UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE,
FISICHE E NATURALI

Dipartimento di Informatica e Comunicazione

Corso di Laurea in Scienze e Tecnologie della Comunicazione Musicale (cl.26)

PLAYER MIDI MULTIPIATTAFORMA PER DOCUMENTI IEEE1599

Relatore: Prof. Luca Andrea Ludovico

Correlatore: Dott. Adriano Baratè

Tesi di:

Marco Garzetti

Matricola: 739482

Anno accademico 2011/2012

INDICE

1 Tecnologie utilizzate → pag. 3

1.1 MIDI → pag. 3

1.2 Il linguaggio C# → pag. 6

1.3 Il formato IEEE1599 → pag. 8

1.4 Il linguaggio XML → pag. 13

1.5 Mono Project → pag. 15

2 IEEE1599 Midi Player → pag. 16

2.1 Descrizione dell'applicazione → pag. 16

2.2 Lettura del file XML – XML Linq → pag. 17

2.3 Conversione IEEE1599 – MIDI → pag. 17

2.4 Output dei messaggi MIDI → pag. 21

3 Descrizione del codice → pag. 23

3.1 Introduzione → pag. 23

3.2 Il file Xml reading.cs → pag. 23

3.3 Il file Main.cs → pag. 31

3.4 Il file MainWindow.cs → pag. 39

4 Conclusioni e sviluppi futuri → pag. 41

5 Bibliografia → pag. 44

PARTE 1: Tecnologie utilizzate

1.1: MIDI

Cenni storici

Il MIDI (musical instrument digital interface) è uno standard di comunicazione digitale sviluppato negli anni ottanta per permettere l'interazione tra strumenti musicali elettronici (e in seguito anche tra strumenti e PC) anche di produttori differenti. Si può dire che il MIDI sia allo stesso tempo:

- un protocollo, che definisce il formato dei messaggi digitali scambiati;
- un insieme di specifiche hardware per la circuiteria elettronica e le connessioni fisiche attraverso le quali questi messaggi vengono trasmessi e ricevuti;
- un formato di file, sviluppato in tempi successivi allo standard originario, che costituisce un formato unico di interscambio tra software per il sequencing;

Prima che lo standard MIDI venisse sviluppato, esistevano diversi sistemi proprietari, perlopiù analogici, per la comunicazione tra strumenti musicali, diversi da produttore a produttore. Tra questi si possono citare, ad esempio, il CV/GATE (analogico) e il Roland DCB (digitale).

Per sopperire alla mancanza di uno standard che potesse consentire la connessione tra strumenti di diversi produttori, Dave Smith e Chet Wood di Sequential Circuits pubblicarono nel 1981 le prime specifiche del MIDI, che fu standardizzato, in seguito ad una contrapposizione tra produttori americani e giapponesi, nel 1983.
Nonostante sia possibile al giorno d'oggi realizzare sistemi di comunicazione più performanti, il MIDI rimane lo standard più diffuso per la grande diffusione, la validità delle specifiche iniziali, e il basso costo di implementazione.

Specifiche hardware

IL MIDI utilizza un protocollo di trasmissione seriale, che consente di raggiungere velocità di 31,250 bit al secondo, generalmente attraverso cavo schermato con connessione DIN pentapolare.

Specifiche software

In sostanza, il MIDI è un protocollo che descrive l'informazione musicale attraverso messaggi in formato binario, in cui troviamo una corrispondenza diretta tra messaggi ed eventi musicali. Un messaggio MIDI non contiene alcuna rappresentazione del segnale audio, ma piuttosto una rappresentazione numerica di un evento musicale (inizio di una nota, fine di una nota, cambio di timbro) e della sua tipologia (altezza della nota, intensità della nota). I messaggi MIDI sono essenzialmente formati da due o tre byte di dati, di cui:

- uno status byte, identificato con un "1" iniziale, che identifica la tipologia del messaggio e il canale a cui esso è riferito (il MIDI supporta, per ogni connessione, 16 canali, a ciascuno dei quali è possibile assegnare unicamente un timbro:

  1TTT CCCC dove i tre bit "T" indicano la tipologia del messaggio, mentre i quattro bit "C" indicano il canale a cui il messaggio si riferisce;

- uno o due data byte, identificati con uno "0" iniziale, contenenti l'informazione che il messaggio veicola (come, ad esempio, l'altezza e l'intensità della nota):

  Con sette bit è possibile rappresentare 128 diversi valori (0 – 127). Avremo quindi a disposizione 128 valori per rappresentare, ad esempio, l'altezza della nota, e 128 valori per rappresentarne l'intensità (velocity).

Di seguito verrà riportata la sintassi dei messaggi MIDI necessari alla trattazione contenuta nel presente elaborato:

NOTE ON	
Status & Channel number	1001 CCCC
Note number	0NNN NNNN
Velocity	0VVV VVVV

NOTE OFF	
Status & Channel number	1000 CCCC
Note number	0NNN NNNN
Velocity	0VVV VVVV

Note on e note off sono i messaggi inviati, rispettivamente, all'istante in cui una nota inizia e all'istante in cui essa finisce. Con channel number si indica il canale a cui il messaggio si riferisce, con note number l'altezza della nota (0 – 127) e con velocity l'intensità (0 – 127).

- Note number → per determinare il note number corrispondente alla nota (espressa con la notazione anglosassone), è necessario tenere in considerazione che il do centrale (C4) della tastiera di un pianoforte corrisponde a un valore di note number 60. Per ogni unità aggiunta o sottratta a 60 ci si allontanerà, ascendendo o discendendo, di un semitono dal do centrale.
- Velocity → possiamo creare un'associazione tra l'indicazione di dinamica comunemente utilizzata nella scrittura musicale e valore di velocity:

ppp	1 – 15
pp	16 – 31
p	32 – 47
mp	48 – 63
mf	64 – 79
f	80 – 95
ff	96 – 111
fff	112 – 127

PROGRAM CHANGE	
Status & Channel number	1100 CCCC
Program number	0PPP PPPP

Il messaggio program change ha la funzione di indicare, per il canale specificato dal campo channel number, il timbro che il dispositivo che riceve il messaggio dovrà ad esso associare.

General MIDI

Il messaggio program change permette di associare ad ognuno dei 16 canali MIDI un differente timbro del dispositivo a cui il messaggio è indirizzato. Il general MIDI è una standardizzazione, pubblicata nel 1991, della corrispondenza tra program number e timbro ad esso associato. Inviare un messaggio program change per il canale 1 con data byte 0000 0000, ad esempio, farà in modo che il dispositivo che lo riceverà assegnI al canale 1 il timbro corrispondente (secondo la tabella General Midi) al program number 0, ossia “Acoustic Grand Piano”. Tutti i messaggi note on e note off riferiti al canale 1 faranno così produrre al generatore timbrico un suono di pianoforte acustico.

