.
void Adjust (Bool primaleg, Battuta *pBattuta, int layer)
Prepares the symbol for the drawing.
void Draw()
Drawing method for the symbol.
void print()
Printing method.
char *Describe(Context)
Description of the symbol**.**

19 Fretboard & Note Headtypes

19.1 Diagram of the classes



Legend:

⟶ **relation "IS A…"**
⇢ **relation "IS REFERRED BY…"**
➡ **relation "USE…"**

19.2 Class DitaCorde

19.2.1 Description

This class contains the fretboards information and provides to draw it on the screen. Its attributes get all important data to describe a fretboard symbol. Like Strumento can be put up or down the note. It is referred by Figura and it can be placed anywhere in a measure (on a note, on a rest, on a blank space). If you want to create a new fretboard only using this class you have to code your information in this mode: to press the 2nd fret whit the 3rd finger on the 6th string you have to set the first (because the order of string number is reversed) character of **frets** string as ‘2’ and of **fingers** string as ‘3’. The other rules and some example are explained in the attributes section. We suggest to use this class creating by DitaFile object that gets information by a fretboards’ archive (standard position).

19.2.2 Father Class

Strumento

19.2.3 Protected attributes

char* name
contains the nomenclature of the chord position (e.g.: F, Cmaj7, Dm5b);

int nstrings
counts the number of strings of instrument (e.g.: Guitar has 6);

char* frets
is a string of **nstrings** characters that specifies for each string the fret to press (e.g.: C on the guitar produces “x32o1o” where ‘o’ means to play the string without pressing any frets, whereas ‘x’ means to mute the string);

char* fingers
as frets is a string of characters that shows which finger must press the string (e.g.: C on the guitar gives “732717” where ‘7’ means that no fret is pressed and rarely compares ‘9’ that means to play with thumb);

int head
indicates the start fret of the chord position if it is different from the first (e.g.: on the guitar Eb is played at the 6th);

char* barre
indicates the starting and the ending string of possible barré (e.g.: on the guitar F has barré that starts from the 1st string and finishes at the 6th string; it is written as “16”); the fret is not indicated because is obtained by the fret pressed on starting string.

19.2.3.1 Public Method

Ditacorde()
is the default constructor that creates a 0-string fretboard (it is not interesting);

Ditacorde(int nstrings, char name[], char frets[], int head, char fingers[], char barre[3])
is the constructor that fills the attributes of Ditacorde class to create a new fretboard;

Ditacorde(Ditacorde &InChord)
is the copy constructor;

~Ditacorde()
is the destroyer that deallocates memory;

char* GetName()
returns the char string with the chord nomenclature;

char* GetNote()
gets the tonic note and returns it as a char string (e.g.: C7 returns “C”);

int GetNStrings()
gets the number of the strings of the fretboard;

void Draw()
draws the fretboard on the screen;

void Print()

prints the fretboard on a PostScript file;

char* Describe()

specifies the properties as a char string used for the save procedure.

19.3 Class DitaFile

19.3.1 Description

This class was created to use the text type archive using its public methods. DitaFile can move in the file always pointing to fretboard position, it can search the user's requested fretboards. The other methods are important because they can count and list all fretboards founded for a chosen nomenclature, all nomenclatures (with unless one fretboard) for a chosen tonic note, the available tonic notes for a chosen number of strings. After you had pointed in the file after a search, you can read information and build a **DitaCorde** object using the **GetChord(…)** method.

19.3.2 Protected Attributes

FILE* fp

is the file pointer to the position fretboards archive;

19.3.3 Public Methods

Ditafile(char* filename)

is the constructor that opens the **filename** archive;

~Ditafile()

is the destroyer that closes the archive;

int Open(char* filename)

opens the **filename** archive;

void Close()

closes the archive;

int IsOpen()

returns TRUE if an archive is open;

int IsOnEOF()

returns TRUE if is the pointer is at the end of the file;

int GoToPos(int pos)

moves the file pointer at the fretboard in the position **pos**;

int Search(const char* ChName)

looks througth the file and finds the fretboard with chord nomenclature **ChName**;

int Search(const char* ChName, int var)

searches the **var** variant fretboard with chord nomenclature **ChName**;

Ditacorde* GetChord(int* outvar=NULL, char* comment=NULL)

reads at the current position in the file and creates a Ditacorde object, returns his pointer and fills the output variables with number of read variant and its possible comment;

int NotesCount(int nstrings)

counts the number of tonic notes with **nstrings** strings;

int NotesList(int nstrings, char myList)**

fills myList array of character strings with the tonic notes and **nstrings** strings, returns the count;

int NamesCount(int nstrings, char* Note)

counts the number of available nomenclatures with the tonic **Note** chosen and **nstrings** strings;

int NamesList(int nstrings, char* Note,char myList)**

as the NoteList fills the array **myList** with the available nomenclatures;

int VarsCount(int nstrings, char* Name)

counts the number of different fretboards for the same chord nomenclature;

int VarsList(int nstrings, char* Name, char comment, int* vars, DitaCorde** Chords)**

makes a list of different fretboards with the same chord nomenclature and fills the output variables as an array of variant identifiers, an array of comment strings and **Chords** as an array of Ditacorde references;

19.3.4 Protected Methods

char* SymTrad(char* ToTrad)

a private function that translates archive text used by GetChord method (#, b and ° are differently coded).

19.4 Class Nota

19.4.1 Modification Description

Class Nota was modified to represent notes with different head types. This new information is contained in **HeadType** attribute and there are methods to read and write this attribute. Other information is contained in the **DxGambo** and **DyGambo** attributes: these specify the variable position of the stem starting point with respect to the note head centre and they also can be read and write using methods. There are two methods **GetXgambo()**,**GetYgambo()** to calculate absolute position of the stem starting point used by many draw methods.

The previously listed attributes are filled by **Adjust(...)** that asks to Milla the context rules to apply; there are two types of rules that inform about the head code to print and the shift of the stem attachment.

The Draw method is exposed in the draw functions: they show the note head, the additional cuts, the stem and the possible hooks…

19.4.2 New Protected Attributes

NoteHead_ID HeadType

specifies the head type of the note;

unsigned char HeadCode[3]

contains the ASCII characters to print as note head if head type is alphanumeric else the ASCII code to print with the music font;: in this case this code is determined by Ajdust method;

int DxGambo, DyGambo

contains the shifts of stem starting point from the note head centre (they are normalised as to the size of note head code, their values can be –1,0,1).

19.4.3 New Public Methods

void DrawTesta()

draws the head of the note;

void DrawTagli()

draws the additional cuts (if it is necessary);

void DrawCode()

draws the hooks;

void PrintTesta()

prints the head of the note;

void PrintTagli()

prints the additional cuts (if it is necessary);

void Print Code()

prints the hooks;

void SetHeadType(NoteHead_ID headtype)

sets the note head type as **headtype**;

NoteHead_ID GetHeadType()

returns the note head type;

void SetHeadCode(unsigned char* codes)

sets the head ASCII codes as **codes**;

void SetDxGambo(int dx), void SetDyGambo(int dy)

sets the horizontal/vertical shift of the stem extremity from the note head centre;

int GetDxGambo(), int GetDyGambo()

returns the horizontal/vertical shift of the stem extremity from the note head centre;

int GetXgambo(), int GetYgambo()

returns the horizontal/vertical *absolute position* of the stem starting point from the note head centre.

19.5 Saving and loading

The saving and loading operations in Moods are based on the definition of a recognition grammar defined for this purpose.
The saving of the class DitaCorde imposes to add to the symbol “tipostrumento” (that contains also “archi”, “strumentiafiato”..), the instrument “tablature” that develops in a sequence of strings that contain all

```
<tipostrumento>:= …
             | <tablature>

<tablature>:= TABLATURE DITA_NUMCORDE INTEGER DITA_NOME STRING
       DITA_TASTI STRING DITA_DITA STRING DITA_BARRE STRING
       DITA_TASTO INTEGER <updown>
```

the necessary information for reconstructing the object. In order to distinguish each information a recognising string precedes.

Example of a saving string::

```
TABL NSTR 6 CHNAM "E7" FRETS "o2o1oo" FINGS "727177" BARRE "00" HEAD 0 UP
```

For the class Nota has been necessary to extend the previous recognition structure in order to enclose the

```
<tiponotacompleta>:= <tiponota> <headnota> <spacing>

<headnota>:= | HEADNOTA <tipoheadnota>

<tipoheadnota>:= CLASSIC
             | ALPHANUM
             | ALPHANUM_SQUARE
             | ALPHANUM_REVERSE
             | CIRCLEX
             | …
             | SQUARE
```

information concerning the type of notehead. We added to the symbol “tiponotacompleta” the identifier “headnota” that includes the type of notehead preceded from a password.

Example of saving strings obtained from the class Nota:

```
BF 8 NW   HEADNOTA  DDIESIS HN 2   EF
BF 97 N4   HEADNOTA  CLASSIC S UP  HN 2   EF
```

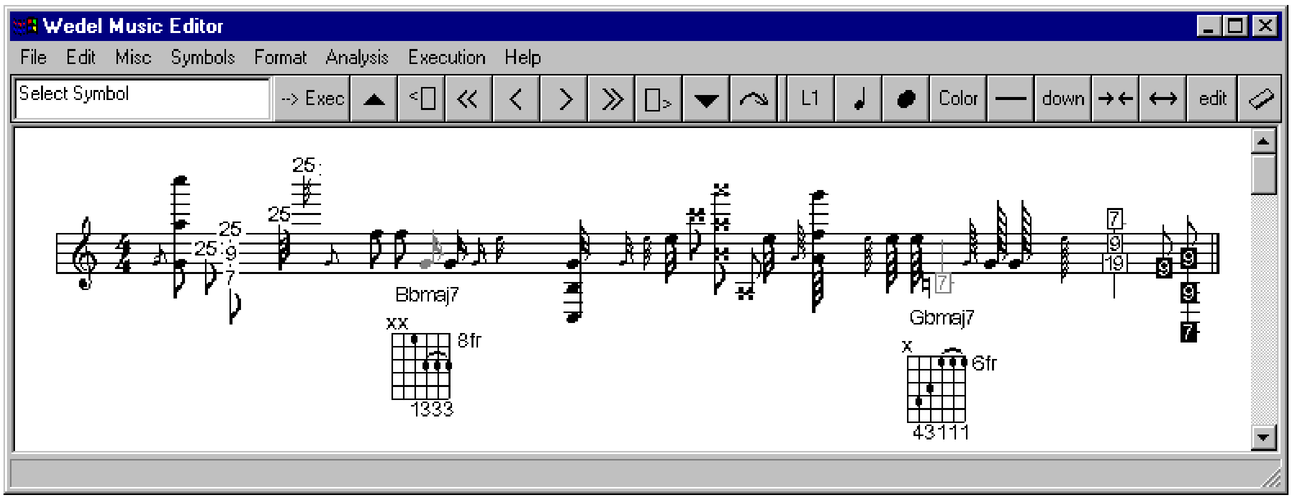
In the case of note with notehead of an alphanumerical type is important to save also the string of ASCII codes inside it. It is important to consider this information typical of the note even if it is in a chord whereas is not necessary to specify the type of notehead (it is already indicated in the chord).

<structnota>:= …
| <headcode>

<headcode>:= | HEADCODE STRING

An analogue distinction has been made for the attributes duration and height. In fact the saving of the string of codes can be found in the construct "structnota" (where we can find also the height) that is specified in each note of a chord.

```
BAC 1
      N4    HEADNOTA  ALPHANUMREVERSE S DWN   FING UP
             NA 137   HN -3    HEADCODE "5"
             NA 1   HN 5    HEADCODE "11"
             NA 136   HN 12    HEADCODE "5" BA   SH   EA
EAC

BAC 112
      N16    HEADNOTA  CLASSIC S DWN   FING UP
                   NA 112   HN 8
                   NA 114   HN 12
                   NA 115   HN 19
EAC
```



Interaction with Milla

Milla consents to understand and apply in the editing phase, all the necessary rules for the correct formatting of a musical text; it is based on rules divided into two groups, introduction and positioning, that intervene when particular conditions of the system verify.
In order to make Milla support our requirements it is necessary to introduce, in a grammar similar to that of the saving, new constructs for rules and conditions.

The rules to be introduced concern the problem of the positioning in two different situations:
1. the acquirement of the ASCII code of the notehead;
2. the shifts of the beginning of the stem from the centre.

The condition that estimates the type of notehead is added to the "condition" symbol in the form <parola chiave per la testa> followed by <tipo di testa>; this consents to move this condition with others of different kind as the duration.

```
<condition>:=…
              | HEAD <HeadType>

<HeadType>:= CLASSIC
              | ALPHANUM
              | ALPHANUM_SQUARE
              | ALPHANUM_REVERSE
              | CIRCLEX
              | …
              | SQUARE
```

Our conditions for acquiring the code result from the combination of the test on the type of notehead and on its duration; the conditions manage also the reduced noteheads.

```
posIF HEAD DDIESIS AND NOTE NOTINA THEN CodeDDiesisS;
posIF HEAD RHYTHMIC AND NOTE SEMIMINIMA THEN CodeRhythmicB;
posIF HEAD RHYTHMIC AND NOTE MINIMA THEN CodeRhythmicW;
```

For the positioning of the stem, on the other hand, the conditions for the application of the rule combine the condition on the type of notehead with that on the direction of the stem.

```
posIF HEAD CLASSIC AND NOTE STEMUP THEN StemStartMiddleRight;
posIF HEAD CLASSIC AND NOTE STEMDOWN THEN StemStartMiddleLeft;
posIF HEAD DDIESIS AND NOTE STEMUP THEN StemStartTopRight;
```

Concerning the creation of new rules, the first type is inserted in Milla through the addition of the HEAD symbol in the construct "Rule" that finds the rules concerning the notehead. In addition we have permitted the rule to specify a code adding to the "staff" construct the variant <parola chiave per il codice> = <codice>.

```
RULEPOS CodeDDiesis HEAD RELNOTA CODE=154;
RULEPOS CodeDDiesisS HEAD RELNOTA CODE=93;
RULEPOS CodeRhythmicT HEAD RELNOTA CODE=170;
```

The kind of rules that concern the distance of the stem is implemented adding the STEMSTART symbol. It reuses the construct "coord" previously specified.

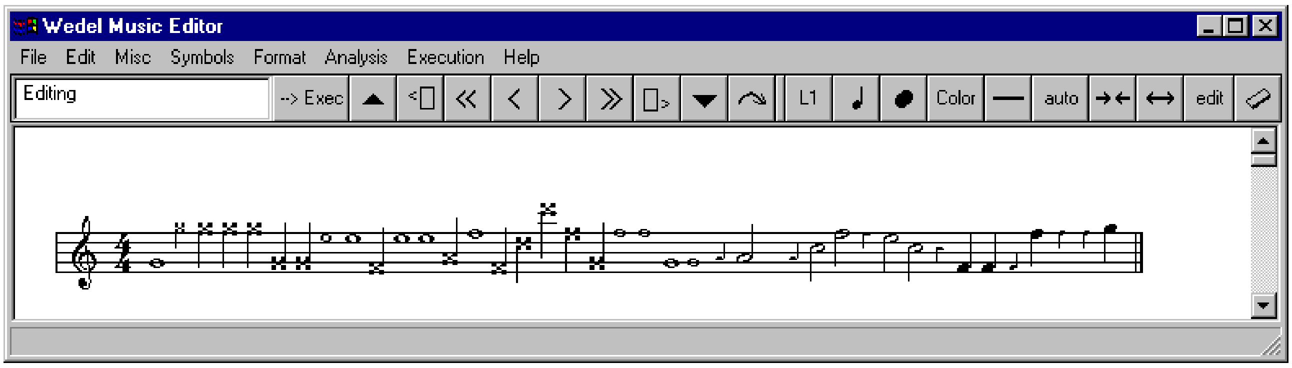
```
RULEPOS StemStartBottomCenter STEMSTART RELNOTA DX=0 DY=-1;
RULEPOS StemStartTopRight STEMSTART RELNOTA DX=1 DY=1;
RULEPOS StemStartMiddleRight STEMSTART RELNOTA DX=1 DY=0;
```

Let's formalise our additions to Milla grammar.

```
<Symbol>:= …
          |  HEAD
          |  STEMSTART

<staff>:=…
         | <code>

<code>:= CODE EQUAL INTEGER
```

The additions to the interpreter language of Milla had as a result the revision of the modules that manage it as the rulemanager and the lexical part as well as the creation of the method for asking to Milla the rules to be applied.

Description of the user's interface

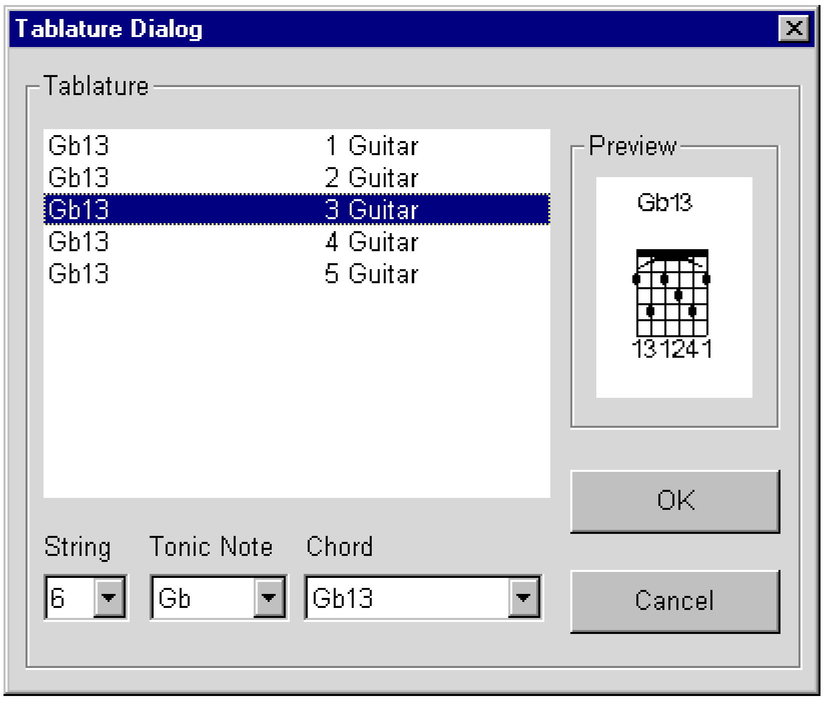


19.5.1 Insertion of fretboard inside Moods

The command for the insertion is in the menu *Symbols* under the word *Fretboards….*
After selecting such word the applicative asks for the choice of the position (note, rest or empty space) where we would like to associate the fretboard.
After selecting the position appears a dialog-box for the choice of the desired fretboard.