1.2: Il linguaggio C#

C# è un linguaggio di programmazione orientato agli oggetti sviluppato da Microsoft all'interno del progetto .NET, divenuto uno standard ISO nel 2003. C# è indipendente dalla piattaforma, nonostante esso sia stato creato per funzionare al meglio con il Framework .NET di Microsoft.

Caratteristiche

La natura object-oriented del linguaggio C# gli conferisce caratteristiche di:

- Incapsulamento: esiste una sorta di confine attorno ad ogni oggetto, che separa il suo comportamento esterno (pubblico) dalla sua implementazione interna (privata);
- Ereditarietà: una classe può ereditare da un'altra per estenderne le sue caratteristiche, riutilizzando le sue funzionalità. Una classe (sottoclasse) può ereditare da una sola classe (superclasse), ma essere "madre" per più classi (sottoclassi), così da formare delle gerarchie di classi.
- Polimorfismo: una variabile di un certo tipo, ad esempio, può riferirsi ad un oggetto superclasse per quel tipo. Ciò permette di sfruttare il fatto che sottoclassi diverse abbiano le stesse caratteristiche della propria superclasse. Inoltre, il polimorfismo consente di dare lo stesso nome a metodi che implementano la stessa funzione logica, ma operano su tipi di dati diversi.

C# è inoltre un linguaggio type-safe. Ciò significa che i tipi possono interagire tra di loro solo attraverso i protocolli da essi definiti. Ad esempio, non è possibile far interagire il tipo "integer" con il tipo "string". Le regole riguardanti i tipi sono molto restrittive in C#: non è così possibile, ad esempio, eseguire conversioni tra tipi incompatibili, se non rendendole esplicite attraverso un cast. Impedendo conversioni tra tipi diversi, il CLR contribuisce a garantire la sicurezza del sistema.

Supporto alla piattaforma: C# è usato principalmente per scrivere codice per piattaforma Windows. Nonostante C# sia un linguaggio standardizzato, le risorse dedicate al supporto di C# su piattaforme diverse da Windows sono molto scarse. Tuttavia, C# può essere utilizzato per scrivere software multipiattaforma utilizzando sistemi che permettano di far girare il codice C# su runtime diverse dalla Microsoft Common Language Runtime. Un esempio è il progetto Mono, che ha il proprio compilatore C# e la propria runtime, compatibile con Linux, Mac e Windows. Mono è

un'implementazione completa e open source del CLI (common language infrastructure), che comprende le librerie framework richieste e l'implementazione delle librerie di classi proprietarie di .NET.

1.3: Il formato IEEE 1599

IEEE 1599 è un formato basato sulla rappresentazione dell'informazione XML che permette di descrivere l'informazione musicale nel suo complesso, mettendone in relazione diverse rappresentazioni. IEEE 1599 è stato sviluppato presso il LIM, Laboratorio di Informatica Musicale dell'Università degli Studi di Milano.
La musica può essere descritta in molti modi diversi: attraverso rappresentazioni grafiche (le partiture), audio (file mp3, waw), video, e descrizioni della performance (MIDI). L'avvento dell'era digitale ha fatto si che nascessero diversi modi di rappresentare in maniera digitale il materiale musicale, dando vita a formati di file diversi, ciascuno dei quali possiede le caratteristiche appropriate per rappresentare al meglio una certa tipologia di informazione. Ad esempio, i file AAC, MP3 e PCM sono utilizzati per codificare le registrazioni audio, Csound e MIDI sono standard per le performance dirette da computer, GIF, JPEG e TIFF sono invece formati adatti a rappresentare l'informazione grafica, e cioè le partiture.
Il formato IEEE 1599 permette di collegare tra di loro le diverse rappresentazioni di un brano musicale per permetterne una fruizione globale, lasciando però inalterato il formato originale di ciascuna rappresentazione. I vari contenuti (simbolico, logico, grafico, audio e video) possono quindi essere collegati (anche in rappresentazioni multiple, come più file audio) da un unico documento.

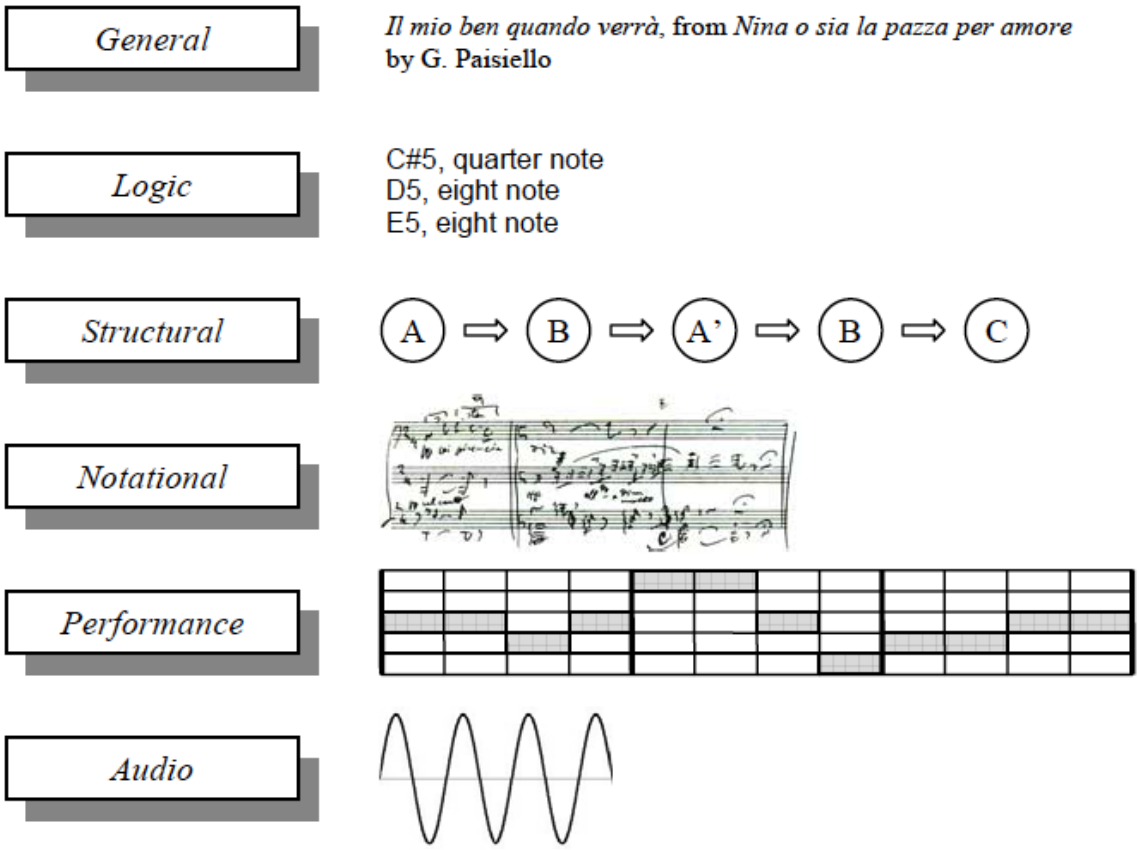
Il formato IEEE1599 non crea nuovi formati per la codifica dell'informazione (nuovi formati audio, o nuovi formati per la rappresentazione delle partiture), ma lascia gli oggetti multimediali nel proprio formato originale, all'esterno del documento XML, il quale invece incorpora una descrizione formale del brano, sottoforma di metadati e di una rappresentazione logica degli eventi musicali, collegata ai vari file all'interno dei vari livelli del formato, i layer, che corrispondono ai diversi livelli di rappresentazione dell'informazione musicale.

I layer

IEEE 1599 utilizza sei differenti layer per rappresentare l'informazione musicale:

- General – i metadati di catalogo relativi al brano;
- Logic – la descrizione logica dei simboli in partitura;

- Structural – l'identificazione degli oggetti musicali e delle loro relazioni reciproche;
- Notational – la rappresentazione grafica della partitura;
- Performance – la descrizione a livello di performance (conforme cioè ai linguaggi utilizzati per "far suonare" degli strumenti digitali) del brano musicale;
- Audio – registrazione digitale (o digitalizzata) del brano;




```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ieee1599 SYSTEM
       "http://www.mx.dico.unimi.it/ieee1599.dtd">
<ieee1599>
       <general>...</general>
       <logic>...</logic>
       <structural>...</structural>
       <notational>...</notational>
       <performance>...</performance>
       <audio>...</audio>
</ieee1599>
```

Ogni layer può contenere più rappresentazioni di quel tipo (ad esempio, il layer audio può far riferimento a più file audio), e non è necessario che sia fornita una rappresentazione per ogni layer.
Seguendo questa suddivisione in livelli, i simboli della partitura saranno descritti nel file XML all'interno del layer LOGIC, mentre le varie rappresentazioni del brano, come i file MP3 e le scansioni della partitura non saranno contenute nel file XML, ma semplicemente collegate e mappate nel layer corrispondente (audio o notational, in questo caso).

Layer Logic - Lo spine

Lo spine, contenuto nel layer logic, è il collante che crea una mappatura del brano a cui si riferiscono tutte le rappresentazioni multimediali degli altri livelli. L'informazione multimediale risulterà quindi mappata e messa in relazione a ogni singolo evento descritto all'interno dello spine. La scelta del significato da attribuire all'evento rappresentato dallo spine è lasciata intenzionalmente libera: nel caso più comune, verrà creato un evento per ogni nota, ma è possibile mappare solo ciò che si ritiene rilevante, come ad esempio i cambi armonici in una partitura jazz.

Lo spine costituisce quindi una lista di eventi musicali ordinati ed etichettati, a cui gli elementi degli altri layer possono far riferimento. La granularità degli eventi sarà una scelta di chi ha realizzato il file XML. Lo spine funge da collante tra i vari layer e rappresenta un livello di astrazione, dato che non necessariamente ogni evento dovrà corrispondere ad un simbolo in partitura.

Gli eventi descritti nello spine non hanno, di per se, alcun significato, ma dovranno avere una controparte in qualche layer. Ogni elemento <event> all'interno dello spine avrà tre etichette (tag): un id univoco che lo identifica, un tag timing (che rappresenta la sua collocazione temporale) e un tag hpos (che ne descrive la collocazione nello spazio della partitura). Tramite queste informazioni, possiamo risalire alla collocazione temporale degli elementi collegati agli eventi dello spine, ed avere una sincronizzazione temporale tra i diversi layer.
Ad esempio, un evento, con il proprio id univoco, potrà essere corrispondente ad una nota, che potrà essere contenuta in alcune partiture e suonata in più file audio. L'id dell'evento permetterà di riferire ad esso le diverse istanze delle diverse rappresentazioni di quell'evento nei vari layer. Sarà quindi possibile capire l'altezza e la durata della nota associata a quell'evento, la posizione all'interno della pagina che contiene la partitura, e la collocazione temporale dell'evento all'interno del file audio, oltre al messaggio MIDI da inviare ad un sintetizzatore per generare la nota rappresentata da quell'evento.

Grazie allo spine, la codifica IEEE1599 permette alle varie descrizioni multimediali del brano di avere dei riferimenti ad una struttura comune. Ciò crea sincronismo tra le varie istanze all'interno di un unico layer (ad esempio, tra diversi file audio) e tra i contenuti di layer diversi (file audio e rappresentazione grafica della partitura, ad esempio).

Lo spine contiene dunque soltanto la mappatura spazio-temporale degli eventi, ma non fornisce di per se alcuna informazione aggiuntiva riguardo ad essi.

Layer Logic – il Los

Il Los, anch'esso parte del layer Logic costituisce, insieme allo spine, la parte fondamentale del formato IEEE1599, poiché contiene le informazioni simboliche rappresentate in partitura.

All'interno del los troviamo una disposizione gerarchica degli elementi in partitura, secondo la quale ogni pentagramma può contenere più parti, ciascuna delle quali può contenere più voci. Come si può osservare dal file XML, troviamo un tag “staff_list”, che contiene la descrizione di tutti i pentagrammi dell'accollatura, un tag “part” per ogni parte, con un attributo che indica a che pentagramma si riferisce, all'interno del quale troviamo poi un tag “voice_list”, con la descrizione di tutte le voci di quella parte, e un tag “measure” per ogni misura, che conterrà a sua volta dei tag “voice” per le varie voci che possiamo trovare all'interno di una misura. In ogni tag voice troveremo un tag “chord” per rappresentare un accordo (anche una singola nota viene rappresentata come un accordo), all'interno del quale troviamo un tag “duration”, che esplicita, con i suoi attributi, la durata dell'accordo. All'interno del tag chord, ogni nota viene rappresentata con un tag “notehead”, che ha come arributi octave (che indica l'ottava su cui si trova la nota), step (che indica l'altezza della nota, con notazione anglosassone) e actual_accidental (che indica se la nota in questione è alterata). Il tag chord possiede un attributo event_ref, che permette di riferirlo all'evento dello spine tramite il suo id univoco, permettendo così di dare una collocazione temporale all'accordo.

Ecco un esempio di come viene rappresentata una nota all'interno del file XML:

```
<part id=”part_1”>
        <voice_list>
                <voice_item id=”part_1_0_voice” staff_ref=”part_1_staff” />
        </voice_list>
        <measure number=”1”>
                <voice voice_item_ref=”part_1_0_voice”>
                        <chord event_ref=”part_1_voice0_measure1_ev0”>
                                <duration num=”1” den=”4” />
                                <notehead>
                                        <pitch octave=”5” step=”D” actual_accidental=”natural” />
                                </notehead>
                        </chord>
                </voice>
        </measure>
</part>
```

1.4: Il linguaggio XML

L' XML (extensible markup language) è un linguaggio di markup (di marcatura) che permette di strutturare il contenuto definendo il significato dei suoi elementi. XML è un linguaggio estendibile perchè permette di creare etichettature personalizzate. Il contenuto di un file XML può essere suddiviso tra:

- DATI → contenuto (testo);
- METADATI → informazioni sul contenuto (etichette, o tag, definibili da chi crea il file).