19.5.2 Description of the dialog-box

The dialog-box controls are: 3 combo-box, one list-box, one preview-window, a button for the confirmation.



Tablature Dialog

Tablature

Gb13	1 Guitar
Gb13	2 Guitar
Gb13	3 Guitar
Gb13	4 Guitar
Gb13	5 Guitar

Preview

Gb13

131241

String: 6
Tonic Note: Gb
Chord: Gb13

19.6 Combo-box

The user can scan the list of the available chords through 3 controls of the combo-box type that specify the following features of the sought fretboard.

1. Number of strings (# Strings): is the combo-box that has the priority; it provides updating the next ones according to the availability of the list of chords.
2. Tonic Note: selects the fundamental of the chord (e.g..: Cm7 has tonic C); this combo-box as well has effects on the list of the available nomenclatures.
3. Nomenclature: visualises all the possible nomenclatures.

19.7 List-box

All the variants available in the list that satisfy the specifications of the three combo-box are listed in the list-box where they are accompanied by the related comment. It is possible to chose the preferred one.

19.8 Preview

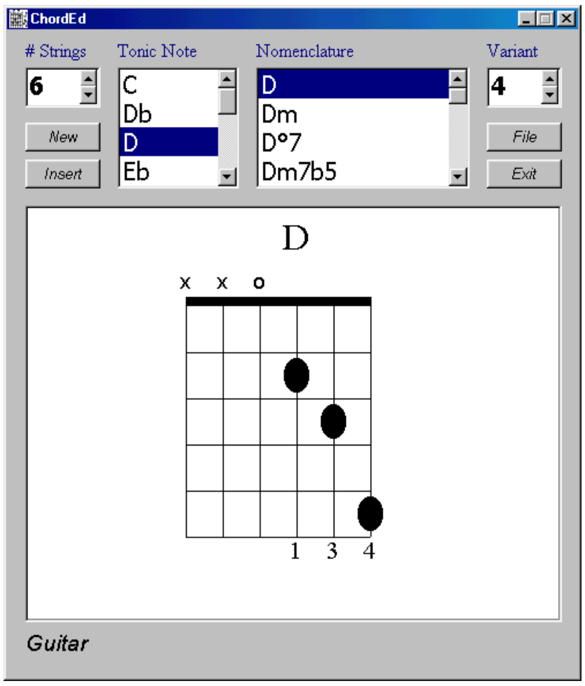
The preview window allows a preview of the fretboard selected in the list-box and a visual confirmation of the chosen chord.

19.9 Button

The Ok button allows positioning the selected chord in the score; it closes the dialog-box.

19.9.1 Description of the editor of the ChordED fretboard

The editor allows **visualising, modifying, creating** new fretboards in the list file elenco that can be used by Moods for the insertions inside the scores.
The interface presents, for the scanning of the fretboards in the current file, the following controls: two spin-edit (edit-box with arrows for the increment) for the choice of the number of strings and of the variant and two list-box containing the list of tonics and nomenclatures.
A big central window allows visualising the selected fretboard and editing it; in addition it is provided of a edit-box for its own comment that is placed under it.
Four buttons allow creating and inserting in the list new chords, choosing the archive to be scanned and exiting from the program.



19.10 Spin-edit # Strings

This control is always busy and allows setting the number of strings in the fretboard.
Among the controls for the scanning of the list it has the priority because if modified it affects all the others.
The number of strings can vary from 2 to 9.

19.11 List-box Tonic Note

It lists all the available tonics with the current number of strings and allows selecting the desired fundamental. <u>The fundamental notes are always expressed in flat for the uniformity of the archive.</u>

19.12 List-box Nomenclature

It lists all the available nomenclatures with the fundamental and the number of selected strings.

19.13 Spin-edit Variant

Each nomenclature can have more variants (positions on the fretboard) and these are found by increasing the content of this control.

19.14 Panel Preview-Edit

The window visualises the fretboard chosen by the other controls and consents the edit in the following modalities:

- Clicking on **name of chord** it is possible to modify it.

- Clicking on the fretboard **the positions of the fingers** on the strings and the eventual **barré** are modified; if we use the left button we can insert a finger on a string, move it if it is already available or delete it by clicking on it. The right button on the other hand is used to set the starting and the ending string of the barré: the first click with the right button determines the beginning whereas the second click sets the end, the barré can be deleted by clicking on it.
- The strings that are not pressed can be played “a vuoto” or can remain silent. The related symbols are “**o**” and “**x**”; clicking on them they alternate.
- The strings that are pressed are referred to the interested fingers (**1,2,3,4,T**) shown down. Clicking on them they can be modified thank to the appropriate list-box that appears.
- The indicators of the starting button are two: the **capotasto** (black bar-line on the top of the fretboard that delimits the end of the handle of the instrument) and the specific button on one side (“**n° tasto fr**”). The two indicators exclude each other. Clicking on the current indicator the arrows for the increment/decrement of the starting button appear.

19.14.1.1 Edit-box Comment

The comment related to a particular fretboard is visualised on this control that permits the modification.

19.14.1.2 Button New

It prepares a new fretboard with empty strings and changes the comment in *Insert by USER*.
The button is active only after loading a file.

19.14.1.3 Button Insert

It inserts the visualised fretboard in the current file. The button activates after a modification of the fretboard or after the command New and deactivates in the consulting phase. To be noticed: the command *Insert* controls if the name respects the specifications required from the archive and thus the messages *Nome non valido* (no valid name) or *Il nome è stato forzato* (the name has been forced) can be visualised; both messages don’t perform the insertion.

19.14.1.4 Button File

It opens a Open-dialog in order to charge the list to be consulted and modified. The predefined extension of the files is “.DAT” and the starting directory is the current one.

19.14.1.5 Button Exit

It ends the applicative.

20 IND module

The IND module is composed from a set of classes; each one implements a musical symbol of indication. The Indications considered are:

- Fingering;
- Expression indications;
- Indications related to the mute;

The Expression indications concern in general all the instruments and thus are typically already present in the scores in order to suggest to the musician a precise interpretation of the musical piece. Belong to this kind of indications:

- Portato;
- Sforzato;
- Accento;
- AccentoForte;
- PostStacc;
- Staccato;
- Punto Sopra.

The indications refer to notes and chords, therefore, in the classes that implement such figures we will refer to them. All the indications are children classes of a generic class Indicazione that descends from DrawObject. In fact all the indications are objects that can be drawn and thus they inherit from DrawObject, through Indicazione, all the necessary methods, redefining each time the Draw method. The classes that belong to this module are:

- Indicazione;
- Diteggiato;
- Espressione;
- Portato;
- Sforzato;
- Staccato;
- Accento;
- AccentoForte;
- PortStacc;
- PuntoSopra;
- Sordina;
- ConSord;
- ViaSord;

20.1 Class Indicazione

20.1.1 Description

This is an abstract class that has the purpose of representing all the indications that can appear on a musical score.

20.1.2 Father class

DrawObject

20.1.3 Children classes

Diteggiato, Espressione, Sordina

20.1.4 Protected attributes

aboveNota

Boolean that defines if the indication will be drawn above or below the figure that it refers to.

20.1.5 Public methods

Indicazione()

Constructor of the class.

Bool GetAboveNota()

It returns the attribute that shows if the symbol is to be drawn above or below the figure it refers to.

void SetAboveNota(Bool above)

It sets the Boolean attribute aboveNota TRUE if the indication goes above or FALSE if it goes below the figure it refers to.

void Draw()

It draws the indication.

20.2 Class Diteggiato

20.2.1 Description

This class allows designing the Diteggiatura. This is expressed through numbers that correspond to the fingers according to the following table:

1. Thumb
2. Forefinger
3. Middle finger
4. Ring finger
5. Little finger

Other signs indicating the fingers can exist but haven't been implemented.

20.2.2 Father class

Indication

20.2.3 Children classes

NONE.

20.2.4 Private attributes

dito

Character that represents the fingering with which the figure it refers to has to be played.

20.2.5 Public methods

Diteggiato()

Initialiser: it sets the default fingering (1) above the figure which it refers to and defines the class identifier.

void SetDito(char dit)

It sets the number related to the finger with which the figure it refers to has to be played.

VUnit GetVU2Up()

It returns the distance from the centre of the fingering symbol to the extreme superior point of the symbol itself.

VUnit GetVU2Dwn()

It returns the distance from the centre of the fingering symbol to the extreme inferior point of the symbol itself.

VUnit GetVU2Rgt()

It returns the distance from the centre of the fingering symbol to the extreme right point of the symbol itself.

VUnit GetVU2Lft()

It returns the distance from the centre of the fingering symbol to the extreme left point of the symbol itself.

void Draw()

It draws the fingering.

20.3 Class Espressione

20.3.1 Description

This is an abstract class that has the purpose of drawing the Espressione symbols. Not all the expression symbols have been implemented, nevertheless the most used for all the instruments have been chosen.

The expression symbols show a particular method to give out a sound or a group of sounds.

20.3.2 Father class

Indicazione

20.3.3 Children classes

AccentoForte, Portato, PortStacc, Sforzato, Staccato

20.3.4 Public methods

void Draw()

It draws the Espressione symbol.

char *Describe(Context)

20.4 Class AccentoForte

20.4.1 Description

This class allows drawing the accento forte symbol.

20.4.2 Father class

Espressione

20.4.3 Children classes

NONE.

20.4.4 Public Methods

AccentoForte()

Initialiser: it sets the Accentoforte symbol above or below the figure it refers to and defines the class identifier.

VUnit GetVU2Up()

It returns the distance from the centre of the Accentoforte symbol to the extreme superior point of the symbol itself.

VUnit GetVU2Dwn()

It returns the distance from the centre of the Accentoforte symbol to the extreme inferior point of the symbol itself.

VUnit GetVU2Rgt()

It returns the distance from the centre of the Accentoforte symbol to the extreme right point of the symbol itself.

VUnit GetVU2Lft()

It returns the distance from the centre of the Accentoforte symbol to the extreme left point of the symbol itself.

void Draw()

It draws the Accentoforte.

20.5 Class Portato

20.5.1 Description

This class allows designing the Portato symbol that on the score appears as an horizontal line; musically it means that the figure it refers to has to be played with a particular expressive intensity for all its duration, so that the following figure is separated by an imperceptible caesura, as if a slur was between the two figures.

Father class

Espressione

20.5.2 Children classes

NONE.

20.5.3 Public methods

Portato()
Initialiser: it sets the Portato above the figure it refers to and defines the identifier of the class.

VUnit GetVU2Up()
It returns the distance from the centre of the Portato symbol to the extreme superior point of the symbol itself.

VUnit GetVU2Dwn()
It returns the distance from the centre of the Portato symbol to the extreme inferior point of the symbol itself.

VUnit GetVU2Rgt()
It returns the distance from the centre of the Portato symbol to the extreme right point of the symbol itself.

VUnit GetVU2Lft()
It returns the distance from the centre of the Portato symbol to the extreme left point of the symbol itself.

void Draw()
It draws the Portato.

char *Describe(Context)
DESCRIBE_TXT

20.6 Class PortStacc

20.6.1 Description

This class is used to represent the portstaccato sign, that foresees two graphic objects obtained with bitmap. Such class allows the positioning of the above mentioned objects above or below the note.

Father class
Espressione

20.6.2 Children classes

NONE.

20.6.3 Public methods

PortStacc()
Initialiser: it sets the symbol above or below the figure it refers to and defines the identifier of the class.

VUnit GetVU2Up()
It returns the distance from the centre of the PortStacc symbol to the extreme superior point of the symbol itself.

VUnit GetVU2Dwn()
It returns the distance from the centre of the PortStacc symbol to the extreme inferior point of the symbol itself.

VUnit GetVU2Rgt()
It returns the distance from the centre of the PortStacc symbol to the extreme right point of the symbol itself.

VUnit GetVU2Lft()
It returns the distance from the centre of the PortStacc symbol to the extreme left point of the symbol itself.

void Draw()
It draws the PortStacc.

20.7 Class Sforzato

20.7.1 Description

This class allows designing the Sforzato symbol that on the score appears as a >; musically it means that the sound must be pointed out by stressing its emission.

Father class
Espressione

20.7.2 Children classes

NONE.

20.7.3 Public methods

Sforzato()
Initialiser: it sets the Sforzato symbol above the figure it refers to and defines the class identifier.
VUnit GetVU2Up()
It returns the distance from the centre of the Sforzato symbol to the extreme superior point of the symbol itself.
VUnit GetVU2Dwn()
It returns the distance from the centre of the Sforzato symbol to the extreme inferior point of the symbol itself.
VUnit GetVU2Rgt()
It returns the distance from the centre of the Sforzato symbol to the extreme right point of the symbol itself.
VUnit GetVU2Lft()
It returns the distance from the centre of the Sforzato symbol to the extreme left point of the symbol itself.
void Draw()
It draws the Sforzato symbol.

20.8 Class Staccato

20.8.1 Description

This is an abstract class that has the purpose of designing the Staccato symbols: Accento and Punto Sopra. The Staccato symbols mean that the duration of the figures they refer to must be decreased of an half or more so that the successive sounds appear separated by caesuras more or less evident.

Father class
Espressione

20.8.2 Children classes

Accento, PuntoSopra

20.8.3 Public methods

void Draw()
It draws the Staccato.

20.9 Class Accento

20.9.1 Description

This class allows designing the Accento symbol that on the score appears as a small full triangle upside down; musically it means that the value of duration of the figure is decreased of more than an half.

Father class

Staccato

20.9.2 Children classes

NONE.

20.9.3 Public methods

Accento()

Initialiser: it sets the Accento symbol above the figure it refers to and defines the class identifier.

VUnit GetVU2Up()

It returns the distance from the centre of the Accento symbol to the extreme superior point of the symbol itself.

VUnit GetVU2Dwn()

It returns the distance from the centre of the Accento symbol to the extreme inferior point of the symbol itself.

VUnit GetVU2Rgt()

It returns the distance from the centre of the Accento symbol to the extreme right point of the symbol itself.

VUnit GetVU2Lft()

It returns the distance from the centre of the Accento symbol to the extreme left point of the symbol itself.

void Draw()

It draws the Accento symbol.

char *Describe(Context)

20.10 Class PuntoSopra

20.10.1 Description

This class allows designing the PuntoSopra symbol that on the score appears as a dot; musically it means that the value of duration of the figure is decreased of an half.

Father class

Staccato

20.10.2 Children classes

NONE.

20.10.3 Public methods

Accento()

Initialiser: it sets the Punto Sopra symbol above the figure it refers to and defines the class identifier.

VUnit GetVU2Up()

It returns the distance from the centre of the Punto Sopra symbol to the extreme superior point of the symbol itself.

VUnit GetVU2Dwn()

It returns the distance from the centre of the Punto Sopra symbol to the extreme inferior point of the symbol itself.

VUnit GetVU2Rgt()

It returns the distance from the centre of the Punto Sopra symbol to the extreme right point of the symbol itself.

VUnit GetVU2Lft()

It returns the distance from the centre of the Punto Sopra symbol to the extreme left point of the symbol itself.

void Draw()

It draws the Punto Sopra symbol.

char *Describe(Context)

20.11 Class Sordina

20.11.1 Description

This class manages the graphic representation of the two symbols related to the mute.

Father class

Indicazione

20.11.2 Children classes

ConSord, ViaSord

20.11.3 Public methods

void Draw()
It draws the two symbols.

20.12 Class ConSord

20.12.1 Description

This class is used to draw the ‘with mute’ sign, that foresees a graphic object obtained with bitmap.

Father class

Sordina

20.12.2 Children classes

NONE.

20.12.3 Public methods

ConSord()
Initialiser: it sets the symbol above or below the figure, dependently on the attribute AboveNota of the symbol. It defines the class identifier.

VUnit GetVU2Up()
It returns the distance from the centre of the with mute symbol to the extreme superior point of the symbol itself.

VUnit GetVU2Dwn()
It returns the distance from the centre of the with mute symbol to the extreme inferior point of the symbol itself.

VUnit GetVU2Rgt()
It returns the distance from the centre of the with mute symbol to the extreme right point of the symbol itself.

VUnit GetVU2Lft()
It returns the distance from the centre of the with mute symbol to the extreme left point of the symbol itself.

void Draw()
It draws the with mute symbol.

char *Describe(Context)

20.13 Class ViaSord

20.13.1 Description

This class allows designing the without mute symbol.

Father class

Sordina

20.13.2 Children classes

NONE.

20.13.3 Public methods

ViaSord()

Initialiser: it sets the without mute symbol above or below the figure and defines the class identifier.

VUnit GetVU2Up()

It returns the distance from the centre of the without mute symbol to the extreme superior point of the symbol itself.

VUnit GetVU2Dwn()

It returns the distance from the centre of the without mute symbol to the extreme inferior point of the symbol itself.

VUnit GetVU2Rgt()

It returns the distance from the centre of the without mute symbol to the extreme right point of the symbol itself.

VUnit GetVU2Lft()

It returns the distance from the centre of the without mute symbol to the extreme left point of the symbol itself.

void Draw()

It draws the without mute symbol.

21 INT module

The classes belonging to the INT module model the musical symbols that are extended on sequences of notes. These symbols, called by means of the neologism *interval symbols*, correspond to executive prescriptions of different kinds (e.g. the crescendo and diminuendo symbols are dynamics signs whereas the phrase mark is an indication of expression); nevertheless all the corresponding classes descend from the unique class *IntEsteso* (abbreviation of “intervallo esteso”, extended interval). The setting of this hierarchy has taken into account much more the graphic behaviour of the symbols than their meaning.
The classes that belong to this module are:

- **Intervallo Esteso** that specialises in:
  1. **Forcella** to which two symbols correspond: *Crescendo* and *Diminuendo*,
  2. **Freccia,**
  3. **Cambio Ritornello,**
  4. **Onda,**
  5. **Legatura Quadra.**
- **Intervallo** that specialises in:
  1. **Legatura,**
  2. **Legatura di Valore,**
  3. **Modifica Ottava.**

21.1 How to use IntRefs

To use the references to horizantal symbols provided in the figures, you have to:

1. be sure that all the horiz. symbols referenced are still "alive", this can be done calling Partitura::SetupIntRefs() that explore all the scores building the references array of the figures of the scores. When the score is loaded this method is called, but no consistency is manteined if horiz. symbols are added or deleted. So call SetupIntRefs to rebuild the references, to optimize in future will be added a flag meaning that a horiz. symbol has been added or removed and the references will be rebuilt only in this case.
2. call the Figura::GetNIntRefs() and Figura::GetIntRef(int n) to scan all the horizontal symbols "connected" with the figure. For example:

```
Figure *pFig;
…
for(i=0; i<pFig->GetNIntRefs();i++)
{
  IntRef *ref=pFig->GetIntRef(i);
  switch(ref->intRef->GetID())
  {
    case CL_LEGATURA: // a slur
    case CL_LEGQUADRA: // a tuple
      switch(ref->type)
      {
        case INT_START:
          // a slur or a tuple is starting from pFig
          …
          break;
        case INT_OVER:
          // a slur or a tuple is "over" pFig
          …
          break;
        case INT_END:
          // a slur or a tuple is ending from pFig
          …
          break;
      }
      break;
  }
}
```

Notes

1. In the references of a figure are reported only the horiz. symbols that are starting/ending on figures of the same layer.
2. The term "over" does not mean that the horiz. symbol is graphically drawn over the figure, it means that a preceeding figure in the same layer has started a horiz. symbol that is ended on a following figure in the same layer.
3. The order of the references of the same type in the figure, is the order in which horiz. symbols have been added.

21.2 Class IntEsteso

21.2.1 Description

The main feature of an interval symbol is that it starts on a figure and it ends on another one. Therefore the IntEsteso class has as fundamental attributes two pointers to generic figures. The name “extended interval” has been introduced to differentiate the symbols that begins and ends on generic figures from those that admit as extremes only notes or chords, that have been named “interval symbols”. In selection phase we permit as extreme the figure of a beam but not a note of a chord. The selection of notes is typically performed by selecting two objects: the exception is the Legatura di Valore for which the selection of a single figure is enough. The class manages a list of “Segmenti di Intervalli” (intervals' segments), each one with its own graphic attributes for the positioning on the screen, height and width, that allows breaking an interval, when it is too long to be extended on a unique staff, and designing it in different segments (it is a problem that concerns DLIOO).

21.2.2 Father class

DrawObject

21.2.3 Children classes

Intervallo, Forcella, Onda, Freccia, CambioRit, LegatQuadra

21.2.4 Connected Types

```
typedef struct Segment {
    Point AbsP;
    VUnit Wdt, Hgt;
    Segment *pNext;
    int nGamboUp;
    int nGamboDwn;
    int nFigure;
} IntEstSegment;
```

IntEstSegment:

It implements a list of segments of Intervallo Esteso. It is composed by the following attributes:
Point AbsP Position of the interval segment.
VUnit Wdt Width of the interval segment.
VUnit Hgt Height of the interval segment.
Segment *pNext Pointer to the successive interval segment.
int nGamboUp;
int nGamboDwn;
int nFigure;

21.2.5 Protected attributes

Figura *PFigStart, *PFigEnd
Pointers to the figure on which the symbol starts and to the figure on which it ends.

Bool Sopra
It is TRUE if the symbol must be placed over the staff.

Bool Remove
It is used to manage the Started lists of ListaIntEstesi with priority determined by the order.

IntEstSegment FirstSegm
It identifies the top of the list of segments of the extended interval.

IntEstSegment* PFirstSegm

It is initialised at NULL (it indicates non-positioning), successively it indicates the top of the list.

IntEstSegment* PActSegm

It shows the segment that we are currently positioning.

NumCode NumericCode

Numeric Code of the interval.

Battuta *pbat1, *pbat2

Pointers to measures.

Bool Inizio

Tells if the interval starts in the current page.

Bool Fine

Tells if the interval ends in the current page.

int voice

Layer where the interval is.

Bool Multiv

Tells if the measure is polyphonic.

Bool StemInt

Bool Auto

Tells if the measure is polyphonic.

int StemFirst

1 = up, 0 = down for the first note of interval.

int StemLast

1 = up, 0 = down for the last note of interval.

Bool Interm

21.2.6 Public methods

IntEsteso()

Class constructor.

~ IntEsteso()

Class destroyer.

void SetSopra(Bool b)

It sets *Sopra = b*.

void GetSopra()

It returns *Sopra*.

void SetRemove(Bool b)

It sets *Remove=b*.

void GetRemove()

It returns *Remove.*

void SetNumericCode(NumCode nc)

It sets *Numeric Code= nc*.

NumCode GetNumericCode()

It returns *NumericCode*.

void SetPFigStart(Figura *pF)

It sets the starting figure of the symbol.

void SetPFigEnd(Figura *pF)

It sets the ending figure of the symbol.

Figura *GetPFigStart()

It returns *PFigStart*.

Figura *GetPfigEnd()

It returns *PFigEnd.*

v**oid AddSegm()**

It adds a segment at the end of the list of the interval segments.

v**oid DeallocaSegm()**

It deallocates the whole list of segments.

IntEstSegm* GetPFirstSegm()

It returns *PFirstSegm*.

v**oid ResetPosSegm()**

It places PActSegm at the top of the list.

v**oid GoNextActiveSegm()**

It moves forward of a position PActSegm.

virtual Bool TestAndSet(Figura *pFStart, Figura *pFEnd, Bool sopra)

It controls if the kind of figure corresponding to the pointers is correct with respect to the kind of interval symbol: if the two pointers coincide or one of the two points to a beam, the method returns FALSE; in the opposite case the method sets the pointer attributes at *Sopra* and returns TRUE.

Point GetAbsPos()

It returns the absolute position of the interval segment currently active.

v**oid SetAbsPosX(VUnit x)**

It sets the component x of the absolute position of the interval segment currently active.

v**oid SetAbsPosY(VUnit y)**

It sets the component y of the absolute position of the interval segment currently active.

v**oid SetWidth (VUnit w)**

It sets the width of the interval segment currently active.

v**oid SetHeight(VUnit h)**

It sets the height of the interval segment currently active.

virtual v**oid SetDim(VUnit w)**

It sets the dimensions (height and width) of the interval segment currently active.

VUnit GetVU2Up()

It returns the height of the interval segment currently active.

VUnit GetVU2Dwn()

It returns always 0.

VUnit GetVU2Lft()

It returns always 0.

VUnit GetVU2Rgt()

It returns the width of the interval segment currently active.

void Draw()

It recalls the DrawSegm for each segment that compose the interval.

virtual void DrawSegm(IntestSegment *)

It draws the segment passed as parameter.

Bool Hit(const Point&, SelObj, DrawObject*&, SymPath&)

See the general documentation of the method in section 1.18.2.

Bool SymbolicHit(SymPath&, SelObj, DrawObject*&)

It decodes the path symPath (see paragraph 1.18.2)

void SetNumBat (short, NumCode)

Deprecated.

NumCode GetNumBat (short)

Deprecated.

void SetBattuta (short n, Battuta *pb)

Set to parameter pbat1 or pbat2 depending on parameter n.

Battuta *GetBattuta (short n)

Returns pbat1 or pbat2 depending on parameter n.

Bool GetInizio ()

Returns parameter Inizio.

void SetInizio (Bool)

Set parameter Inizio.

Bool GetFine ()

Returns parameter Fine.

void SetFine (Bool)

Set parameter Fine.

void SetVoice (int a)

Set parameter voice.

int GetVoice ()

Returns parameter voice.

void SetMultiVoice (Bool a)

Set parameter Multiv.

Bool GetMultiVoice ()

Returns parameter Multiv.

void SetStemInt (int a)
Set parameter StemInt.
Bool GetStemInt ()
Returns parameter StemInt.
void SetAuto (int a)
Set parameter Auto.
Bool GetAuto
Returns parameter Auto.
void SetStemFirst (int a)
Set parameter StemFirst.
int GetStemFirst ()
Returns parameter StemFirst.
void SetStemLast (int a)
Set parameter StemLast.
int GetStemLast ()
Returns StemLast.
void SetIntermed (int a)
Set parameter Interm.
Bool GetIntermed ()
Returns parameter Interm.
void print ()
Printing method
virtual void AdjustVUFig ()
Adjust the symbol.
virtual char *Describe (Context)
Returns description of symbol in MDS.
virtual char *DescribeNet (Context, NumCode)
Returns description of the symbol.
virtual void ValutaLegature ()
Implemented in class Legatura, evaluate slurs.
void UpdateGambiInterni (Figura *pF)
Update stem up or down for the figure inside the slurs.
void void printSegm (IntEstSegment *)
Print the segment passed as parameter.

21.3 Class Forcella

21.3.1 Description

This class permits the representation of the forcella symbol in its two possible forms:
< shows *crescendo*
> shows *diminuendo*

21.3.2 Father Class

IntEsteso

21.3.3 Protected Attributes

Bool Crescendo
If it is TRUE, forcella indicates *crescendo*; if FALSE, *diminuendo*.

21.3.4 Public methods

Forcella()
Constructor of the class.
void SetCrescendo(Bool cr)
It puts *Crescendo* = *cr*.
Bool GetCrescendo()
It returns *Crescendo*.
void SetDim(VUnit w)
It sets the width and the height of the symbol.
void DrawSegm(IntEstSegment*)
It draws the symbol segment passed as parameter.
char *Describe (Context)
Returns description of symbol in MDS.

char *Describe (Context, NumCode)

Returns description of symbol in MDS.

21.4 Class CambioRit

21.4.1 Description

This class is used to design the symbol that changes the refrain, constituted by a number, a dot and an horizontal line.

21.4.2 Father Class

IntEsteso

21.4.3 Private Attributes

Text Testo

Attribute of the Text kind.

21.4.4 Public methods

CambioRit()

Constructor of the class.

void SetText(unsigned char ch)

It sets the Text attribute at the ch value.

unsigned char GetText()

It returns the value contained in Testo (1 character).

void SetDim(VUnit w)

It sets the width and the height of the symbol.

void DrawSegm(IntEstSegment*)

It draws the symbol segment passed as parameter.

void printSegm(IntEstSegment*)

It prints the symbol segment passed as parameter.

virtual void SetColor(int col)

Sets the color passed as parameter.

char *Describe (Context)

Returns description of symbol in MDS.

char *Describe (Context, NumCode)

Returns description of symbol in MDS.

21.5 Class Freccia

21.5.1 Description

This class is used to represent the arrow graphic object. Such symbol has different dimensions according to the number of note which it refers to.

21.5.2 Father Class

IntEsteso

21.5.3 Public methods

Freccia()

Constructor of the class.

void SetDim(VUnit w)

It sets the width and the height of the symbol.

void DrawSegm(IntEstSegment*)

It draws the symbol segment passed as parameter.

void printSegm(IntEstSegment*)

It prints the symbol segment passed as parameter.

char *Describe (Context)

Returns description of symbol in MDS.

char *Describe (Context, NumCode)

Returns description of symbol in MDS.

21.6 Class LegatQuadra

21.6.1 Description

This class is used to represent the squared slur that includes also a number.

21.6.2 Father Class

IntEsteso

21.6.3 Private Attributes

TNumerico TNum

Attribute of a TNumerico type due to the relation IS_PART_OF with the TNumrico class.

Bool SenzaLinea

Attribute to specify if line does not have to be drawn.

21.6.4 Public methods

LegatQuadra()

Constructor of the class.

~LegatQuadra()

Destroyer of the class.

Bool SetNumero(int n)

It sets the number of the attribute TNum at n.

int GetNumero()

It returns the number set in the TNum attribute.

void SetDim(VUnit w)

It sets the width and the height of the symbol.

void DrawSegm(IntEstSegment*)

It draws the segment of symbol passed as parameter.

Bool GetSenzaLinea ()

Returns parameter SenzaLinea.

void SetSenzaLinea (Bool)

Set parameter SenzaLinea.

void printSegm(IntEstSegment*)

It prints the segment of symbol passed as parameter.

virtual void SetColor(int col)

Sets the color passed as parameter.

char *Describe (Context)

Returns description of symbol in MDS.

char *Describe (Context, NumCode)

Returns description of symbol in MDS.

void ValutaLegature ()

Evaluate slurs.

float GetDurataTerzina (float terz, float den)

Returns duration of irregular group.

21.7 Class Onda

21.7.1 Description

This class is used to represent the wave symbol that can be placed above or below a sequence of notes. According to the number of notes it will be longer or shorter.

21.7.2 Father Class

IntEsteso

21.7.3 Public methods

Onda()

Constructor of the class.

void SetDim(VUnit w)

It sets the width and the height of the symbol.

void DrawSegm(IntEstSegment*)

It draws the segment of symbol passed as parameter.

void printSegm(IntEstSegment*)

It prints the segment of symbol passed as parameter.

char *Describe (Context)

Returns description of symbol in MDS.

char *Describe (Context, NumCode)

Returns description of symbol in MDS.

21.8 lass Intervallo

21.8.1 Description

This class represents an immediate specialisation of *IntEsteso*. The only difference consists in the fact that the object *Intervallo* admits as extremes only notes and chords (and not notes inside the chord): the only exception is Legatura di Valore that admits as extremes not chords but notes inside a chord.

21.8.2 Father Class

IntEsteso

21.8.3 Children classes

Legatura, ModifOttava

21.8.4 Public methods

Bool TestAndSet(Figura *pFStart, Figura *pFEnd, Bool sopra)

It controls if the proposed pointers correspond to notes or chords. In the assertive case, it sets the pointers and *Sopra* at the indicated values and returns TRUE. Otherwise, or if the pointers coincide, does not act and returns FALSE.

21.9 Class Legatura

21.9.1 Description

This class permits the representation of the slur that consists in a mid-ellipsis, oriented upwards or downwards. We decided to link the orientation of the slur to the *Sopra* attribute that the class inherits from IntEsteso: if the slur is drawn above the musical line is always represented as a superior mid-ellipsis; if it is under the line, is always represented as an inferior mid-ellipsis.

21.9.2 Father Class

Intervallo

21.9.3 Children classes

legatValore

21.9.4 Private Attributes

Bool InitSlur (Slur& slur, IntEstSegment *p1)

Initialises the slur for the interval segment passed as parameter.

21.9.5 Public Methods

Legatura()

Constructor of the class.

void SetDim(VUnit w)

It sets the width and the height of the symbol.

void DrawSegm(IntEstSegment*)

It draws the segment of symbol passed as parameter.

void printSegm(IntEstSegment*)

It prints the segment of symbol passed as parameter.

char *Describe (Context)

Returns description of symbol in MDS.

char *Describe (Context, NumCode)

Returns description of symbol in MDS.

void AdjustVUFig ()

Prepares symbols to be drawn.

void ValutaLegature ()

Evaluate slurs.

Bool Hit (const Point& p, SelObj selObject, DrawObject*& drwObj, SymPath& symPath)

Handle hit with the mouse on screen, returns TRUE if success.

21.10 Class ModifOttava

21.10.1 Description

This class is used to represent the musical symbol of the octave up.

21.10.2 Father Class

Intervallo

21.10.3 Public methods

ModifOttava()
Constructor of the class.
void SetDim(VUnit w)
It sets the width and the height of the symbol.
void DrawSegm(IntEstSegment*)
It draws the segment of symbol passed as parameter.
Bool TestAndSet(Figura *pFStart, Figura *pFEnd, Bool sopra)
It controls if the proposed pointers correspond to notes or chords. In the assertive case, it sets the pointers and *Sopra* at the indicated values and returns TRUE. Otherwise, or if the pointers coincide, does not act and returns FALSE.
void printSegm(IntEstSegment*)
It prints the segment of symbol passed as parameter.
char *Describe (Context)
Returns description of symbol in MDS.
char *Describe (Context, NumCode)
Returns description of symbol in MDS.

21.11 Class LegaturaValore

21.11.1 Description

This class is used to represent the musical symbol of tie.

21.11.2 Father Class

Legatura

21.11.3 Public methods

LegaturaValore()
Constructor of the class.
~LegaturaValore()
Destroyer of the class.
char *Describe (Context)
Returns description of symbol in MDS.
Bool TestAndSet(Figura *pFStart, Figura *pFEnd, Bool sopra)
It controls if the proposed pointers correspond to notes or chords. In the assertive case, it sets the pointers and *Sopra* at the indicated values and returns TRUE. Otherwise, or if the pointers coincide, does not act and returns FALSE.

22 PAR module

The PAR module contains the classes related to the management of a complete main score.

22.1 Class ListaSpartiti

22.1.1 Description

This class has the task of the management of the scores inside a main score. Its main functionalities are:

- Alignment of the measures and setting of the bar lines that constitute the main score.
- Management of the brace brackets.

Each score inserted in the list is endowed with a unequivocal numeric code (the type NumCode is defined as short).

22.1.2 Father Class

Lista

22.1.3 Protected Attributes

NumCode CodeCounter

Counter for assigning numeric codes to the elements of the list.

Rectangle MusicRect

The rectangle that identifies the space on the screen reserved to the main score.

22.1.4 Public Methods

ListaSpartiti()

It performs *Init.*

~ListaSpartiti()

It performs *Free.*

void Init()

It initialises the list (and *CodeCounter*) with one score.

void Free()

It deallocates the list.

void SetMusicRect(const Rectangle& r)

It sets the rectangle *MusicRect* equal to *r*.

Rectangle GetMusicRect()

It returns *MusicRect.*

Spartito *GetSpart(NumCode nc)

It returns the pointer to the score in position nc.

void PlaceSpartiti(int LeftMargin, int TopMargin, Vunit& width)

It places the scores of the main score, included staffs and brace brackets. It does not performs the alignment of measures. The scores are positioned from the top to the bottom in the order of the list.

void AlignBattute()

It places the measures that appear in the current page. In order to work correctly, each part must present the same number of measures and the measures belonging to the same column must have the same number of figures in each layer. The method determines the number of measures that can enter in the current page: it does not align on the right the measures.

void PlaceIntervalli(Bool onlyLegature=TRUE)

It disposes the interval symbols in the score list.

void GoBatt(NumCode nb)

It carries all the scores to the measure #nb.