XML è quindi, più propriamente, un metalinguaggio, ovvero un linguaggio che permette di definire altri linguaggi, per creare documenti strutturati.

XML è stato sviluppato negli anni '90 da un gruppo di lavoro, chiamato XML working group, e le sue specifiche sono state ufficializzate nel 1998. La principale esigenza che ha portato alla creazione di un linguaggio di etichettatura estendibile è stata quella di avere un linguaggio di markup che lasciasse maggiore libertà nella definizione di tag personalizzati.

Tag

All'interno di un documento XML è possibile, a seconda del tipo dell'informazione da strutturare e del software che dovrà leggere il documento, definire dei tag, ovvero dei marcatori, per definire la semantica del contenuto, rispettando alcune regole molto rigide. La mancanza dell'osservazione delle regole renderà il documento non conforme alle specifiche W3C, e quindi impossibile da validare.

Il sistema di etichettatura dell'XML si basa su due oggetti:

- Elementi, che racchiudono i frammenti di testo da etichettare;
- Attributi, che definiscono informazioni aggiuntive riguardo ad un elemento.

Ad esempio, il tag <ieee1599> rappresenta il tag radice di un documento IEEE 1599, in quanto racchiude tutte le informazioni e gli altri tag del documento, e possiede due attributi, “creator” e

“version”, che ne definiscono il creatore e la versione:

```
<ieee1599 creator="Finale Plugin" version="1.0">
</ieee1599>
```

XML e IEEE 1599

Diversi motivi e caratteristiche hanno portato all'utilizzo dell'XML come linguaggio per la creazione di documenti IEEE 1599. Tra queste, la principale è la sua capacità di descrivere entità come gli oggetti musicali così come sono, senza aggiungere informazioni inutili. L'XML permette una strutturazione gerarchica dell'informazione, perfettamente compatibile con la forte gerarchizzazione dell'informazione musicale. Inoltre, l'XML è espandibile e lascia spazio a futuri sviluppi del linguaggio musicale e a disparati usi di questo sistema di rappresentazione. Certamente, una rappresentazione di un brano musicale realizzata con un linguaggio XML non è immediatamente fruibile da un esecutore, ma necessita di appositi software che la possano convertire in informazione fruibile.

1.5: MONO PROJECT

Mono è una piattaforma software per lo sviluppo di applicazioni, e in particolare rappresenta un'implementazione open source del Framework Microsoft .NET, basata sugli standard ECMA. Mono include diversi componenti, tra cui:

- un compilatore C#, compatibile con tutte le versioni (fino alla 4.0) di C#, conforme alle specifiche ECMA;
- Mono runtime, un runtime che implementa il CLI (common language infrastructure) specificato dagli standard ECMA;
- Libreria di classi di base, compatibili con le classi del framework .NET di Microsoft;
- Libreria di classi di Mono, che aggiungono delle funzionalità alle funzionalità offerte dalle classi Microsoft, come quelle di supporto a GTK#;

Mono permette quindi di godere di tutti i benefici dell'utilizzo del framework .NET, come la popolarità e la gestione automatica della memoria, con il vantaggio di costituire un ambiente di sviluppo multipiattaforma: Mono funziona su Windows, Mac OSX, Linux e molti altri OS.
Inoltre il CLR (common language runtime) di Mono permette di scegliere tra diversi linguaggi di programmazione.

Gtk#

Gtk# è una libreria che permette di creare applicazioni GNOME e interfacce grafiche usando Mono, compatibili con Windows, MacOSX e Linux. GTK# su Windows permette di creare applicazioni graficamente simili alle applicazioni Windows classiche, mentre il suo utilizzo su MAC richiede il server X11 (un supporto alle interfacce utente grafiche).

PARTE 2: IEEE1599 Midi Player

2.1: Descrizione dell'applicazione

L'applicazione di cui il presente elaborato cerca di illustrare il funzionamento è sostanzialmente un lettore di file IEEE1599, il cui unico scopo è quello di creare e dare in output, partendo dalle informazioni contenute nel file XML, dei messaggi MIDI che possano "far suonare" un brano musicale a uno strumento elettronico, o a un sintetizzatore software.
Per fare questo sono necessarie, in primo luogo, le informazioni riguardo le note che devono essere suonate, e quindi la loro collocazione temporale, l'altezza e la durata. Queste informazioni possono essere ricavate dall'analisi del contenuto del layer Logic di un file XML IEEE1599.

In particolare, esaminando i tag <notehead>, all'interno del los, possiamo avere un "elenco" delle note che devono essere suonate, ed avere già a disposizione l'informazione sull'altezza (elemento "pitch") e sulla durata (elemento "duration" dell'elemento "chord" che contiene l'elemento "notehead" al quale facciamo riferimento). L'informazione riguardo alla collocazione temporale, invece, è ricavabile osservando l'attributo "timing" dell'elemento "event" all'interno dello spine al quale l'elemento "chord" si ricollega tramite l'attributo "event_ref".

A queste informazioni possiamo aggiungere altre informazioni utili ai fini della riproduzione MIDI, come lo strumento da associare ad ogni parte e la dinamica con la quale le note devono essere riprodotte. Per quanto riguarda lo strumento da associare ad ogni parte, si è scelto di lasciare la scelta all'utente permettendogli di associare ad ogni parte uno strumento scelto tra i 128 messi a disposizione dalle specifiche General MIDI, mentre per quanto riguarda la dinamica, non deducibile dalle informazioni contenute nel layer Logic, si è scelto di utilizzare un valore di velocity MIDI pari a 100 per tutte le note.

Ragionando a livelli di astrazione dell'informazione musicale, un MIDI Player per IEEE1599 realizza una "conversione" tra livello di rappresentazione logica e livello performance.

L'applicazione realizzata avrà quindi come input un file XML in formato IEEE1599, e avrà la funzione di fornire in output dei messaggi MIDI corrispondenti agli eventi del brano musicale rappresentato in formato testuale all'interno del file XML.

Per fare questo, sarà necessario creare:

1) del codice che proceda alla lettura degli elementi e degli attributi del file XML;
2) del codice che esegua una sorta di conversione tra i valori e le unità di misura utilizzati nel formato IEEE1599, e quelli contemplati dalle specifiche MIDI;
3) una parte di codice che gestisca l'output dei messaggi MIDI.

2.2: Lettura del file XML – XML Linq

Il primo passo per realizzare le funzioni richieste all'applicazione è quello di creare del codice che estragga le informazioni necessarie dal file XML. Il framework .NET mette a disposizione diverse API per lavorare con i dati XML.
Una di queste è la libreria di classi xml.linq, attraverso la quale l'intero documento XML può essere visto come un albero di oggetti (organizzati in modo gerarchico), chiamato DOM (document object model). Ogni oggetto che costituisce il file XML è rappresentato attraverso un tipo: XDocument (l'intero documento), XElement (l'elemento) e XAttribute (l'attributo). Questi tipi implementano dei metodi, che possono essere utilizzati per estrarre dei dati corrispondenti a certi criteri, e consentono di effettuare direttamente delle proiezioni utilizzando il linguaggio per le interrogazioni LINQ.