Bool GoForward()

It moves forward of a measure.

Bool GoBackward()

It moves backward of a measure.

Bool GoNextPage()

It moves forward of a page. It returns FALSE if it is the last page, otherwise it returns TRUE.

Bool GoPrevPage()

It moves backward of a page. It returns FALSE if it is the first page, otherwise it returns TRUE.

void DrawPage()

It draws the current page.

void RedrawPage()

It recalculates and redraws the page.

void DrawTagli(const Point&)

It draws the leger lines needed to arrive to the indicated point.

Bool Hit(const Point& p)

See general documentation in section 1.18.2.

Bool Hit(const Point&,SelObj,DrawObject*&,SymPath&)

See general documentation in section 1.18.2.

Bool Hit(const Rectangle&,SelObj,short layer,SymPath&,SymPath&)

See general documentation in section 1.18.2.

Bool SymbolicHit(SymPath&,SelObj,DrawObject*&)

See section 1.18.2.

void Add(Spartito *pSp)

It adds a score at the bottom of the list. It reorganises the type of scores on the basis of the position that each one currently assumes in the main score.

void AddAfter(Spartito *pSp, Spartito *pSpRef)

It adds a new score after that passed as reference. It reorganises the type of scores on the basis of the position that each one currently assumes in the main score.

void Del(Spartito *pSp)

It deletes the score from the list. It reorganises the type of scores on the basis of the position that each one currently assumes in the main score.

Bool DeleteSym(DrawObject*)

It deletes the assigned symbol (that can be also a brace bracket).

int Save(FILE *)

It saves the scores on the file passed as parameter.

void Suona (MoodsMidiWrite *)

It generates the playable file via Midi interface.

void printPage()

It prints the page

void GiustificaBattute (Battuta, TipoGiust, double kGiust)**

It calls justification method on the measure of the part.

void GiustificaDaA (int daBat, int aBat, TipoGiust, double kGiust)

It calls justification method on a range of measures of the part.

int GetNumeroBattute()

It returns number of measure.

int AlignBattuteLineBreak()

This method is used for line breaking.

void StretchBattute (Battuta vbat, UL spacecol)**

This method stretch the measures to reach a specific target in spacing.

UL JustVerify (Battuta vbat, double kGiust)**

Check the final spacing of measure with a given tuning parameter.

Bool IsLastPage ()

TRUE if last page.

void SetupIntRefs();

It sets for all the figures of all the scores the references to the horiz. symbols starting/ending/over the figure.

22.2 Class Partitura

22.2.1 Description

This class represents the musical main score, and contains the list of scores (in this case the most appropriate name is not score but *part*).

Besides the LiooWin also this class is used as connection between the generic part of the user interface and the musical classes.
Its main aim is to acquire the external events and to translate them in calls to methods of the musical classes (Hit and SymCommand methods).
Besides this it provides the functionalities for:

- Setting the internal status in order to interpret correctly the external events and perform the commands.
- Designing the page of the score.
- Changing page.
- Changing the number of staffs on which the measures are to be drawn.
- Inabling/disabling the design of the leger lines to the mouse.

The executing commands concerns both the MASE/MASAE and the DLIOOs lecterns: as a matter of fact the execution of the command foresees its representation in symbolic form through a structure comprehending the coding of the type of command, the type of objects involved, the arguments and the addresses (in symbolic form) of the objects (see section 1.18), such structure is available for a transmission via web and for its execution by the remote lectern.

At the actual state four types of settings for the commands exist.

- Immediate Execution of the command (DoCommandImmediate): the execution of this kind of commands does not requires the selection of objects in the main score.
- Execution through the selection of an object (DoCommand): it is used for the insertion of figures, ornaments, etc. The execution of the command is performed by pressing on the left button of the mouse.
- Execution through the selection of two objects (DoCommand): it is used for the insertion of intervals (slurs, change of refrain, etc.). The execution of the command is performed by pressing on the left button of the mouse.
- Execution through the selection with Rectangle (DoCommandRectSel): the execution of the command is performed by releasing the left button of the mouse.

22.2.2 Father class

None

22.2.3 Connected types

enum StateImpag

It shows the current status of the paging. It can assume the following values:
BEGIN_IMP The paging has been activated.
END_BATT The delivering of the measures from MASAE to DLIOO.
END_IMP The paging has been ended.

22.2.4 Protected Attributes

ListaSpartiti listaSpa
List containing the parts of the main score.

SymCmd UsrSymCmd
Command in symbolic form set by the user. The type SymCmd is defined in section 1.19.4.

SymCmd NetSymCmd
Command in symbolic form transmitted (or received) via web.

char fileName[FILE_NAME_LEN]
Name of the file of loading/saving.

Lista listaSel
List of selected symbols.

Bool showTagli
If TRUE it visualises the leger lines of the notes.

StateImpag Impaginazione
It shows the current status of the paging execution of the musical piece.

SaveType SaveMethod

ListaParentesi listaPar
List containing the parenthesis of partitura.

char fileLoadName[FILE_NAME_LEN];
char fileLoadPath[FILE_NAME_LEN];
char fileSaveName[FILE_NAME_LEN];
char fileSavePath[FILE_NAME_LEN];

float LeftMargin, TopMargin, BottomMargin;
Right margin and left margin of main window principale

float DimSpazio, DimFig;
default dimension of space and figure (deprecated)

float Scala;

Bool FirstSel;

used to manage commands in TwoSel

long* ETT;
NumCode ETTLenght;

ListaEtichette *listaEti;

StateExecution StatoEsecuzione;

state variable to control in which execution step we are: paging (impaginazione), init exec, execution

22.2.5 Protected methods

void DeselectAll()

It deselects all the objects that have been selected.

void SelectAll()

It selects all the objects that have been gathered.

void ResetPath(SymCmd& sc)

It turns to zero all data of Path1 and Path2, contained in sc.

Bool InsertSel(const Point& p)

Through the scanning of the scores, it performs the selection via insertion:

1. It acquires an object through the Hit method.
2. It puts on the first element of the array a reference to the *p* point.
3. It performs the command activated on the selected object.

void Suona (MoodsMidiWrite *)

Creates file Midi.

22.2.6 Private methods

Bool SingleSel(const Point& p)

The method performs the selection of a single object:

1. It acquires an object through the Hit method.
2. It selects the object and adds it to the list *listaSel.*
3. If an other object is already present in the list it deselects and deletes it from the list.

At the end of the selection it provides recalling the appropriate methods for performing the command set on the selected object.

4.

Bool ImmediateSel(const Point& p)

If the command involves all the scores, it recalls the appropriate method for its execution, otherwise it performs the selection for the immediate execution of the command:

1. It acquires an object through the Hit method.
2. It performs the activated command on the selected object.

Bool TwoSel(Point p)

Double selection for the interval insertion.

Bool DoCommandBarra(const Point& p)

It sets the bar line of a measure for all the scores.

Bool DoCommandDelBat(const Point& p)

It deletes the measure individuated by p.

virtual Bool DoCommandGruppo()

It manages the execution of the commands related to the beams. It is used in order to pass from beams to single notes and vice versa.

Bool DoCommandDelColFig(const Point& p)

22.2.7 Public Methods

Partitura()

It performs nothing.

virtual ~ Partitura()

It performs nothing.

float GetLeftMargin();
float GetTopMargin();
float GetBottomMargin();
void SetLeftMargin(float);
void SetTopMargin(float);
void SetBottomMargin(float);
void SetDimSpazio(float);
void SetDimFig(float);
void SetScala(float);

float GetDimSpazio();
float GetDimFig();
float GetScala();

virtual void KillLioo();

void SetImpaginazione(StateImpag)

It sets the status of the attribute of paging.

Bool GetImpaginazione()

It returns FALSE if the Impaginazione attribute is END_IMP.

Bool GetBeginImpag()

It returns TRUE if the Imapginazione attribute is BEGIN_IMP.

Bool GetFirstEndBattImp()

It returns the state of paging: end batt imp the first time.

Bool GetEsecuzione();
Bool GetBeginEsec();
Bool DoCommandAddBattuta(const Point& p);

StateExecution GetStatoEsecuzione();
void SetStatoEsecuzione(StateExecution);

void BeginPaint()

If *showTagli* is active, it deletes the leger lines to the mouse arrow.

void EndPaint()

If *showTagli* is active, it puts the leger lines to the mouse arrow.

void ShowTagli()

It inables the design of the leger lines of the mouse arrow.

Bool HideTagli()

It disables the design of the leger lines of the mouse arrow.

Bool GetShowTagli()

It returns TRUE if the leger lines were active.

virtual void DrawTagli(const Point&)

It draws the necessary leger lines to arrive to the indicated point.

virtual void New()

It initialises the list of scores and the other attributes.

virtual void Free();
virtual void Init();

void SetMusicRect(const Rectangle& r)

It sets at *r* the rectangle of screen dedicated to the main score (it acts on the attribute *MusicRect* of ListaSpartiti)

virtual void GoForward()

It moves forwards of a measure.

virtual void GoBackward()

It moves backwards of a measure.

virtual Bool GoNextPage()

It draws the next page on the auxiliary pixmap (or window). It returns FALSE if we are on the last page otherwise it returns TRUE.

virtual void Draw()

It draws the current page of the main score.

virtual void Redraw()

It redraws the current page recalculating the positions of the scores and the alignments.

virtual void GoTop()

It is placed at the beginning of the main score and it designs the second page in the area reserved to the scrolling (window or pixmap).

virtual void AddSpartito(Spartito *pSp)

It adds a score on the bottom of the list (i.e. down in the main score).

void AddNewSpartito(Spartito *pSpRef, Bool sopra, int nStaff=1, char* name=NULL)

It creates and adds a score compatible with those already present, that is to say with the same number of measures and for each measure the same number of figures. It places the new score above or below the score passed for reference.

The new score has the specified number of staffs (nStaff) and the specified name.

Spartito *GetFirstSpa()

It returns the pointer to the first score of the list.

Spartito *GetNextSpa(Spartito *psp)

It returns the score of the list that follows that pointed from *psp*.

Spartito *GetPrevSpa(Spartito *psp)

It returns the score of the list that precedes that pointed from *psp*.

Spartito *GetSpartNum(int n)

It returns the score of the list that has position n.

Spartito *GetSpart(NumCode nc)

It returns the score of the list that has NumericCode equal to nc.

void SetCommand(CmdID cmdId, SelType stype=NO_SEL,SelObj sobj=ANY_SEL)
ArgType arg0=0, ArgType arg1=0, ArgType arg2=0, ArgType arg3=0,
ArgType arg4=0, char *txt=NULL, ArgType arg5=0, ArgType arg6=0,
ArgType arg7=0

It sets the command IsrSymCmd, the type of objects to select, the type of selection to be used and the arguments of the command.

void ResetCommand()

It resets UsrSymCmd in order not to perform the commands.

SymCmd *GetSymCmd()

It returns the pointer to UsrSymCmd.

void SetUsrSymCmd(SymCmd sc)

It sets UsrSymCmd at sc.

Bool Hit(const Point&p)

See general documentation in section 1.18.2.

Bool Hit(const Rectangle&,SelObj,short layer,SymPath&,SymPath&)

See general documentation in section 1.18.2.

Bool SymbolicHit(SymPath&,SelObj,DrawObject*&)

See section 1.18.2.

void DoCommandImmediate()

It executes the commands that do not need the selection of objects of the main score.

virtual Bool DoCommandRectSel(const Point& p)

It is recalled from the method LeftButtonRelease of LiooWindow. If the point is internal and is in selection RECT_SEL it requires to the mouse the rectangle indicated by the user and performs the selection through the Hit method with rectangle. If the selection is successful it performs the set command.

Bool DoCommandInterv(SymCmd sc)

It inserts an interval.

virtual Bool DoCommand(const Point& p)

The method LeftButtonPress of LiooWindow is recalled. According to the kind of selection set with SetCommand it recalls the appropriate method. It recalls, according to the kind of selection set, the appropriate method to manage the situation. In particular for the kind of selection RECT_SEL it activates the mouse for the selection through the rectangle and deselects the objects eventually selected.

virtual Bool DoNetSymCmd()

It executes NetSymCmd, that is to say the command transmitted through the web.

virtual CmdResult SymCommand(SymCmd&, Battuta *pb=NULL)

It performs the commands that concern the whole main score. See general documentation of the method SymCommand in section 1.19.

Bool DeleteSym(DrawObject *s)

It deletes the symbol pointed by *s*.

virtual void NetInvioComando(SymCmd sc)

It performs nothing. It is redefined and used by the children classes. MASAE recalls it every time that a command has been successfully executed, DLIOO each time that the user requires the execution of a command.

virtual void ReadFromNetwork()

The method is used to test periodically the buffer of the web (see NET module).

virtual void LoadPindex()

It loads the “Tabella delle Parti” (ONCM Table) (see NET module).

virtual void InitParte()

When a main score is loaded from a file, by means of this method the related parts are sent to the DLIOOs (see NET module).

virtual Bool GoPrevPage();

go back one measure

virtual void GoBatt(NumCode np);
void GoBattIfOut(NumCode nb);

int GetNumSpa(void) { return listaSpa.GetNumObj(); };

void SetSaveMethod(SaveType NewSaveMethod);
SaveType GetSaveMethod(void);

void ResetACL(void);

Bool CopiaFileMDS2HID(char * SourcePath,char *SourceName,char *DestPath,char *DestName);

void SetNetSymCmd(SymCmd sc);

sets UsrSymCmd

Bool CmdAddLettera(SymCmd & SC);

virtual int GetIDleader(char* parte);
virtual void SetAbilCompress(char* parte, Bool abil);
virtual void SetAbilCompressAll(Bool abil);
virtual Bool GetAbil_Compress(int i);

Bool LoadS(char*);

load from string passed via network

void Save();

save on file

void Print();

print on file

Bool ImportMidi(char *fName,int trc);

load from midi file

Bool ExportMidi(char *fName,int trc);

generates midi file

Bool DelSpartito(SymCmd SCmd);
void SetTimeExec(Battuta *pbt);

passes the position of the measure inserts its duration in execution
id is 1 ... NumeroBattute

void SetTimeExec(int pos, long time);

long GetTimeExec(int nc);

passes the position of the measure returns its duration in execution
nc is 1 ... NumeroBattute

void ReadETT(char *filename);

reads file .ETT (with extension) and inserts data in the ETT array after creating it.
deletes any preceding ETT array

void SaveETT(char* filename);

void BuildETT(NumCode num_batt);

build ETT array of dimension num_batt

virtual void SincrBatt(NumCode);
virtual void EseguiDebug(void*,void*,void*);

virtual void InitEsecuzione(char*);
virtual void JumpToBatt(NumCode&) {};
virtual void InitImpaginazione();
virtual void RichPageForward() {};

void AddEtichetta(char,NumCode);
char GetEtichetta(int n);

returns the label of #n position

Bool FindEtichetta(char);

NumCode GetBattEtichetta(char);

returns the progressive number of the measure related to the label passed as parameter

int GetNumEtichette();

int GetNumeroBattute();

int CalcLivello(Spartito *ps1, Spartito *ps2);

virtual void EditingGoBottom();
virtual void EditingJumpBatt(int nbat);
virtual void EditingGoTop();
virtual void RequestJump();
virtual void GestLabel();
virtual void NetPiu5();
virtual void NetMeno5();

virtual void GoBottom();

positions at the end of partitura

Bool OkInitEsec();
Bool OkEsecuzione();
Bool OkJump();
Bool OkCompila();

void AddParentesiGraffa(NumCode ns, NumCode ne, NumCode nc);
void AddParentesiQuadra(NumCode ns, NumCode ne, NumCode nc);

void CalcLivelli();
void CalcProlBarre();

void ScoreJustify();

Bool IsLastPage();

void SetupIntRefs();
It sets for all the figures of all the scores the references to the horiz. symbols starting/ending/over the figure.

22.3 Class Parte

22.3.1 Description

This class represents the musical part of DLIOO and contains the score with one or more musical lines (for the moment the implementation consists of a unique score with one staff that foresees the carriage return). It will have to be able to represent the parts on the lecterns allowing the disposition of more than one line of main score for each orchestral (e.g., the piano needs two staffs) in a unique page differently from the current MASAE that can visualise only one per score.

22.3.2 Father class

Partitura

22.3.3 Children classes

LoMas

22.3.4 Public methods

Parte()
It is the constructor of the Parte. It uses the "new()" method of Spartito through a pointer from ListaSpartiti in order to initialise the first element of the list (for the moment the list is constituted from a unique element).

void New()
It initialises the list of scores and the other attributes.

void SetNPentagrammi(int n5)
It sets the number of staffs for page. It needs to have MusicRect set.

void GoForward()
It moves forward of a measure.

void GoBackward()
It moves backward of a measure.

Bool GoNextPage()
It draws the successive page on the auxiliary pixmap (or window). It returns FALSE if we are on the last page otherwise it returns TRUE.

Bool GoPrevPage()
It draws the previous page on the auxiliary pixmap (or window). It returns FALSE if we are on the first page otherwise it returns TRUE.

virtual void AddSpartito(Spartito *pSp)
It adds a score on the bottom of the scores' list of Parte.

void DrawTagli(const Point&)
It draws the additional line to reach the indicated point.

void Draw()
It draws the score.

void Redraw()
It redisposes Parte's symbols and redraws the page.

void InitParte()
When we load a main score from a file, thanks to this method we send the related parts to the DLIOOs (see NET module).

Bool ImmediateSel(const Point& p)
If the command involves all the scores, it recalls the appropriate method for its execution, otherwise it performs the selection for the immediate execution of the command:

1. It acquires an object by means of the Hit method.
2. It performs the activated command on the selected object.

Bool DoCommandGruppo()

It manages the execution of the commands related to the beams. It is used to pass from beams to single notes and vice versa.

CmdResult SymCommand(SymCmd)

It performs commands that concern the whole main score. See general documentation of the SymCommand in section 1.19.

Bool DoNetSymCmd()

It performs NetSymCmd, thus performs a command arrived from the web.

23 SCA module

The SCA module contains the classes that implement symbols referred to Battuta that have to be written outside the measure itself; most of them are indications concerning the beating of time.

23.1 Class NumBattuta

23.1.1 Description

NumBattuta is a simple specialisation of TNumerico that automatically sets the necessary font for its representation.

23.1.2 Father class

TNumerico

23.1.3 Children classes

NONE.

23.1.4 Public methods

NumBattuta()

Constructor that recalls the TNumerico constructor (and thus that of Text).

NumBattuta(int n)

Constructor that recalls the TNumerico constructor (and thus that of Text), thus sets the text corresponding to the number *n*.

char *Describe(Context)

23.2 Class NumGrande

23.2.1 Description

This is the class used for the representation of a number above the staff in order to indicate the quantity of empty measures and the progressive number of a set of equal measures. The difference between the two representations consists in the dimensions of the character, for this reason from the NumGrande class two sub-classes have been derived: NumPausa and NumUguale. The symbol is managed by the measure thus NumGrande class is in relationship IS_REFERRED_BY with Battuta.

23.2.2 Father class

TNumerico

23.2.3 Children classes

NumPausa, NumUguale

23.2.4 Public methods

NumGrande()

Class constructor that recalls the constructor of the class TNumerico.

Bool SetTxt(unsigned char *s)

It sets the text attribute (it allocates the necessary memory and recalls the homonym method of Text). If everything is alright it returns TRUE.

23.3 Class NumUguale

23.3.1 Description

It is the first class derived from NumGrande and it draws the numbers on the equal measures. The symbol is centred in the space occupied by the measure.

23.3.2 Father class

NumGrande

23.3.3 Children classes

NONE.

23.3.4 Public Methods

NumUguale()

Class constructor that recalls the constructor of the NumGrande class, assigns the class identifier and sets the font of the text.

23.4 Class NumPausa

23.4.1 Description

It is the second class derived from NumGrande and it draws the quantity of empty measures defined by the corresponding rest. It represents the number that is positioned above the space occupied by the measure.

23.4.2 Father class

NumGrande

23.4.3 Children classes

NONE.

23.4.4 Public Methods

NumPausa()

Class constructor that recalls the constructor of the NumGrande class, assigns the class identifier and sets the font of the text of big dimension (nBigTextFon).

23.5 Class Lettera

23.5.1 Description

The instances of the class Lettera are used for identifying particular points of the main score, in order to give to the orchestral a more precise reference. This class, derived from Text, allows the representation of both alphabetical letters and numbers, representing them with a big font (nBigTextFon). The symbol is placed above the measure and the relationship between the class Battuta and the class Lettera is of the IS_REFERRED_BY kind.

23.5.2 Father class

Text

23.5.3 Children classes

NONE

23.5.4 Public methods

Lettera()

Class constructor that recalls the constructor of the Text class, assigns the class identifier and sets the font of the text.

Bool SetTxt(unsigned char *s)

It sets the text attribute (it allocates the necessary memory and recalls the homonym method of Text). If everything is alright it returns TRUE.

char *Describe(Context)

23.6 Class TSalto

23.6.1 Description

The class TSalto is derived from the class Text and has a IS_REFERRED_BY relationship with the Battuta class. It represents the textual indications of repetition and makes available to the user textual characters and the two conventional signs of jump. The indications that it contains are written in correspondence of a bar line that signals the repetition of entire measures.

23.6.2 Father class

Text

23.6.3 Children classes

NONE

23.6.4 Public methods

TSalto()

Class constructor that recalls the constructor of the Text class, assigns the class identifier and sets the font of the text.

Bool SetTxt(unsigned char *s)

It sets the text attribute (it allocates the necessary memory and recalls the homonym method of Text). If everything is alright it returns TRUE.

char *Describe(Context)

23.7 Class Movimento

23.7.1 Description

It is the class that manages the agogic indications that are written above the staff and sets the general movement of the musical piece; the related symbols appear in the first measure that it manages and can indicate the titles of the movement (‘Allegro’, ‘Andante’). This class manages two sub-classes: TMovimento to manage the text and Metronomo for the associated metronomic indications. Movimento foresees a pointer to the Metronomo class (whose presence is not mandatory) and a IS_PART_OF relationship with TMovimento; it is besides in a IS_REFERRED_BY relationship with Battuta.

23.7.2 Father class

DrawObject

23.7.3 Children classes

NONE

23.7.4 Protected attributes

VUnit VU2Up()

Current dimension toward the top, with respect to the point AbsPos, of the text of Movimento with or without the presence of the symbols of the class Metronomo.

VUnit VU2Dwn()

Current dimension toward the bottom, with respect to the point AbsPos, of the text of Movimento with or without the presence of the symbols of the class Metronomo.

VUnit VU2Rgt()

Current dimension toward the right, with respect to the point AbsPos, of the text of Movimento with or without the presence of the symbols of the class Metronomo.

TMovimento TMov

Object of the TMovimento class that provides to the representation of the symbols of such class.

Metronomo *ptrMetr

It is a pointer to the Metronomo class.

Public methods

Movimento()

It is the creator of the class that assigns the class identifier, initialises the pointer ptrmetr at NULL and sets the text font.

Movimento()

Deallocates the eventual metronome connected to the class.

SetMetronomo()

It creates a pointer to the Metronomo.

DelMetronomo()

It destroys the pointer to the Metronomo.

SetPos(DrawObject *d,const Point &p)

It places the object related to TMovimento starting from the position AbsPos, updates the attributes VU2Rgt, VU2Up, VU2Dwn, verifies the presence of an object of the Metronomo kind and eventually it positions it updating, if necessary, the above described parameters.

Draw()

It draws the TMovimento symbol, verifies the presence of a symbol of the Metronomo kind and if it finds it, it represents it by recalling the method of such class.

GetVU2Rgt()

It returns the dimension of the TMovimento symbol on the right with respect to the position of AbsPos.

GetVU2Lft()

It always returns zero.

GetVU2Up()

It returns the dimension of the TMovimento symbol upward with respect to the position of AbsPos.

GetVU2Dwn()

It returns the dimension of the TMovimento symbol downward with respect to the position of AbsPos.

MovSetTxt(unsigned char *p, ClassID ident, Bool punt, int s)

It sets the string pointed by p of the Tmovimento symbol and if the Metronomo symbol is present it sets the type of small note (ident), the eventual associated augmentation dot (punt) and the number.

23.8 Class TMovimento

23.8.1 Description

This class represents the text that specifies the movement of the piece and is in a IS_A relationship with Text.

23.8.2 Father class

Text

23.8.3 Children classes

NONE

23.8.4 Public methods

TMovimento()

Class constructor that recalls the constructor of the Text class, assigns the class identifier and sets the font of the text (nTextFont).

Bool SetTxt(unsigned char *s)

It sets the text attribute (it allocates the necessary memory and recalls the homonym method of Text). If everything is alright it returns TRUE.

23.9 Class Metronomo

23.9.1 Description

This class provides a metronomic expression consisting in a small note, eventually pointed, and an integer number assigned to it. This all is included in an expression containing a equal sign and contained in a bracket. For this reason, the Metronomo class foresees IS_PART_OF relationships with Text, TNumerico and Nota and presents an IS_REFERRED_BY relationship with Movimento and an IS_A with DrawObject.

23.9.2 Father class

DrawObject

23.9.3 Children classes

NONE

23.9.4 Protected attributes

VUnit VU2Rgt()

Current dimension of the Metronomo symbol toward the right, with respect to the point AbsPos.

VUnit VU2Lft()

Current dimension of the Metronomo symbol toward the left, with respect to the point AbsPos.

VUnit VU2Up()

Current dimension of the Metronomo symbol toward the top, with respect to the point AbsPos.

VUnit VU2Dwn()

Current dimension of the Metronomo symbol toward the bottom, with respect to the point AbsPos.

Text TParAperta

It represents the open bracket.

Text TParChiusa

It represents the closed bracket.

Text TUguale

It represents the sign of equal.

Nota *ptrNota

It is a pointer to an object of the Nota type that will be used in the representation of the symbol.

TNumerico MetroTNum

It represents the number.

23.9.5 Public methods

Metronomo()

It is a constructor of the class that sets the class identifier and the font of all the parts of the text, it initialises a pointer to the zero note and it sets the values of open and closed bracket and of the equal sign.

SetTNNumero(int s)

It recalls the SetNumero(int n) method inherited by the TNumerico class.

SetNota(ClassID ident, Bool punt)

It sets the type of small note on the basis of the ident passed together with the eventual additional dot.

SetPos(DrawObject *drwobj,const Point& posizione)

It places the Metronomo setting the position of the different components with respect to AbsPos. It updates the horizontal dimensions taking into account the dimension of the different symbols and of the space among them. It compares the vertical dimensions of the different elements of Metronomo by setting the bigger ones.

GetVU2Up()

It returns the dimension of the Metronomo symbol upward with respect to the position of AbsPos.

GetVU2Dwn()

It returns the dimension of the Metronomo symbol downward with respect to the position of AbsPos.

GetVU2Rgt()

It returns the dimension of the Metronomo symbol on the right with respect to the position of AbsPos.

GetVU2Lft()

It returns the dimension of the Metronomo symbol on the left with respect to the position of AbsPos.

Draw()

It draws all the parts of the Metronomo by recalling the appropriate methods of the class.

23.10 Class Scansione

23.10.1 Description

This class allows setting the beating of time chosen by the director through vertical bar lines whose number can be selected by the user. It has a IS-REFERRED_BY relationship with Battuta.

23.10.2 Father class

DrawObject

23.10.3 Children classes

NONE

23.10.4 Protected attributes

int numscan

It represents the number of bar lines per time beating.

23.10.5 Public attributes

Scansione()

It is the creator of the class that sets the identifier and puts numscan at zero.

GetVU2Rgt()

It returns the dimension of the Scansione symbol on the right with respect to the position of AbsPos.

GetVU2Lft()

It always returns zero.

GetVU2Up()

It returns the dimension of the Scansione symbol upward with respect to the position of AbsPos.

GetVU2Dwn()

It returns the dimension of the Scansione symbol downward with respect to the position of AbsPos.

Draw()

It draws the bar lines.

SetNumScan(int nscans)

It sets the numscan attribute at nscans.

int GetNumScan()

It returns the value of the numscan attribute.

char *Describe(Context)

24 SPA module

This module contains the classes related to the management of the score.

- The **Pentagramma** class is used to draw the staffs on the screen.
- The **ListaBattute** class deals with the management of the measures, with their organisation on the screen and performs the scrolling of the pages.
- The **ListaIntEst** class deals with the management of the extended intervals.
- The **Spartito** class manages themeasures by means of the ListaBattute class and should provide their synchronisation with the interval symbols.
- The **ParGraffa** class permits connecting two adjacent staffs inside a main score.

24.1 Class Pentagramma

24.1.1 Description

This is a very simple class that provides to draw the staff on which the notes will be placed. Besides the area occupied from the staff an empty space is added above and below it, in which other notes can be placed and are still considered related to the staff.

24.1.2 Father Class

DrawObject

24.1.3 Protected Attributes

VUnit width

True width of the staff.

VUnit Max Width

Maximum width of the staff. It is set according to the width of the graphic screen; *width≤ Max Width* must be always valid.

VUnit spaceUpDwn

It measures an empty space above and below the staff.

24.1.4 Public Methods

Pentagramma()

It initialises the staff with null values.

Pentagramma(VUnit w)

It initialises the staff with *w* width.

void SetWidth(VUnit w)

It sets the staff width at *w*.

void SetMaxWidth(VUnit w)

It sets the maximum staff width at *w*.

VUnit GetMaxWidth()

It returns the maximum staff width.

void SetSpaceUpDwn(VUnit spc)

It sets the space to be reserved above and below the staff.

VUnit GetSpaceUpDwn()

It returns the space to be reserved above and below the staff.

VUnit GetVU2Rgt()

It returns *width*.

VUnit GetVU2Up()

It returns the height of the staff as well as the quantity of empty space above the staff (*spaceUpDwn*).

VUnit GetVU2Lft()

It returns 0.

VUnit GetVU2Dwn()

It returns the empty space below the staff (it takes *spaceUpDwn* – 1 to avoid overlapping with other scores).

void Draw()

It draws the staff by starting from the line 0 that has the co-ordinate of AbsPos, and then augmenting the y till the 5 lines are drawn.

void DrawTagli(const Point&p)

It draws the leger lines outside the staff that are necessary to reach the point *p*. The same algorithm for computing the height of the note on the basis of the co-ordinate of the point, adopted from the Battuta for inserting notes, is followed.
The design is performed by XOR method in order to permit the deletion through redrawing. In fact the method is used for drawing the leger lines outside the staff until the position of the mouse.

void print()

Printing method.

24.2 Class ListaBattute

24.2.1 Description

This class has the task of the management of measures. It inherits from the Lista class the features that are typical of a list. The specific functionalities that it offers concern mainly:

- The insertion/deletion of the measures.
- The placement and the design of the measures on the staffs of the page.
- The management of the numbers of the measures.
- The page scrolling of the measures.
- The selection of musical objects in the page.
- The deletion of musical objects inside the page.

Each measure that is added to the list is endowed with a numeric unequivocal code. To this end, after each new insertion, the counter *CodeCounter* is updated and is not decreased in case of deletions. For DLIOO the same value of *CodeCounter* as MASAE is used (see method *AddAfter*).

24.2.2 Father Class

Lista

24.2.3 Protected Attributes

NumCode CodeCounter

Counter to endow each measure with a numeric code.

NumCode PagCounter

Counter to number progressively each page of the musical piece.

Node *firstBatPage

Pointer to the node containing the reference to the first measure of the page.

Node *firstBatNextPage

Pointer to the node containing the reference to the first measure of the following page. If it has NULL value it means that the current page is the last one.

Node* CurrentNode

Pointer to the node corresponding to the measure that Partitura is positioning.

Intestazione* PrevIntest

Pointer to the heading of the last measure that Partitura has positioned.

24.2.4 Protected Methods

Node* CalcRigo(Node* n,Pentagramma* pnt,int rigo)

It calculates the positions of the measures, starting from the *n* node, on the line of the staff that finds itself on the line *rigo*. It returns the pointer to the node of the first measure of the successive line. If there are no other measures it returns NULL. The algorithm that is followed is:

1. The measures that have been set are scanned in order to occupy the less space as possible, taking into account the width of the staff that determines that the measure that surely does not enter in the staff is to be placed in the successive line.
2. The difference between the available space on the staff and the space occupied from the "restricted" measures is divided between the measures in order to set the correct value of distance between the figures to occupy the whole staff.
3. The measures are positioned on the staff.

The first time that the measures are set also the type of heading is set, because the first measure of each line must have the clef and key signature and the first measure of the page also the time. In addition also the changes of heading between a measure and the successive are considered.
A particular case occurs when a measure is wider than the staff, even when it is "restricted". In this case the measure is skipped because a fundamental hypothesis states that *a measure*

cannot be broken. To cope with this problem, in the insertion of figures through graphic interface a maximum limit to the number of figures per measure has been introduced.

24.2.5 Public methods

ListaBattute()

It initialises the pointers at the beginning of the page at NULL and calls the Init method.

~ ListaBattute()

Deallocates the list together with the measures through the Free method.

void PageSetUp(Pentagramma pent[],int nPent)

It sets the position of the measures for the whole page, starting from firstBatPage and determines the first measure of the following page. The array of the Pentagrammi for the page and the number of staffs usable are endowed.

void SetPagCounter(NumCode)

It sets *PagCounter* at the value passed as parameter.

NumCode GetPagCounter()

It returns the value *PagCounter.*

void SetBattPage(Battuta*)

It sets the measure passed as parameter of beginning of the page with the appropriate page number.

Battuta *GetfirstBatPage()

It returns the value *firstBatPage*.

Battuta *GetfirstBatNextPage()

It returns the value *firstBatNextPage*.

v**oid Init()**

It initialises the list of measures with a measure with tremble clef, in do maggiore (0#) and time 4/4. It assigns code 1 to such measure and puts *CodeCounter* = 2.

v**oid Free()**

It deallocates the whole list of measures, measures included.

v**oid SetCodeCounter(NumCode)**

It sets the CodeCounter value at nc.

NumCode GetCodeCounter()

It returns the value of CodeCounter.

v**oid GoTop(Pentagramma pent[],int nPent)**

It positions itself on the first page and recalls PageSetUp.

Bool GoForward()

It moves forwards of one measure (it returns FALSE if this is not possible).

Bool GoBackward()

It moves backwards of one measure (it returns FALSE if this is not possible).

Bool GoNextPage()

It moves forwards of one page. It returns FALSE if we are on the last page otherwise it returns TRUE.

Bool GoPrevPage()

It moves backwards of one page. It returns FALSE if we are on the first page otherwise it returns TRUE.

Bool PageDwn()

It moves forwards of one page. It returns FALSE if we are on the last page otherwise it returns TRUE.

void SetupCurrent(Pentagramma *pentagr, VUnit PosX, Vunit IntWdt, VUnit Batt2Up, VUnit Batt2Dwn, tipoSpartito tsp)

It disposes the current measure on the staff pointed by *pentagr*, with distance *PosX* from the beginning of the staff. It sets:

- *VU2Figur*e of the measure at *IntWdt,*
- *VU2Up* and *VU2Dwn* of the measure at the values passed in the arguments,
- The extensions of the bar line of the measure (depending on the type of score *tsp*).

It makes "scrolling forwards" *CurrentNode* and *PrevIntest.*

void CurrentInNextPage()

It puts the current node as first node of the following page.

void DrawPage()

It draws the current page.

void RedrawPage(Pentagramma pent[],int nPent)

It redisposes the measures of the page and redraws the page.

Bool Hit(const Point& p, SelObj, DrawObject*&, SymPath&)

It selects in the current page the object of the kind indicated from SelObj that contains the *p* point. See section 1.18.2 for a more complete description.