2.3: Conversione IEEE1599 – MIDI

Come detto in precedenza, le informazioni di nostro interesse ai fini della creazione dei messaggi MIDI direttamente estratte dal file XML IEEE1599 sono:

- Il timing (ossia la collocazione temporale delle note);
- L'altezza delle note;
- La durata delle note;

Timing

Per quanto riguarda il timing, esiste una fondamentale differenza tra il modo in cui questo viene rappresentato in un documento IEEE1599 e il modo in cui è necessario indicarlo ai fini dell'output dei messaggi MIDI così come viene gestito dalle librerie. Per ricavare l'informazione relativa alla

collocazione temporale delle note dal file XML dovremo far riferimento al valore dell'attributo “timing” degli elementi “event”, a cui fanno riferimento le singole note “notehead”, attraverso l'elemento padre “chord”, che contiene un attributo (event_ref) che corrisponde esattamente all'id dell'evento contenente le informazioni sulla temporizzazione.
L'attributo timing di ogni evento, tuttavia, non rappresenta il timing assoluto in cui quell'evento avviene, bensì i vtu (virtual time units) trascorsi dall'evento precedente. Occorrerà quindi, in primo luogo, attribuire un significato in un'unità di misura temporale reale (ad esempio, millisecondi) ai vtu, e in secondo luogo esprimere il timing di ogni evento in modo assoluto, quindi non relativo all'evento precedente ma all'istante di inizio del brano musicale.
Per attribuire un valore in millisecondi ai vtu è necessario disporre di due informazioni:

- il bpm del brano (beats per minute), considerando che ogni battito corrisponde ad un quarto;
- i vtu corrispondenti ad un quarto;

Per quanto riguarda il bpm, non essendo questo rappresentato tra le informazioni contenute nel file XML, la scelta è lasciata all'utente. Un discorso a parte merita la questione dei vtu/quarto: questo valore è spesso contenuto all'interno di un tag “vtu_amount” contenuto nell'elemento logic/los/staff_list/staff/time_signature/time_indication, che tuttavia è opzionale.
Non potendoci basare sul contenuto di questo tag, si è scelto di risalire a questo valore analizzando la differenza di timing (in vtu) tra due eventi successivi, riconducendo questo valore a quello corrispondente ad una nota da un quarto (potendo disporre dell'informazione riguardante la durata frazionaria della nota). Ad esempio, supponiamo di avere una nota da tre quarti. Ad essa sarà associato un certo evento, con il suo timing. Si presume (anche se ciò non sempre si verifica) che il timing dell'evento successivo rappresenterà i vtu trascorsi dall'inizio della nota che stiamo considerando all'inizio di quella successiva, e che quindi la differenza tra i due valori di timing corrisponderà ai vtu di durata della nota da tre quarti. Per ricavare l'informazione sui vtu corrispondenti ad un quarto, sarà quindi sufficiente dividere i vtu relativi alla durata della nota da tre quarti per tre. Dato che non è sempre vero che la differenza di timing tra due eventi successivi possa essere associata alla durata della nota corrispondente al primo evento (a volte la fine di una nota non corrisponde con l'inizio di quella successiva), il valore di vtu per quarto viene ricavato osservando il valore più frequente tra l'elenco dei valori di vtu per quarto ricavati dall'analisi della durata in vtu di tutte le note del brano.
Una volta trovata la quantità di vtu per un quarto, sarà sufficiente sapere quanti quarti ci sono in un minuto (bpm) per ricavare attraverso poche operazioni la durata in millisecondi di un vtu, e quindi

trovare il timing assoluto in millisecondi per ogni evento (e, quindi, per ogni nota).

Altezza

Nel file XML IEEE1599, l'informazione relativa all'altezza delle note è contenuta globalmente in tre attributi dell'elemento “pitch”, figlio dell'elemento “notehead”:

<pitch octave=”5” step=”D” actual_accidental=”natural” />

- octave: indica l'ottava su cui si trova la nota, in modo simile a quello utilizzato nella notazione anglosassone (ad esempio, un valore di 4 significherà che la nota si trova sull'ottava centrale della tastiera del pianoforte);
- step: corrisponde al nome della nota, espresso in notazione anglosassone (A, B, C, D, E, F, G);
- actual_accidental: indica se e come la nota deve essere alterata (natural → nessuna alterazione, sharp → + un semitono, flat → - un semitono);

Una semplice funzione, all'interno del codice, dovrà calcolare, dati questi tre parametri, il MIDI note number corrispondente. Considerando che C4 (il do centrale) corrisponde ad un notenumber 60, è facile calcolare che C0 (il primo do) corrisponderà a un notenumber pari a 12. Da qui, l'algoritmo esegue il calcolo associando ai valori dell'attributo “step” i seguenti valori:

STEP	NOTENUMBER
C	12
D	14
E	16
F	17
G	19
A	21
B	23

che verranno aumentati o diminuiti di un'unità nel caso in cui il valore dell'attributo actual_accidental sia, rispettivamente, sharp o flat. Per riportare il notenumer all'ottava corretta, basterà infine aggiungere 12 semitoni per ogni incremento di un'unità del valore dell'attributo “octave”.

Durata

Nel file XML la durata è espressa nel tradizionale modo frazionario con gli attributi “num” e “den” all'interno del tag “duration”, figlio del tag “chord”. Tale durata vale per tutte le note che compongono l'accordo, quindi per tutte le note rappresentate da tag “notehead” contenuti nello stesso elemento “chord”:

<duration num=”1” den=”4” />

Avendo a disposizione i vtu di un quarto, la durata della nota in termini di multipli o frazioni di quarti, e la durata in millisecondi di un vtu, basterà eseguire qualche operazione per ottenere la durata in millisecondi di ciascuna nota.

Parte

Una volta calcolati timing, altezza e durata di ogni nota, manca soltanto l'informazione relativa alla parte a cui ciascuna nota appartiene. Il MIDI può ricevere messaggi “indirizzati” verso 16 canali, ognuno dei quali può corrispondere a un diverso strumento. Il codice dovrà quindi incorporare una funzione che assegna un numero da 1 a 16 ad ogni parte.

Per ogni nota, dovranno essere generati due messaggi:

- MIDI NOTE ON: con channel number e note number, marcato con il timing della nota;
- MIDI NOTE OFF: con channel number e note number, marcato con il timing di fine della nota (timing iniziale + durata della nota);

Oltre ai messaggi strettamente legati alla riproduzione delle note, il software dovrà generare, prima dell'inizio della riproduzione, dei messaggi di PROGRAM CHANGE, per fare in modo che lo strumento che riceverà i messaggi MIDI associ ad ogni parte (indicata dal channel number) il timbro general MIDI scelto dall'utente.