Bool Hit(const Rectangle& r, SelObj,short mainLayer,SymPath& SP1, SymPath& SP2)

It reports in the current page the symbols of the SelObj kind that are inside the rectangle *r*: the first object that is found in the rectangle is identified by *SP1*, the last one by *SP2*. In this case the selection take place in the specified layer through the parameter mainLayer. See section 1.18.2 for a more complete description.

Bool SymbolicHit(SymPath&, SelObj, DrawObject*&)

It decodes a path (see paragraph 1.18.2)

Bool FigPtr2SymPath(SymPath&, Figura*)

It reconstructs the path of a figure starting from the pointer.

Bool CmdAddBattuta(Battuta*,Bool insAfter, VUnit spaceUD)

It adds a new measure after or before the measure indicated according to the value of insAfter (TRUE ⇒ Dopo). The heading of the new measure is initialised in a way that it is equal to that of the previous measure. The parameter *spaceUD* is used to set to the new measure the space that has to remain empty above and below the staff. It reorganises the numbers of measures.

Bool CmdAddBattuta(Battuta*,Bool insAfter, VUnit spc, ArgType& nc, ArgType& np)

On the contrary of the previous command if the lectern is a MASAE, it returns the values of NumericCode (nc) and NumProgress (np) in the new measure, otherwise, if the lectern is a DLIOO one, the parameters are used to set the attributes NumericCode and NumProgress of the measure.

Bool CmdDelBattuta(Battuta*)

It deletes a measure of the list of measures and reorganises the number of measures.

Battuta *GetBatt(NumCode nc)

It returns the pointer to the measure with NumericCode nc.

Battuta GetBattWithFig(Figura fig)

It returns the Battuta that contains the figure. If the figure is not in this measure it returns NULL.

void Add(Battuta* bat)

It adds the measure passed as parameter to the bottom of the list.

void AddAfter(Battuta* bat)

It adds the measure passed as parameter in the position specified by the attribute *NumProgress* of the measure.

Bool DeleteSym(DrawObject* sym)

It deletes from the current page the symbol *sym* recalling the DeleteSym methods of the measures until one of these finds it and deletes it. It returns TRUE if the symbol has been deleted.

void Suona (MoodsMidiWrite *)

It creates the file to be played via Midi interface.

void SetfirstBatPage (Battuta*)

It sets the value of firstBatPage.

void GoBatt (NumCode nb)

Sets the position on measure nb.

Bool PageUp (Pentagramma pent[], int nPent, Bool)

It goes back one page, returns FALSE if it’s not possible.

void ResetCurrent ()

Takes if possible the intestazione (heading) of the previous measure.

Battuta* GetCurrentBat ()

It returns pointer to the current measure.

Bool MoveCurrentBatForw ()

Get next node of the list.

Bool CheckCurrent (tipoSpartito ts, Vunit& IntWdt, Vunit& Wdt, Bool firstColonna)

Checks the list of measures for the visualisation.

void SetupCurrentSimboli ()

Calls SetupSimboli of battuta.

void SetupSimboli ()

Calls SetupSimboli of battuta.

int Save (FILE*, Context)

Register the list of the measures on file in MDS format.

void ComprimiBattute (Bool abil_compress)

Used for multirest measures.

void CompressBatt (Battuta* pbat1, Battuta* pbat2, int nb)

Used for multirest measures.

int GetNumeroBatutteVere ()

Counts the number of measures, counting only one time multirest measures .

Bool printPage (Pentagramma[], int)

Printing method for the actual page.

void printPage ()

Printing method for the actual page.

24.3 Class ListaIntEst

24.3.1 Description

This class represents a list of extended intervals. All the symbols of interval related to any figure of the score are collected in this unique list. The order of the symbols inside the list is meaningless.

Since more than one extended interval can start and end on the same couple of figures, the couple of attributes *PfigStart, PFigEnd* of a symbol does not constitute a key for the research in the list. This implies that if starting from a figure or from a couple of figures we want to find the intervals that start or end on it/them, we have to scan the whole list.

The class has as task the positioning and the designing of the intervals. The intervals are drawn above or below the selected figures according to the value set in the *Sopra* attribute of the interval. To make easier the readability of the musical piece it has been decided to represent the intervals outside the staff: Legatura di Valore is an exception when two notes of two adjacent chords are selected. In addition the intervals are drawn following an order due to the type of the interval, according to the following list:

1. Legature di Valore
2. Legature Quadrate
3. Legature
4. Forcelle (Diminuendo e Crescendo)
5. Modifica Ottava
6. Cambio di Ritornello
7. Onda
8. Freccia.

In the case of an user interface of the DLIOO/PDLOO kind, it can happen that the interval extension foresees the possibility of breaking the interval in more segments having thus the possibility to perform a carriage return.

24.3.2 Father Class

Lista

24.3.3 Protected Attributes

NumCode CodeCounter

Numeric code of the intervals.

Lista LStartedSopra, LStartedSotto

These lists are employed only during the scanning of the measures for the positioning of the figures belonging to this list. They contain the interval symbols (placed respectively above and below the staff) that are currently begun but not yet ended. The starting x-axis value has already been assigned to the symbols of these lists, but not yet the width and nor the starting y-axis value (this latter can depend on the interval symbols that are found on the successive figures).

Bool Revision Y

When it is TRUE, the scanning of the measures produces only a review of the vertical positioning of the interval symbols, without modifying the width and the x-axis. It is to be put FALSE for the first scanning of the measures, and TRUE for the second one, in order to let it correct only eventual overlappings between symbols that have not been considered by the first scanning.

Bool onlyLegature

24.3.4 Protected Methods

Node* GetIntStartF(Figura *pF, Node *pN, Bool Accordo)

It scans the list by starting from the *pN* node in order to determine the next node corresponding to an interval that starts on the **pN* figure. The intervals Legatura di Valore inserted in a chord, are directly positioned and skipped. If it returns NULL, it has found nothing.

Node* GetIntEndF(Figura *pF, Node *pN)

It scans the list by starting from the *pN* node to determine the next node corresponding to an interval that ends on the **pF* figure. If it returns NULL, it has found nothing.

24.3.5 Public Methods

ListaIntEst()

Constructor that initialises the empty list.

~ ListaIntEst()

Destroyer that recalls the Free method().

void Init()

It initialises the list of intervals as void list.

void Free()

Deallocates all the pointers connected to the list.

void SetCodeCounter(NumCode nc)

It sets the Code Counter value at nc.

NumCode GetCodeCounter()

It returns the CodeCounter value.

IntEsteso *GetNumCodeInt(NumCode)

It returns the pointer to the extended interval with numeric code equal to the passed parameter. If such interval does not exists it returns NULL.

void SetRevisionY(Bool ry)

It sets *RevisionY* equal to *ry*.

Bool AddLegVal (CmdID cmd, Figura *pF1, Figura *pF2, Battuta *pb1, Battuta *pb2, TipoInserimento, ArgType Arg, ArgType& nc, int voice, Bool Multiv, int StemFirst, int StemLast,)

According to a command code *cmd*, an interval symbol is allocated for tie, added to the list and its attributes are assigned. If the pointers assigned are not compatible with the kind of interval (see *TestAndSet* method of the classes IntEsteso and Intervallo), the method performs nothing and returns TRUE.

Bool Add(CmdID cmd, Figura *pF1, Figura *pF2, Battuta *pb1, Battuta *pb2, TipoInserimento, ArgType Arg, ArgType& nc, int voice, Bool Multiv, Bool StemInt, int StemFirst, int StemLast, Bool Interm, Bool DrawInt)

According to a command code *cmd*, an interval symbol is allocated, added to the list and its attributes are assigned. If the pointers assigned are not compatible with the kind of interval (see *TestAndSet* method of the classes IntEsteso and Intervallo), the method performs nothing and returns TRUE.

void Add(IntEsteso *pInt)

It adds a symbol of IntervalloEsteso.

void Add(IntEsteso *pInt, ArgType& nc)

It adds a symbol of IntervalloEsteso.

void AddStartedSopra(IntEsteso *pInt)

It adds the symbol pointed by *pInt* to the list *LStartedSopra*. It maintains the list ordered to permit the positioning following the order according to the type of interval.

void AddStartedSotto(IntEsteso *pInt)

It adds the symbol pointed by *pInt* to the list *LStartedSotto*. It maintains the list ordered to permit the positioning following the order according to the type of interval.

void AddStarted(IntEsteso *pInt)

It adds the symbol pointed by *pInt* to the list *LStartedSopra* or *LStartedSotto* depending on its attribute *Sopra*.

void UpdateStartedSopra(const Point& Plv, const Point& P, VUnit YPent)

It prolongs the “started above” symbols till *P.x* or *Plv.x*, in order to overhang *P* and the staff of which we assign *YPent* (co-ordinate of the inferior line). It updates the vertical position of such symbols in order to avoid overlapping among them: the symbols are placed from the bottom toward the top following the order of the list *LStartedSopra*. We consider *Plv* or *P* according as the analysed interval is Legatura di Valore or not.

void UpdateStartedSopra(Figura *pF, VUnit Ypent, Bool inAccordo, Bool fine=TRUE)

It prolongs the "started above" symbols till the figure pointed by *pF*, in order to overhang such figure and the staff of which we assign *YPent* (co-ordinate of the inferior line). It updates the vertical position of such symbols in order to avoid overlapping among them: the symbols are placed from the bottom toward the top following the order of the list *LStartedSopra*.

void UpdateStartedSopra(const Point& Plv, const Point& P, Vunit Ypent, Bool fine=TRUE, Figura *pF, Bool inAccordo=FALSE)

It prolongs the "started above" symbols till the figure pointed by *pF*, in order to overhang such figure and the staff of which we assign *YPent* (co-ordinate of the inferior line). It updates the vertical position of such symbols in order to avoid overlapping among them: the symbols are placed from the bottom toward the top following the order of the list *LStartedSopra*.

void UpdateStartedSotto(const Point& Plv, const Point& P, VUnit Ypent, Bool fine=TRUE, Figura* pF=NULL, Bool inAccordo=FALSE)

It prolongs the "started below" symbols till *P.x* or *Plv.x*, in order to underlie *P* and the staff of which we assign *YPent* (co-ordinate of the inferior line). It updates the vertical position of such symbols in order to avoid overlapping among them: the symbols are placed from the top toward the bottom following the order of the list *LStartedSotto*. We consider *Plv* or *P* according as the analysed interval is Legatura di Valore or not.

void UpdateStartedSotto(Figura *pF, VUnit YPent, Bool fine=TRUE, Figura* pF=NULL, Bool inAccordo=FALSE)

It prolongs the "started below" symbols till the figure pointed by *pF*, in order to underlie such figure and the staff of which we assign *YPent* (co-ordinate of the inferior line). It updates the vertical position of such symbols in order to avoid overlapping among them: the symbols are placed from the top toward the bottom following the order of the list *LStartedSotto*.

void UpdateStarted(Figura *pF, VUnit Ypent, Bool inAccordo, Bool fine=TRUE)

It updates the width (and consequently the height) and the vertical position of the started symbols in order to overhang or underlie the figure pointed by *pF* (always outside the staff). It avoids overlapping among symbols.

void RemoveStarted(IntEsteso *pInt)

It deletes *pInt* from the "started". The deletion is performed considering the list of the "started" as a stack (in that only the last interval inserted can be deleted). If we want to delete an interval that is not the last inserted this has to be marked through the attribute *Remove* of IntervalloEsteso and will be physically deleted only when the above lying interval is deleted.

void ClearStarted()

It voids the "started" lists (obviously it does not deallocates the symbols).

void CancIntestFig(Figura *pF)

It finds the eventual symbols of IntervalloEsteso that start and end on the figure and deletes them.

void CancIntestBat(Battuta *pbat)

It finds the eventual symbols of IntervalloEsteso and deletes them.

void ResetPos()

Deallocates the Segmenti of all the intervals of the list.

void FindPrevStartedFig(Figura *pF)

It puts in the "started" the symbols started in the preceding pages that end on the figure pointed by *pF*.

void FindPrevStartedBat(Battuta *pB)

It puts in the "started" the symbols started in the preceding pages that end on the measure pointed by *pB*.

void ExamineFig(Figura *pF, Vunit Ypent, Bool Accordo, Bool inAccordo)

It checks if symbols start or end on the figure pointed by *pF* interval. In the assertive case it inserts or deletes from the "started" the found symbols. In any case it updates the dimensions and positions of the started.

void ExamineInizioPentagr(Battuta *pB)

If the assigned measure starts on a staff, it sets the position of each started symbol at the beginning of the measure.

void ExamineBat(Battuta *pB)

It checks if symbols start or end on the measure pointed by *pB* interval. In the assertive case it inserts or deletes from the “started” the found symbols. In any case it updates the dimensions and positions of the started on each figure of the measure.

void EndPage(Battuta *pB)

It ends the design of the started symbols at the end of the assigned measure and voids the two lists.

void EndRow(Battuta *pB)

It ends the design of the started symbols at the end of the assigned measure and adds a segment to the intervals of the “started”.

void Draw()

It draws the interval symbols that begin and end on the page (those outside the page have PFirstSegm=NULL)

Bool Hit(const Point& p, SelObj sobj, DrawObject*& objsel, SymPath&, symPath)

See section 1.18.2.

Bool SymbolicHit(SymPath&, SelObj, DrawObject*&)

It decodes the path symPath (see paragraph 1.18.2)

void ValutaLegature()

Calls ValutaLegature of class Legatura for each object

void SetOnlyLegature (Bool onlyLeg=TRUE)

Sets onlyLegature to TRUE.

void print()

It prints the interval symbols.

void AdjustVUFig()

It calls the setup on screen for the symbol

Bool DeleteSym (DrawObject *pD)

It deletes from the list the symbol passed as parameter (via pointer) e deallocate it.

int Save (FILE *, Context)

It register the list of intervals on file.

char *Describe (Context, NumCode)

It returns the description of symbol in MDS format.

void SetupIntRefs(Battuta*,Lista*)

Internal method it should not be called.

24.4 Class Spartito

24.4.1 Description

The class represents a score that consists in an unique staff, in other words an **orchestral part**. Its derivation from DrawObject is necessary since this class can be instanced as a part of a main score and, in such case, it is not the only graphic object that compares in the main window. The class contains the sequence of the measures and the interval symbols. Its main aim is the “synchronisation” of these two lists.

In case the score is part of a main score, it uses only the first of the allocated staffs (*pentagr[0]* i.e. **pentagr*); in this case the main score refers to this staff concerning its dimensions.

24.4.2 Father Class

DrawObject

24.4.3 Connected Types

enum tipoSpartito

SPART_UNICO It marks the score when it is unique in a main score.
SPART_SUPERIORE It marks the upper score of a main score.
SPART_GENERICO It marks one of the scores inside the main score.
SPART_INFERIORE It marks the inferior score of a main score.

24.4.4 Protected Attributes

tipoSpartito tipoSpart

Type of score.

int nRighi

Number of staff lines.

Pentagramma pentagr[MAXRIGHI]

Evident staffs of the score.

ListaBattute listaBat

List containing all the measures of the score.

ListaIntEst listaInt

List of all the interval symbols.

NumCode NumericCode

Numeric code of the score (NumCode is defined as short).

char strumes[STRUM_NAME_LEN]

Name of the instrument.

int SpaceInit

Bool ProlBarraUp, ProlBarraDown

short NumberOfStaff

int NCorde[3]

24.4.5 Protected Methods

void DrawPentagrammi ()

Draw the staffs.

24.4.6 Private Methods

Figura *GetNextFigura(DrawObject *pobj)

It returns the successive figure (if it is not a space) researching it also in the successive measure.

Bool Ordina(Figura *&f1, Figura *&f2)

It checks that f1 precedes f2, otherwise it changes the pointers. It returns FALSE if the figures do not belong to the same layer.

24.4.7 Public Methods

Spartito()

It initialises the data of the score.

~ Spartito()

It recalls Free().

void Free()

It calls the homonym method of ListaBattute and of ListaIntEst; it deallocates the eventual brace bracket.

void New()

It starts a new score, deletes the list of measures.

void SetTipo(tipoSpartito ts)

It sets the type of score.

tipoSpartito GetTipo()

It returns the type of score.

void SetWidth(VUnit w)

It sets the width of the score (it performs the homonym method on the first staff associated with the score).

void SetMaxWidth(VUnit w)

It sets the maximum width of the score (it performs the homonym method on the first staff associated with the score).

void SetSpaceUpDwn(VUnit sp)

It sets the space above or below the score (it performs the homonym method on the first staff associated with the score).

void SetCodeCounterBat(NumCode nc)

It sets the value of CodeCounter at nc.

NumCode GetCodeCounterBat()

It returns the value of CodeCounter.

void SetCodeCounterint(NumCode)

It sets the value of the CodeCounter of the list of IntervalliEstesi listaInt at nc.

NumCode GetCodeCounterInt()

It returns the value of CodeCounter of the list IntervalliEstesi listaInt.

NumCode GetPagCounter()

It returns the value of the attribute PagCounter of ListaBattute.

void SetBattPage(Battuta*)

It sets the measure passed as parameter as measure of the beginning of the page with the appropriate number.

Battuta *GetfirstBatPage()

It returns the value of the attribute *firstBatPage* of ListaBattute.

Battuta *GetfirstBatNextPage()

It returns the value of the attribute *firstBatNextPage* of ListaBattute.

VUnit GetVU2Up()

It performs the homonym method on the first staff associated to the score.

VUnit GetVU2Dwn()

It performs the homonym method on the first staff associated to the score.

VUnit GetVU2Lft()

It performs the homonym method on the first staff associated to the score.

VUnit GetVU2Rgt()

It performs the homonym method on the first staff associated to the score.

VUnit GetMaxWidth()

It performs the homonym method on the first staff associated to the score.

IntEsteso *GetNumCodeInt(NumCode)

It returns the pointer to the extended interval with numeric code equal to the passed parameter. If this interval does not exist it returns NULL.

void SetNumericCode(NumCode nc)

It sets the numeric code of the score.

NumCode GetNumericCode()

It returns the numeric code.

int Save(FILE *)

It saves the score on a file.

void DeleteBat(Battuta *bat)

It deletes the selected measure.

Bool DeleteSym(DrawObject *s)

It deletes the selected symbol.

void EraseBattuta(Battuta *bat)

It substitutes the selected measure with spaces; in addition it deletes eventual symbols of extended interval.

Bool DeleteFigure(Battuta *bat,NumCode FigCode)

It deletes the figure of the bat measure with NumericCode FigCode.

void SetNPentagrammi(int, const Rectangle&)

It sets the number of staffs and places them in the indicated rectangle. The positioning of the staffs in the rectangle is performed in order to provide each staff with the same empty space above and below it.

int GetNPentagrammi()

It returns the number of staffs that has been set.

void SetNCorde(int staff, int nCorde)

It sets the number of lines of the staff indicated by “staff” (0,1,2).

int GetNCorde(int staff=0)

It returns the number of lines of a staff.

void GoBatt (NumCode nb)

It is used for positioning on the measure nb.

Bool GoForward()

It moves forward of a measure without redrawing.

Bool GoBackward()

It moves backward of a measure without redrawing.

Bool GoNextPage()

It moves forward of a page. It returns FALSE if it is the last page, otherwise it returns TRUE.

Bool GoPrevPage()

It moves backward of a page. It returns FALSE if it is the first page, otherwise it returns TRUE.

Bool PageUp()

It moves backward of a page. It returns FALSE if it is the first page, otherwise it returns TRUE.

Bool PageDwn()

It moves forward of a page. It returns FALSE if it is the last page, otherwise it returns TRUE.

Battuta* GetBattNum(int n)

It returns the pointer to the *n* measure of the list.

Battuta* GetBatt(NumCode nc)

It returns the pointer to the measure that has numeric code nc.

Battuta* GetNextBat(Battuta* pbt)

It returns the pointer to the measure following *pbt*.

Battuta* GetPrevBat(Battuta* pbt)

It returns the pointer to the measure preceding *pbt*.

Battuta* GetLastBat()

It returns the pointer to the last measure of the list.

void SetStrumEsec(char*)

It sets strumes to the value passed as parameter.

char *GetStrumEsec()

It returns strumes.

void SetPos(DrawObject *d,const Point& p)

It sets the position of the score and of the associated staff. Such method is necessary only if the score appears inside a main score.

void ResetCurrent()

Referring to *listaBat,* it initialises the scanning of the current page.

Bool CheckCurrent(VUnit& IntWdt, VUnit& Wdt)

It provides the width of the heading and the width of the measure of *listaBat* currently examined.

void SetupCurrent(VUnit PosX, VUnit IntWdt)

It sets the “current” measure of *listaBat* by placing it in the position *PosX* with respect to the first staff and putting *VU2Figure* of the measure equal to *IntWdt*. It scrolls forward of a measure in the list.

void CurrentInNextPage()

It indicates that the “current” measure is the first of the successive page.

void PlaceIntervalli()

It disposes the interval symbols. Pay attention: it performs three successive scannings of the portion of measures’ list present in the page.

void Draw()

It draws the current page of the score.

void Redraw()

It redraws the current page of the score repositioning the measures.

void DrawTagli(const point&)

It draws the leger lines that are necessary to arrive to the indicated point. It scrolls the array of the staffs in order to see in which staff the point is to be found and then it recalls the method DrawTagli of the staff found.

Bool Hit(const Point&, SelObj,DrawObject*&, SymPath&)

It finds an object from the measures’ list.

Bool Hit(const Rectangle&,SelObj, short layer, SymPath&,SymPath&)

It performs a multiple selection on the measures’ list.

Bool SymbolicHit(SymPath&,SelObj,DrawObject*&)

It decodes the path.

Bool FigPtr2SymPath(Sympath&, Figura*)

It determines the path starting from a pointer.

void AddBattuta(Battuta *bat)

It adds a measure at the end of the score.

void AddBattutaDopo(Battuta *bat)

It adds a measure at the end of the score.

void AddIntEsteso(IntEsteso *intesteso)

It adds an extended interval in the list of extended intervals of the score.

CmdResult SymCommand(SymCmd&, Battuta *pb=NULL)

See the general documentation of the SymCommand method in section 1.19. In this class the parameter of the method is passed for reference, in fact by the insertion of a new measure, MASAE must modify some arguments of the command in order to pass the *NumericCode* and the progressive number to DLIOO (see ListaBattute).

void Suona (MoodsMidiWrite *)

It generates the file to be played via Midi interface.

void Init ()

It initialises the object.

void SetProlBarraUp (Bool)

Sets the parameter ProlBarraUp.

void SetProlBarraDown (Bool)

Sets the parameter ProlBarraDown.

Bool GetProlBarraUp ()

Returns the parameter ProlBarraUp.

Bool GetProlBarraDown ()

Returns the parameter ProlBarraDown.

void SetSpaceInit (int)

Sets the parameter SpaceInit.

int GetSpaceInit ()

Returns the parameter SpaceInit.

unsigned long GetNumeroBattute ()

Returns the number of measures.

unsigned long GetNumeroBattuteVere ()

Returns the number of measures counting only one the multirest measures.

void InitImpagina ()

Sets the page number to 0.

void SetPageCounter (NumCode)

Sets the value of the attribute PageCounter of ListaBattute.

void SetfirstBatPage (Battuta *)

Sets the value of the attribute firstBatPage of ListaBattute.

Battuta* GetBattCompNum (int n, Bool forward, int &nb)

Returns the pointer to #n measure of the list in compression multirest mode.

void ComprimiBattute (Bool abil_compress)

Compress multirest measure.

Battuta* GetBattNumProg (NumCode np)

Returns the pointer to the measure with progressive number=np.

Battuta* GetCurrentBat ()

Returns the current measure.

Bool MoveCurrentBatForw ()

Move the caret of the list one measure ahead.

void SetupCurrent (Vunit PosX, Vunit IntWdt, Vunit MaxWdt)

Setup measures for visualisation.

void AddIntEsteso (IntEsteso *intesteso, ArgType& nc)

Adds an extended interval (intervallo esteso) in the list of extended intervals of spartito.

void SetupCurrentSimboli ()

Calls the SetupCurrentSimboli of Lista Battute.

Bool GetStem (NumCode nb1, Figura *pF1, NumCode nb2, Figura *pF2, Bool ForAll=TRUE)

Returns TRUE if stem upward, FALSE if downward.

Bool Intermed (NumCode nb1, Figura *pF1, NumCode nb2, Figura *pF2)

Used to determine the bow of the slur.

char *Describe (Battuta *)

Returns the MDS format description.

DrawObject *GetObject (SymCmd *MyCmd)

Returns a pointer to the object in the model from a command with the mouse.

char *GetNome (void)

Returns strumes.

void print ()

Printing method.

void printdir ()

Printing method.

void SetupIntRefs();

It sets in the figures of the score the references to the horiz. symbols starting/ending/over each figure.

void SetNumberOfStaff(short nStaffs);
short GetNumberOfStaff();

Notes:

The addition of ListaIntEst is recent. Due to this "new entry" some commands directed to Battuta must now be directed to Spartito, that has to implement new methods in order to manage them. In fact we have still to cope with the following situations:

- When we delete a figure, ListaIntEst must be informed in order to delete all the symbols that start and end on the figure.
- When a single note passes to a chord, ListaIntEst must know the pointers of both note and chord in order to update the symbols to this change
- When a whole measure is deleted, ListaIntEst must know all the figures that have been deleted.

25 Acronyms

- ACL: Additional command list included into the CWF files
- ALM, Action Log Manager for storing in protected mannger the information related to the actions performed on the WDF objects according to the WDFGPC. It included into the UCM.
- AM, Action Manager: a part of the Local Distributor.
- CMN: Common Music Notation, a generic way to describe a certain type of music notation
- ETT: execution time trend include into the EWF files
- FINALE: a program for music notation quite diffuse among publishers. It has a format Enigma.
- KC, Key Cache: for caching key on the Local Distributor
- LDID: Local Distributor ID
- MILLA: Music Intelligent Language for describing the formatting rules of music notation.
- MuseData: A format used by CCRAH
- NIFF: A pretended interchange format for music
- Non-VIP: non visually impaired people
- SCORE: a format and program for music notation, very used by publishers in the past
- SMDL: Standard Music Description Language. An obsolete and never used standard for music coding derived from HyTime.
- UCM, Use Control Manager: Manger including the Key Cache and Action Log Manager
- VIP: visually impaired People
- WDF CLEAR HEADER: main file of the WDF object containing all details of classification, identification and protection, and all the whole description of the WDF structure, without protection details and checksum. It is used only for classification purpose into the Local Distributor.
- WDF Editor, see WEDEL Editor.
- WDF HEADER: main file of the WDF object containing all details of classification, identification and protection, and all the whole description of the WDF structure. It contains also the key for opening the Macro Components of the WEDEL object.
- WDFALM: WDF Action Log Manager, the demon collecting and negotiating the registration of the allowed operation and activities of the Local Distributor Clients.
- WDFCID: WDF Component identification number. It is comprised of three parts. The WDFID of the container object (if any), a code of the component type, a generated number.
- WDFCR: WDF Classification Record. The collection of fields describing the classification information of a WDF object or component
- WDFCWP: A record collecting the information related to watermarking audio and image files.
- WDFGPC: WDF General Permission Code. The list of allowed operations with the corresponding parameters for the accounting
- WDFID: WDF identification number comprised of two numbers, the Publisher ID and a generated number.
- WDFIR: WDF identification record. The collection of fields describing the identifications data of the WDF object or component.
- WDFitem: an element of a WDF section, a component of a WDF. It presents a WDFID, a textual description and a referred file according to its type.
- WDFitemCH: an element of a WDF section, a component of a WDF as mentioned in the WDF CLEAR HEADER. It presents a WDFID, a textual description and a referred file according to its type.
- WDFOOM: WDF Object Oriented Model, the OO model for WDF classification, identification, permission, protection aspects.
- WDFPIR: WDF Print Information Record. The collection of the objects that can be printed directly from the WDF structure with their settings and list of files.
- WDFPR: WDF Protection Record. The collection of fields describing the protection information of a WDF object or component.
- WDFPRCH: WDF Protection Record Clear header version. The collection of fields describing the protection information of a WDF object or component without the essential parts which are only contained in the encrypted header WDF HEADER and in particular in its WDFPR.
- WEDELEditor: Editor and Navigator of the WDF objects.
- WEDELMED: WEDEL Music Editor.

- WEDELOOMM: WEDEL Object Oriented Music Model.
- WNF: WEDEL Normal Form

26 File Extensions

Acronym	Description
AAC	Audio file
AVI	Video file, standard format
AWF	File describing the content and the classification of each audio file included. Each audio file has to present a AWF file for allowing the search and the identification.
BMP	Image file, standard format
BWF	file describing the content and the classification of each images included. Each image file has to present a BWF file for allowing the search and the identification.
CWF	additional command list file
DOC	file MS-Word
DWF	file describing the content and the classification of each document included. Each document has to present a DWF file for allowing the search and the identification.
EDF	Execution file containing the synchronisation among an audio file and a symbolic or an audio and an image score, a sort of the ETT of MOODS.
EPS	Encapsulated PostScript
KC	Key Cache or
ETT	Execution time trend of MOODS, a file containing how the measures of a symbolic music score are consumed during the execution. It is used for the paging mechanism of execution;
FON	files containing fonts for visualisation of symbols and user interface
GIF	Image file, standard format
HTML	file html for standard browser
HTM	Such as HTML
IWF	file containing the list of images belonging to a main score or a part for and image score music sheet
JPG	Image file, standard format
LWF	file containing its classification, identification and text corresponding to a lyric for a music part. The file contains also the reference in terms of WDFCID to the music part to which is assigned and any references to the singles notes.
MID	Midi file
MP3	Audio file in MP3 format
MPEG	Video file, in standard format
MWF	Milla files for music editor
PCX	Image file, standard format
PDA	file containing fonts for printing music scores
PDF	Adobe Acrobat format
PIC	Image file, standard format
PS	PostScript file
SND	Sound file, standard file
SWF	symbolic music file
TBL	table for visualising and printing music scores
TGA	Targa format for images
TIF	Image file, standard format
TIFF	Such as the TIF
TTY	True type font sources
TXT	file in standard ASCII
VWF	file describing the content and the classification of each video file included. Each video file has to present a VWF file for allowing the search and the identification.
WAV	Wave file in PCM, standard format
WDF	WEDELMUSIC object at whole
WMF	Window Meta file format, vectorial drawing format
WNF	WEDEL Normal Form
WTK	WEDEL Tool Kit
XML	A XML file in standard format

27 Bibliography and References

[Abbott85] C. Abbott, ``Guest Editor's Introduction to the Special Issue on Computer Music," ACM Computing Surveys, Vol. 17, N. 2, pp. 147-151, June, 1985.

[AdvTMod97] Sushil Jajodia, Larry Kerschberg – «Advanced Transaction Models and Architectures» - Kluwer 1997

[AIIA99] P. Bellini, F. Fioravanti and P. Nesi, ``Lavoro Cooperativo e Gestione delle Voltate nelle Orchestre," AIIA Notizie, pp. 52-55, September, 1999.

[AIIA99b] P. Bellini, F. Fioravanti, P. Nesi, ``Cooperative work and automatic page turning in orchestras", Workshop on Intelligent Systems for Art and Entertainment, AIIA, Spcial Interest Group on Intelligent Interfaces, University of Naples Federico II, Faculty of Science, 16 April 1999.

[Algor] «Cryptographic algorithms» - http://www.eskimo.com/~weidai/algorithms.html

[ALV99] Altavista search engine, http://image.altavista.com/, 1999

[Anderson91b] D. P. Anderson and R. Kuivila, ``Formula: A Programming Language for Expressive Computer Music," IEEE Computer, pp. 12-21, July, 1991.

[AppCrypt94] Bruce Schneier - «Applied Cryptography» - John Wiley & Sons, 1994.

[Baggi91] D. L. Baggi, ``Computer-Generated Music, special issue," IEEE Computer, pp. 6-9, July, 1986.

[Baggi91b] D. L. Baggi, ``Neurswing: An Intelligent Workbench for the Investigation of Swing in Jazz," IEEE Computer, pp. 60-63, July, 1991.

[Blostein91] D. Blostein and L. Haken, ``Justification of Printed Music," Communications of the ACM, Vol. 34, N. 3, pp. 88-99, March, 1991.

[Blostein92] D. Blostein and H. S. Baird, ``A Critical Survey of Music Image Analysis," in: Structured Document Image Analysis, (H. S. Baird and H. Bunke and K. Yamamoto, ed.), Springer Verlag, NewYork, USA, pp. 405-434, 1992.

[Blostein92b] D. Blostein and N. P. Carter, ``Recognition of Music Notation: SSPR'90 Working Group Report," in: Structured Document Image Analysis, (H. S. Baird and H. Bunke and K. Yamamoto, ed.), Springer Verlag, NewYork, USA, pp. 572-573, 1992.

[Booch94] G. Booch, Object-Oriented Design with Applications}. California, USA: The Benjamin/Cummings Publishing Company, 1994.

[Byrd84] D. A. Byrd, ``Music Notation by Computer," Department of Computer Science, Indiana University, USA, UMI, Dissertation Service, http://www.umi.com, 1984.

[BYTE98] N. Baldini, P. Bellini, F. Fioravanti and P. Nesi, ``Progetto MOODS: la musica incontra l'informatica," Byte Italia, pp. 76-82, Dicembre, 1998.

[Camurri91] A. Camurri and C. Canepa and M. Frixione and R. Zaccaria, ``HARP: A System for Intelligent Composer's Assistance," IEEE Computer, pp. 64-75, July, 1991.

[Capella98] Rick Taube, ``CCRMA Capella Music Editor," CCRMA, Stanford University, California, USA, 1998.

[Carter89] N. Carter, ``Automatic Recognition of Printed Music in the Context of Electronic Publishing," Dept. of Physics and Music, University of Surrey, www.npcimaging.com/thesis, February, 1989.

[Carter92] N. P. Carter and R. A. Bacon, ``Automatic Recognition of Printed Music," in: Structured Document Image Analysis, (H. S. Baird and H. Bunke and K. Yamamoto, ed.), Springer Verlag, NewYork, USA, pp. 456-476, 1992.

[Chua91] Y. S. Chua, ``Composition Based on Pentatonic Scales: A Computer-Aided Approach," IEEE Computer, pp. 67-71, July, 1991.

[CM98] Rick Taube, ``CCRMA, Common Music," CCRMA, Stanford University, California, USA, 1998.

[CompSec89] Jennifer Seberry and Josed Pieprzyk – «Cryptography: An Introduction to Computer Security» - Prentice-Hall, 1989.

[Cope91] D. Cope, ``Recombinant Music," IEEE Computer, pp. 22-28, July, 1991.

[CopyProt94] Chourhury, Maxemchuk, Paul, Schulzrinne – «Copyright Protection for Electronic Publishing over Computer Networks» - IEEE network magazine June 1994

[COX95] Cox, I.J.; Kilian, J.; Leighton, T.; Shamoon, T. Secure Spread Spectrum Watermarking for Multimedia. Princeton, NJ: NEC Research Institute, Technical Report 95-10, October 1995.

[Crypt] Doug Stinson - «Cryptography Theory and Practice»

[Dannemberg93] Dannenberg, R. B., ``A Brief Survey of Music Representation Issues, Techniques, and Systems," Computer Music Journal, 17/3, pp.20-30, 1993.

[Dannenberg86] R. B. Dannenberg, ``A Structure for Representing, Displaying and Editing Music," in: Proc. of the International Computer Music Conference, International computer Music Association, pp. 241-

248, October, 1986.

[Dannenberg90] R. B. Dannenberg, ``A Structure for Efficient Update, Incremental Redisplay and Undo in Graphical Editors,'' Software Practice and Experience, Vol. 20, N. 2, pp. 109-132, Febrary, 1990.

[Dannenberg91] R. B. Dannenberg and C. L. Fraley and P. Velikonja, ``Fugue: A Functional Language for Sound Synthesis,'' IEEE Computer, pp. 36-42, July, 1991.

[Dannenberg91b] R. B. Dannenberg and D. Rubine and T. Neuendorffer, ``The Resource-Instance Model of Music Representation,'' in: Proc. of the International Computer Music Conference, International Computer Music Association, pp. 428-432, October, 1991.