2.4: Output dei messaggi midi

Una volta ricavati dalla lettura e conversione del file XML tutti i parametri necessari alla creazione di messaggi MIDI temporizzati, si rende necessario l'utilizzo di una libreria di classi che possa gestire l'output di messaggi MIDI attraverso le periferiche collegate al computer. Esistono diverse librerie di terze parti utilizzabili, e in particolare:

- C# Midi Toolkit;
- RtMidi;
- PortMidi;

C# Midi Toolkit, così come altre librerie, presenta il vantaggio di essere utilizzabile in C#, ma fa utilizzo della dll per la gestione di input/output multimediali di Windows winmm.dll, ed è quindi inutilizzabile per la realizzazione di un'applicazione multipiattaforma. RtMidi e PortMidi, invece, non fanno utilizzo di questa dll, e sarebbero quindi utilizzabili anche su altre piattaforme, ma sono librerie C++, che non possono essere utilizzate direttamente in C#, se non creando un Wrapper, ossia del codice "ponte" che fa si che le funzioni C++ proprie della libreria possano essere utilizzate all'interno di codice "managed", come quello scritto in C#. In particolare, il progetto PortMidi contiene già al suo interno un'implementazione di base (pm_managed) che consente di implementare le funzioni desiderate all'interno di un progetto CLR (common language runtime).
Si è quindi inizialmente scelto di utilizzare la libreria Portmidi, implementando tutte le funzioni necessarie all'output di messaggi MIDI in linguaggio managed. La scelta presentava tuttavia un problema: un progetto CLR non è compilabile su piattaforme diverse da Windows.
Data l'impossibilità di reperire o adattare una libreria che rispondesse alle caratteristiche di:

- compatibilità con il linguaggio C#;
- possibilità di utilizzo in multipiattaforma;

si è scelto di risolvere il problema rinunciando alla compatibilità diretta, a livello di codice, con il linguaggio C#, sfruttando un progetto di test già realizzato all'interno di Portmidi, opportunamente modificato, che consente di creare un'applicazione console a cui l'applicazione principale possa passare i parametri dei messaggi MIDI temporizzati da dare in output in maniera testuale, mantenendo quindi la parte di lettura dell'XML e creazione dei messaggi e la parte che gestisce

l'output dei messaggi MIDI su due processi differenti.

Una volta letto il file XML, l'applicazione passerà quindi al processo che gestisce l'output dei messaggi i parametri per la creazione dei messaggi MIDI in modo testuale, che il codice Portmidi memorizzerà in un array di messaggi temporizzati che a sua volta verrà passato al codice di gestione dell'output e mandato in riproduzione.

PARTE 3: Descrizione del codice

3.1: Introduzione

Le funzionalità realizzate dal codice dell'applicazione sono state divise tra tre file di progetto .cs, con il solo scopo di mantenere delimitata l'implementazione delle funzioni, anche in vista dell'utilizzo, in futuro, del codice all'interno di applicazioni più complete. In particolare, troviamo:

- il file Xml_reading.cs, che contiene l'implementazione del namespace Xml_reading, contenente tutti i tipi e le funzioni necessarie alla lettura del file Xml e alla conversione dei parametri, per dare in output una struttura contenente una rappresentazione dei messaggi compatibile con l'input richiesto dal processo di output dei messaggi MIDI;
- il file Main.cs, ovvero il punto di inizio dell'esecuzione principale, che contiene il codice che controlla l'interazione tra i due processi: inizializza la lettura del file xml, gestisce il play e lo stop avviando il processo di output dei messaggi MIDI, passa a quest'ultimo i parametri in formato testuale e termina il processo;
- il file MainWindow.cs, che contiene l'implementazione di tutte le funzionalità dell'interfaccia grafica;

3.2: Il file Xml reading.cs

Il codice contenuto all'interno del namespace Xml_reading contiene l'implementazione della classe xml_reading, che contiene tutti i metodi per la lettura dell'xml, oltre alla definizione di quattro tipi, utilizzati all'interno della classe xml_reading.
Per avviare la lettura del file xml è necessario creare un'istanza della classe xml_reading, e richiamare poi attraverso questa istanza i metodi che eseguono le necessarie operazioni di lettura e conversione dell'xml:

METODO xml_initialize(string file_name)

```
public void xml_initialize(string file_name)
                {
                        file = XDocument.Load (file_name);

title=file.Element("ieee1599").Element("general").Element("description").Element("main_title").Value.ToString();
```

```
                int number = 1;
                foreach (XElement part_element in file.Elements("ieee1599").Elements("logic").Elements("los").Elements("part"))
                {
                    part_list.Add(new part(part_element.Attribute("id").Value.ToString(), number));
                    number++;
                }

                CompileNoteArray();
                CompileEventArray();
            }
```

questo metodo riceve in ingresso il percorso del file xml da aprire, crea un nuovo XDocument, ed estrae dal layer general del file xml le informazioni riguardo il titolo dell'opera. Inoltre, attraverso la lettura degli elementi contenuti nel tag “part” del los, crea una struttura (denominata part_list) composta da istanze di tipo part, che ha come campi l'id della parte, e un numero progressivo, partendo da 1, che indicherà il part number. Per comodità, da questo metodo vengono richiamati altri due metodi necessari alla lettura del file xml: CompileNoteArray() e CompileEventArray:

METODO CompileNoteArray()

```
public void CompileNoteArray()
            {

                foreach (XElement note_element in file.Elements("ieee1599").Elements("logic").Elements("los").Elements("part").Elements("measure").Elements("voice").Elements("chord").Elements("notehead"))

                {

                    note_octave = Convert.ToInt32(note_element.Element("pitch").Attribute("octave").Value);
                    note_step = Convert.ToChar(note_element.Element("pitch").Attribute("step").Value);

                    try
                    { note_alteration = note_element.Element("pitch").Attribute("actual_accidental").Value;}
                    catch
                    { note_alteration = "natural";}

                    int notenmbr = 0;
                    int alteration = 0;
                    if(note_step == 'C'){notenmbr = 12;};
                    if(note_step == 'D'){notenmbr = 14;};
                    if(note_step == 'E'){notenmbr = 16;};
                    if(note_step == 'F'){notenmbr = 17;};
                    if(note_step == 'G'){notenmbr = 19;};
                    if(note_step == 'A'){notenmbr = 21;};
                    if(note_step == 'B'){notenmbr = 23;};
```

```
                    if(note_alteration == "natural"){alteration = 0;};
                    if(note_alteration == "flat"){alteration = -1;};
                    if(note_alteration == "sharp"){alteration = +1;};

                    note_number = notenmbr + (note_octave*12) + alteration;


                    note_chord_eventref = note_element.Parent.Attribute("event_ref").Value.ToString();
                    note_duration_num =
Convert.ToInt32(note_element.Parent.Element("duration").Attribute("num").Value.ToString());
                    note_duration_den =
Convert.ToInt32(note_element.Parent.Element("duration").Attribute("den").Value.ToString());
                    note_part_id =
note_element.Parent.Parent.Parent.Parent.Attribute("id").Value.ToString();

                    foreach (part part in part_list
                        .Where (a => a.id == note_part_id)
                        .Select (a => a)){note_part_number = part.number;}

                            note.Add(new nota(note_part_id, note_chord_eventref, note_part_number,
note_duration_num, note_duration_den, note_octave, note_step, note_alteration, note_number));

                }
            }
```

Questo metodo esamina tutti gli elementi “notehead” contenuti nel file xml, procede alla conversione dei parametri riguardo l'altezza della nota in notenumber, attraverso la lettura degli attributi dell'elemento “pitch”, ed estrae altre importanti informazioni riguardanti la nota, come l'id dell'evento a cui essa è collegata (note_chord_eventref), il numeratore e il denominatore della frazione che rappresenta la sua durata (note_duration_num e note_duration_den) e l'id della parte, attraverso il quale viene ricavato anche il partnumber attraverso la lettura della struttura part_list. Una volta memorizzati tutti i valori dei parametri in variabili provvisorie, per ogni nota, si procede all'aggiunta di un'istanza di tipo nota, con il passaggio dei relativi parametri al costruttore di default, a una struttura di nome note. Il tipo nota ha tra i campi tutti i parametri sopracitati.

METODO CompileEventArray()

```
            public void CompileEventArray()
            {

                foreach (XElement evento in
file.Elements("ieee1599").Elements("logic").Elements("spine").Elements ("event"))
                {
                    event_event_id = evento.Attribute("id").Value;
                    event_event_relativeTiming = Convert.ToInt32(evento.Attribute("timing").Value);

                    //creazione di un array contenente tutti i valori possibili di vtu per quarto
                    foreach (nota nota in note
                                .Where (a => a.chord_eventref == event_event_id &&
note.IndexOf(a) != 0)
```

```
                                        .Select (a => a))
                                    {
                                    int index = note.IndexOf(nota)-1;
                                    int vtu =
(((event_event_relativeTiming/note[index].duration_num)*note[index].duration_den)/4);

                                    if (vtu != 0)
                                    { note_vtu_list.Add (vtu);

                                    }

                                    }


                                    eventi.Add(new evento (event_event_id, event_event_relativeTiming));

                        }

                        quarter_vtu = note_vtu_list.GroupBy(item => item).OrderByDescending(g =>
g.Count()).Select(g => g.Key).First();
                                    note_vtu_list.Clear();

                }
```

Il metodo CompileEventArray, in modo analogo a quanto avviene con le note, procede alla lettura degli elementi “event”, all'interno dello spine, e crea una struttura, denominata eventi, composta da istanze di tipo evento, passando come parametri al costruttore di default l'id dell'evento e il timing relativo di ciascun evento.

Oltre a questo, all'interno di questo metodo è contenuto l'algoritmo per la definizione del parametro quarter_vtu: come si può vedere dal codice, viene creata una struttura, denominata note_vtu_list, contenente tutti i possibili valori di vtu per quarto ricavati dall'analisi della durata in vtu di ciascuna nota (ricondotti al proprio valore per un quarto). Gli elementi della struttura vengono poi ordinati attraverso un'istruzione linq, in modo tale da estrarre soltanto un valore, corrispondente al valore più ricorrente all'interno della lista.

METODO event_timing()

```
private void event_timing()
                {
                 bpm = Convert.ToDouble(string_bpm);
                 double ms_vtu = ((double)1/((double)bpm/(double)60000))/(double)quarter_vtu;


                            for (int i = 0; i < eventi.Count(); i++) //ad ogni evento assegna un timing assoluto,
sommando al suo timing relativo il timing assoluto dell'evento precedente.
                            {
                                    if (i != 0)
                                    {
                                            eventi[i].absolute_timing = eventi[i].relative_timing + eventi[(i-
```

```
1)].absolute_timing;
                                        }
                                        else if (i == 0)
                                        {
                                                eventi[i].absolute_timing = 0;
                                        }

                                eventi[i].ms_timing = ms_vtu * eventi[i].absolute_timing;

                                foreach (nota nota in note
                                                .Where (a => a.chord_eventref == eventi[i].id)
                                                .Select (a => a))
                                        {

                                                nota.ms_timing = eventi[i].ms_timing;
                                        //assegnazione della durata in ms alla nota
                                                nota.ms_duration                                                =
((quarter_vtu*4)/nota.duration_den)*nota.duration_num * ms_vtu; //vtu durata * ms_vtu
                                        max_timing = Convert.ToInt32(nota.ms_timing) + 1000;
                                        }

                                }
                }
```

Questo metodo si occupa di realizzare tutta la conversione del timing relativo espresso in vtu associato agli eventi in timing assoluto espresso in millisecondi, e di assegnarlo alle note associate agli eventi, per poi procedere anche al calcolo della durata di ogni nota in millisecondi.
ms_vtu rappresenta la quantità di millisecondi associati ad un vtu, calcolata partendo dal bpm (valore memorizzato nella variabile bpm), supponendo che ogni battito corrisponda ad un quarto, ed avendo a disposizione il valore quarter_vtu, che indica la quantità di vtu presenti in un quarto.
Fatto ciò, viene assegnato un valore al campo absolute_timing, corrispondente al timing assoluto di ogni evento, partendo dal quale viene calcolato, per ogni evento, il valore di timing assoluto in millisecondi (ms_timing), assegnato anche alle note associate ad ogni evento. Per ciascuna nota, infine, viene calcolata la durata in millisecondi, conoscendo i vtu per quarto (quarter_vtu), la durata in quarti della nota e i millisecondi per vtu (ms_vtu). Ad ogni assegnazione del timing assoluto in millisecondi ad una nota, viene aggiornato anche il valore max_timing, che rappresenta il timing totale del brano, con un'aggiunta di un secondo, dato che questo valore verrà utilizzato per gestire la chiusura del processo di output dei messaggi MIDI.

METODO Create_midimessagearray()

```
public void Create_midimessagearray()
        {

            foreach (nota nota in note)
            {
                midi_message.Add(new midi_message("noteon", nota.part_number,
Convert.ToInt32(nota.ms_timing), Convert.ToInt32(nota.note_number)));
                midi_message.Add (new midi_message("noteoff", nota.part_number,
Convert.ToInt32(Convert.ToInt32(nota.ms_timing)+nota.ms_duration), nota.note_number));
            }

        }
        }
```

Il metodo Create_midimessagearray crea una struttura, denominata midi_message, contenente istanze di tipo midi_message, che ha come campi i parametri utili ai fini della creazione dei messaggi MIDI, ovvero type (noteon o noteoff), part_number (il numero della parte, che corrisponderà a un canale MIDI da 1 a 16), timing (il timing assoluto, in millisecondi) e notenumber (il numero della nota, come richiesto dalle specifiche MIDI). Per creare la struttura midi_message, viene letta la struttura contenente le note (chiamata note) e, per ognuna di esse, vengono aggiunti alla struttura midi_message due messaggi, uno di tipo noteon, con timing corrispondente a quello della nota (nota.ms_timing) e uno di tipo noteoff, con timing corrispondente a quello della nota sommato alla durata in millisecondi della nota (ms_duration).