[Dannenberg93] R. B. Dannenberg, ``A Brief Survey of Music Representation Issues, Techniques, and Systems,'' Computer Music Journal, Vol. 17, N. 3, pp. 20-30, 1993.

[DEL96] Delaigle, J. F., De Vleeschouwer, C. and Macq, B. (1996). Digital Watermarking. Conference 2659- Optical Security and Counterfeit Deterrence Techniques, San Jose, February 1996. SPIE Electronic Imaging: Science and Technology, 99-110.

[DIG98] Digimarc Corporation. http://www.digimarc.com

[Donora] L. Donora, "Semiografia della nuova musica", Collana Studi Musica, G. Zaniboni

[DT99] Audio on Demand Server of Deutsche Telekom AG, http://www.audio-on-demand.de/, 1999

[EMARK] Brassil, Maxemchuk, O'Gorman «Electronic Marking and Identification Techniques to Discourage Document Copying« AT&T Bell Laboratories, Murray Hill - NJ

[Fingerhut99] Fingerhut Michel, "The IRCAM Multimedia Library: a digital Music library«, IRCAM 1999 available at http://mediatheque.ircam.fr/articles/texts/fingerhut99a

[Fioravanti98] F. Fioravanti, P. Nesi, «A Language for Music Managing'», Technical Report, Department di Sistemi e Informatica, University of Florence, DSI-RT23/98, 1998, Deliverable of MOODS project DE5.1, 1998.

[FireIS] Cheswick, Bellovin - «Firewalls and Internet Security: Repelling the Wily Hacker»

[FoundCrypt] Oded Goldreich – «Foundations of Cryptography»

[Fredriksen89] Fredriksen J, Martin M, Puig de la Bellacasa R, and von Tetzhner S: The needs of people with disabilities Published for EEC by Fundesco Telefonica, Madrid, June 1989

[Gordon85] J. W. Gordon, ``System Architectures for Computer Music,'' ACM Computing Surveys, Vol. 17, N. 2, pp. 191-234, June, 1985.

[Gourlay86] J. S. Gourlay, ``A Language for Music Printing,'' Communications of the ACM, Vol. 29, N. 5, pp. 388-401, May, 1986.

[Gourlay87] J. S. Gourlay, ``Spacing a Line of Music,'' The Ohio State University, Computer and Information Science Research Center, Columbus, Ohio, USA, OSU-CISRC-10/87-TR35, 1987.

[greenpaperUC] European Commission Green Paper, Living and Working in the Information Society: People First. COM(96)389.

[HandAppCrypt] Menezes, van Oorschot, Vanstone – «Handbook of Applied Cryptography»

[Haus91] G. Haus and A. Sametti, ``Scoresynth: A System for the Synthesis of Music Scores Based on Petri Nets and a Music Algebra,'' IEEE Computer, pp. 56-59, July, 1991.

[Hegazy87] W. A. Hegazy and J. S. Gourlay, ``Optimal Line Breaking in Music,'' The Ohio State University, Computer and Information Science Research Center, Columbus, Ohio, USA, OSU-CISRC-8/87-TR33, 1987.

[Heussenstamm87] G. Heussenstamm, ``The Norton Manual of Music Notation,'' Norton \& Company, New York, London, 1987.

[ICDAR99] S. Marinai, P. Nesi, ``Projection Based Segmentation of Musical Sheets", Proc. of the 5th International Conference on Document Analysis and Recognition, ICDAR'99, sponsorizzata dallo IAPR (International Association on Pattern Recognition, TC10 and 11), Bangalore, India, 20-22 September 1999, (in Press).

[Icking97] W. Icking, ``MuTEX, MusicTEX, and MusiXTEX,'' in: Beyond MIDI - The Handbook of Musical Codes, (E. Selfridge-Field, ed.), The MIT Press, London, pp. 222-231, 1997.

[ICMC95] Proceedings of the International Computer Music Conference, CA, USA, 1995.

[ICMC96] Proceedings of the International Computer Music Conference, Honk Kong, 1996.

[ICMC97] Proceedings of the International Computer Music Conference, Greece, 1997.

[ICMC98] Proceedings of the International Computer Music Conference, USA, 1998.

[ICMC99] Proceedings of the International Computer Music Conference, China, 1999.

[IEEEC99] P. Bellini and F. Fioravanti and P. Nesi, ``Managing Music in Orchestras,'' IEEE Computer, pp. 26-34, September, 1999.

[IEEECOMP99] P. Bellini and F. Fioravanti and P. Nesi, ``Managing Music in Orchestras,'' IEEE Computer, pp. 26-34, September, 1999.

[Jaffe91b] D. Jaffe and L. Boynton, ``An Overview of the Sound and Music Kits for the NeXT Computer," in: The Well-Tempered Object: Musical Application of Object-Oriented Software, (S. T. Pope, ed.), MIT Press, Cambridge, pp. 107-118, 1991.

[Jaffe97] D. Jaffe, ``The NeXT ScoreFile," in: Beyond MIDI - The Handbook of Musical Codes, (E. Selfridge-Field, ed.), The MIT Press, London, pp. 146-152, 1997.

[Johnson91] M. L. Johnson, ``Toward an Expert System for Expressive Musical Performance," IEEE Computer, pp. 30-34, July, 1991.

[Kato92] H. Kato and S. Inokuchi, ``A Recognition System for Printed Piano Music Using Musical Knowledge and Constraint," in: Structured Document Image Analysis, (H. S. Baird and H. Bunke and K. Yamamoto, ed.), Springer Verlag, NewYork, USA, pp. 435-455, 1992.

[Keefe91] R. Keefe, ``Composing by Musical Analog: A Look at Planetary Orbits," IEEE Computer, pp. 72-75, July, 1991.

[Kouroupetroglou95] Kouroupetroglou G. and Giza Nemeth. "Speech Technology for Disabled and Elderly People." 1995. In (Roe 1995).

[Kruijthoff97] Kruijthoff, A. "Survey on Access to Public and University Library Catalogues." TESTLAB, Extra Deliverable 2, CGL, Grave, 1997.

[Loy85] G. Loy and C. Abbott, ``Programming Languages for Computer Music Synthesis Performance Analysis," ACM Computing Surveys, Vol. 17, N. 2, pp. 235-265, June, 1985.

[LYC99] Lycos search engine, http://www.de.lycos.de, 1999

[Manderbacka] Ch,Manderbacka, O, Jyrhämä, D. Langolff and al: User needs, Tide project PLAY, D2.3, EEC, April 199

[MOODS] Bellini, P., and Fioravanti, F., and Nesi P., ``Managing Music in Orchestras," IEEE Computer, pp.26-34, September, 1999.

[MOODSLanguage] P. Bellini, F. Fioravanti, P. Nesi, M. B. Spinu, "MOODS a Music Format for New Applications", Computing in Musicology, MIT press, 1999/2000.

[Morita91] H. Morita and S. Hasimoto and S. Ohteru, ``A Computer Music System that Follows a Human Conductor," IEEE Computer, pp. 44-53, July, 1991.

[MP99] MP3.com homepage

[MS95] Microsoft Corporation. "Removing Barriers: Active Accessibility." Electronically published on the internet. August 1995. http://www.microsoft.com/enable/dev/msdn4.htm

[MS97] Microsoft Corporation. "Active Accessibility for software developers." Electronically published on the internet. July 1997. http://www.microsoft.com/enable/dev/msaa.htm

[MUSE] http://dbs.cordis.lu/EN_PROJl_search.html (Search for Multimedia Networks)

[MusicKit98] R. Taube, CCRMA, ``Music Kit", tech. rep., CCRMA, Stanford University, California, USA, 1998.

[Nesi98] P. Nesi, «Managing OO Projects Better», IEEE Software, pp.12-24, July-Aug 1998.

[Newcomb91] S. R. Newcomb, ``Standard Music Description Language with Hypermedia Standard," IEEE Computer, pp. 76-79, July, 1991.

[NIFF695] , ``NIFF 6a: Notation Interchange File Format," NIFF Consortium, July, 1995.

[PEL99] L. Pelliccia, «RIAA releases yearend anti-piracy statistics«, http://www.riaa.com/piracy/press/040699.htm

[Pennycook85] B. W. Pennycook, ``Computer-Music Interfaces: A Survey," ACM Computing Surveys, Vol. 17, N. 2, pp. 267-289, June, 1985.

[PIT96] Pitas, I. A method for signature casting on digital images. In IEEE International Conference on Image Processing (ICIP'96), volume III, pages 215-218, Lausanne, Switzerland, September 1996.

[Pope91] S. T. Pope, ``The Well-Tempered Object: Musical Application of Object-Oriented Software," (S. T. Pope, ed.), MIT Press, Cambridge, 1991.

[Protocols] «Security Protocols Overview» - http://www.rsa.com/standards/protocols/

[Rader96] G. M. Rader, ``Creating Printed Music Automatically," IEEE Computer, pp. 61-68, June, 1996.

[RIAA99] RIAA-FAQ, «Is technology helping the recording industry keep up with all the changes?«, http://www.ria.com/techn/techn_faq.thm, 1999

[Roads85] C. Roads, ``Research in Music and Artificial Intelligence," ACM Computing Surveys, Vol. 17, N. 2, pp. 163-190, June, 1985.

[Ross87] T.Ross, ``Teach Yourself. The Art of Music Engraving," Hansen Books, Miami, London, 1987.

[Roush88] D. Roush, ``Music Formatting Guidelines," The Ohio State University, Computer and Information Science Research Center, Columbus, Ohio, USA, OSU-CISRC-3/88-TR10, 1988.

[Scaletti88] C. A. Scaletti and R. E. Johnson, ``An Interactive Environment for Object Oriented Music Composition and Sound Synthesis," in: OOPSLA 1998, (N. K. Meyrowitz, ed.), Conference on

Object Oriented Programming Systems, Languages and Applications, SIGPLAN Notices 23(11), November 1988, San Diego, California, USA, pp. 222-233, Sept. 25-30, 1988.

[Schottstaedt97] B. Schottstaedt, ``Common Music Notation," in: Beyond MIDI - The Handbook of Musical Codes, (E. Selfridge-Field, ed.), The MIT Press, London, pp. 217-221, 1997.

[SCP98] SysCoP, System for Copyright Protection. http://syscop.igd.fhg.de

[SDMI99] SDMI « Secure Music Initiative«, http://www.sdmi.org, 1999

[SecDataCom94] Man Young Rhee – «Cryptography and Secure Data Communications» - McGraw-Hill, 1994.

[SelfridgeField97] E. Selfridge-Field, ``Beyond MIDI - The Handbook of Musical Codes," The MIT Press, London, 1997.

[Serp] «SERPENT A Candidate Block Cipher for the Advanced Encryption Standard» - http://www.cl.cam.ac.uk/~rja14/serpent.html

[SMDL10743] ISO/IEC DIS 10743, ``Standard Music Description Language," ISO/IEC, 1995.

[Smith97] L. Smith, ``SCORE," in: Beyond MIDI - The Handbook of Musical Codes, (E. Selfridge-Field, ed.), The MIT Press, London, pp. 252-282, 1997.

[Smoliar91] S. W. Smoliar, ``Current Research in Computer-Generated Music," IEEE Computer, pp. 54-56, July, 1991.

[Sola87] F. J. Sola, ``Computer Design of Musical Slurs, Ties and Phrase Marks," The Ohio State University, Computer and Information Science Research Centre, Columbus, Ohio, USA, OSU-CISRC-10/87-TR32, 1987.

[Stephanidis95a] Stephanidis C. and Michael Sfyrakis. "Current Trends in Man-Machine Interfaces: Potential Impact on People with Special Needs." In (Roe 1995).

[Stephanidis95b]. Stephanidis C., Anthony Savidis and Demosthenes Akoumianakis. "Tools for User Interfaces for All." In (Placencia Porrero and others 1995).

[Taupin97] D. Taupin and R. Mitchell and A. Egler, ``Using TEX to Write Polyphonic or Instrumental Music ver T.77," hprib.lps.u-psud.fr, 1997.

[TESTLAB] Technical Annex, TESTLAB project programme, SVB, Amsterdam, 1996.

[TheNet96] P. Nesi and N. Baldini and L. Mengoni, ``TheNET: Manuale Tecnico di Riferimento," Dipartimento di Sistemi e Informatica, Facolta` di Ingegneria, Universita` di Firenze, RT 31/96, Florence, Italy, 1996.

[VS94] Van Schyndel, R.G.; Tirkel, A.Z.; Osborne, C.F. A digital watermark.In: Int. Conf. on Image Processing, vol. 2, page 86-90, 1994.

[WAL91]Wallace, G. K. The JPEG still picture compression standard. Communications of the ACM, vol 34, no. 4, pp. 30-40, (1991).

[WEL99] A. Welsh « Recording industry releases 1998 consumer profile«, http://www.riaa.com/stats/press/consumer98.htm

[Wood89] D. Wood, ``Hemidemisemiquavers...and other such things. A concise guide to music notation," The Heritag Music Press, Dayton, Ohio, USA, 1989.

[ZK95] Zhao, J. and Koch, E. Embedding Robust Labels Into Images For Copyright Protection. In: Proc. of the international Congress on Intellectual Property Rights for Specialised Information, Knowledge and New Technologies, Vienna, Austria, August 21–25, 1995.

Consulted deliverables of PLAY project:

- Tide project PLAY, User needs, D2.3, EEC, (Ch, Manderbacka1, O. Jyrhämä, D. Langolff, and al), April 1997.

Consulted deliverables of CANTATE project:

- D1-1 -- Survey of Music Libraries
- D2-1 -- Survey of Music Publishers
- D3-3 -- Report on SDML Evaluation
- D5-3 -- Development Model with Summary and Recommendation
- CANTATE, Final Report

Consulted deliverables of HARMONICA project:

- D1.1.1 -- The concept "Key collection" in relation to the different types of users needs.
- D1.2.1 -- Cataloguing rules and bibliographic data formats
- D1.2.2 -- Classification systems
- D1.2.3 -- Subject heading and thesauri

- D1.3.1 -- Minimum Catalogue Information necessary for Search and Retrieval: Music Libraries, Sound Archives, Music Information Centers
- D2.1.1 -- Existing surveys of users needs
- D2.1.2 -- Report on preliminary study
- D2.2.1 -- Existing surveys of interface requirements
- D2.2.2 -- Report on preliminary study
- D3.1 -- Analogue documents, carriers and formats
- D3.2 -- Networking and digitisation
- D3.3 -- Archiving and managing digital information (preservation) including transfer
- D3.4 -- Local and networked access to digital information collections
- D4.1.1 -- Report from the Forum meeting in Athens (14-03-'97). Forum Meeting at which representatives of the current music projects in Europe focused on the technical aspects of their work.
- D4.1.2 -- Report from the Forum meeting in Amsterdam (28-06-'97). Forum meeting which examined the current status of music libraries and the needs of the users.
- D4.1.3 -- Report from the Forum meeting in San Sebastian (27-06-98). The draft outline of this Forum Meeting is available at: http://www.svb.nl/project/harmonica/Forum_outline.htm.
- D4.1.4 -- Report from the Forum meeting in Paris (20-11-98)

Consulted deliverables of MOODS project:

- DE2.1 -- Detailed System Requirements
- DE2.2 -- Co-operative End-User Practices on Scores, with examples
- DE2.3 -- User Practices on Databases of Scores, with examples
- DE5.1 -- MOODS Format for Scores, with examples
- DE7.3 -- MOODS for Schools of Music, with examples
- DE7.4 -- MOODS for Publishers, with examples
- DE7.5 -- MOODS for Orchestras and Theatres, with examples

Consulted deliverables of IMPRIMATUR project:

- IMP/I4062/A, Watermarking Technology for Copyright Protection: General Requirements and Interoperability
- Protection of Technological, Measures, INSTITUTE FOR INFORMATION LAW, AMSTERDAM, NOVEMBER 1998
- State of the Art 2, October 1997, Report prepared by Chris Barlas on behalf of the IMPRIMATUR Consortium, www.imprimatur.alcs.co.uk, Authors Licensing and Collecting Society Ltd., Marlborough Court, 14 - 18 Holborn, London EC1N 2LE, www.alcs.co.uk.
- FORMATION AND VALIDITY OF, ON-LINE CONTRACTS, INSTITUTE FOR INFORMATION LAW, AMSTERDAM, JUNE 1998
- Privacy, Data Protection and Copyright: Their Interaction in the Context of Electronic Copyright Management Systems, INSTITUTE FOR INFORMATION LAW, AMSTERDAM, JUNE 1998

References to other related and considered projects:

- CANTATE: http://www.svb.nl/project/cantate/cantate.htm
- CONCERTO: http://www.converto.org/concerto/concrt_e.htm
- COPEARMS: http://www.ifla.org/copearms
- DECOMATE: http://cdservera.blpes.lse.ac.uk/decomate/
- HARMONICA: http://www.svb.nl/project/harmonica/harmonica.htm
- IMEASY: http://www.dsi.unifi.it/~hpcn/wwwimeasy/wwwpag.html
- IMPRIMATUR: http://www.imprimatur.alcs.co.uk/
- MIRACLE: http://www.svb.nl/project/Miracle/miracle.htm
- MOODS: http://www.dsi.unifi.it/~moods
- MUSE: http://dbs.cordis.lu/cordis-cgi/srchidadb
- MUSICWEB: http://sun1.rrzn.uni-hannover.de/musicweb
- MUSTUTOR: http://www.ilsp.gr/mustutor/MusTutor.htm
- OCTALIS: http://www.igd.fhg.de/www/igd-a8/projects/octalis/index.html
- PLAY: http://www.svb.nl/project/play/play.htm
- studio-online: http://www.ircam/studio-online

- TALISMAN: http://www.igd.fhg.de/www/igd-a8/projects.html
- VENIVA: http://web.tin.it/marsilio/veniva/
- ECUP copyright legislation (www.kaapeli.fi/~eblida/ecup/lex/lex.htm)
- Towards an Information Society Approach; Communication Towards a European Framework for Digital Signatures and Encryption; Green Paper on Legal Protection for Encrypted Services, http://www2.echo.lu/legal/en/ecommerc/digsig.html, http://www2.echo.lu/legal/en/ecommerc/ecommerc.html, and project ECUP, TECUP, COPEARMS, etc.