METODO xml_read()

```
public void xml_read() //metodo per avviare la lettura del file xml
            {
                event_timing();
                Create_midimessagearray();
            }
```

Il metodo xml_read() non è altro che un metodo che invoca i metodi event_timing() e Create_midimessagearray().

La lettura del file xml, quindi, avviene invocando due metodi:

- xml_initialize, che inizializza la lettura e a sua volta invoca i metodi CompileNoteArray() e CompileEventArray();
- xml_read, che invoca i metodi event_timing() e Create_midimessagearray().

La scelta di mantenere separata l'invocazione dei due gruppi di metodi è dettata dal fatto che l'utente, dopo aver premuto “stop”, potrebbe voler cambiare il bpm del brano, e sarebbe necessario reinvocare il metodo event_timing per ricalcolare tutti i valori assoluti di timing associati alle note, e quindi ricreare la struttura contenente le istanze di tipo midi_message attraverso l'invocazione del metodo Create_midimessagearray.

TIPI

Il namespace Xml_reading contiene anche la definizione dei quattro tipi usati per creare le strutture di parti, note, eventi e midi message, chiamati rispettivamente part, nota, evento, midi_message, ciascuno con i propri costruttori di default, in modo che alla creazione di un'istanza di ciascuno di questi tipi (che nel codice descritto in precedenza avviene all'atto dell'aggiunta di un oggetto a una struttura) sia possibile il passaggio dei parametri che verranno automaticamente assegnati ai campi di ciascun tipo.

TIPO part

```
public class part
{
        public string id;
        public int number;
        public int gm;


        public part (string id, int number)
        {
                this.id = id;
                this.number = number;
                this.gm = 0;
        }
}
```

TIPO nota

```
class nota
                        {

                                public string part_id;
                                public string chord_eventref;
                                public int part_number;
                                public int note_number;
                                public int duration_num;
                                public int duration_den;

                                public int octave;
```

```
                public char step;
                public string alteration;
                public double ms_timing;
                public double ms_duration;


        public nota (string part_id, string chord_eventref, int part_number, int duration_num, int duration_den, int octave, char step, string alteration, int note_number)
        {
                this.part_id = part_id;
                this.chord_eventref = chord_eventref;
                this.part_number = part_number;
                this.duration_num = duration_num;
                this.duration_den = duration_den;
                this.octave = octave;
                this.step = step;
                this.alteration = alteration;
                this.note_number = note_number;

        }

                }
```

TIPO evento

```
class evento
{
        public string id;
        public int relative_timing;
        public int absolute_timing;
        public double ms_timing;

        public evento(string id, int event_relativeTiming)
        {
                this.id = id;
                this.relative_timing = event_relativeTiming;

        }
}
```

TIPO midi_message

```
public class midi_message
{

        public string type;
        public int part_number;
        public int timing;
        public int note_number;

        public midi_message(string type, int part_number, int timing, int note_number)
        {
                this.type = type;
                this.part_number = part_number;
                this.timing = timing;
                this.note_number = note_number;
        }
}
```

3.3: Il file Main.cs

Portmidi

Per spiegare il funzionamento di questa parte di codice è necessario fare una premessa su come è stato modificato il codice del progetto di test di Portmidi, la cui compilazione produce un eseguibile da utilizzare per avviare il processo di output dei messaggi MIDI.

Il progetto di test di Portmidi, con le opportune modifiche, è stato utilizzato per la compilazione dell'eseguibile utilizzato per l'output dei messaggi in quanto le funzionalità necessarie erano già implementate. E' quindi bastato modificare il codice in modo che il passaggio dei parametri in modo testuale potesse essere gestito direttamente dal codice dell'applicazione, in modo semplice e veloce. Per avviare la riproduzione (e quindi, l'output dei messaggi MIDI) Portmidi richiede il passaggio di una struttura, con istanze di tipo PmEvent, che rappresentano i messaggi. Il tipo PmEvent ha due campi di nostro interesse:

- message, ovvero il messaggio MIDI, che può essere creato utilizzando la funzione PmMessage, con il passaggio dei tre parametri che corrispondono ai valori dei tre byte che costituiscono un messaggio MIDI, cioè:
    - status e channel number;
    - note number, oppure program number per i messaggi di program change;
    - velocity, oppure 0 per i messaggi di program change;
- timing, che costituisce la marcatura temporale in millisecondi per il messaggio contenuto nel campo message;

Una volta creata la struttura (che nel codice è denominata semplicemente buffer) contenente tutti i messaggi MIDI con la propria marcatura temporale, basterà invocare la funzione PmOpenOutput() passando il numero di device su cui inviare i messaggi (Portmidi esegue una scansione delle periferiche disponibili e le identifica assegnando a ciascuna periferica un numero):

```
Pm_OpenOutput (&midi,
        device_number,
```

```
        DRIVER_INFO,
        OUTPUT_BUFFER_SIZE,
        (latency == 0 ? NULL : TIME_PROC),
        (latency == 0 ? NULL : TIME_INFO),
        100);
```

e successivamente la funzione Pm_Write, passando la struttura contenente i messaggi da riprodurre:

```
Pm_Write(midi, buffer, index);
```

Il codice che si occupa della creazione della struttura è invece un ciclo while che, fino al raggiungimento dell'istruzione di break, continua a leggere i messaggi passati in formato testuale dall'utente (nel caso di applicazione console), oppure dal codice principale C#, attraverso la funzione

```
p.StandardInput.WriteLine();
```

```
while (index < 1000000)
    {
        int type = get_number("");

        int note = get_number("");
        int timing = get_number("");

        buffer[index].timestamp = timing;

        /*

         type 1x -> noteon channel x // notenumber // timestamp;
         type 2x -> noteoff channel x // notenumber // timestamp;
         type 9x -> program change channel x // program // 0;
         type 89 -> set MIDI device number // device number // 0;
         type 0 // 0 // 0 -> PLAY STREAM;


         */

        if (type == 101){buffer[index].message = Pm_Message(0x90, note, 100);} //noteon channel 1
        if (type == 201){buffer[index].message = Pm_Message(0x80, note, 100);} //noteoff channel 1
        if (type == 102){buffer[index].message = Pm_Message(0x91, note, 100);} //noteon channel 2
        if (type == 202){buffer[index].message = Pm_Message(0x81, note, 100);} //noteoff channel 2
        if (type == 103){buffer[index].message = Pm_Message(0x92, note, 100);} //noteon channel 3
        if (type == 203){buffer[index].message = Pm_Message(0x82, note, 100);} //noteoff channel 3
        if (type == 104){buffer[index].message = Pm_Message(0x93, note, 100);} //noteon channel 4
        if (type == 204){buffer[index].message = Pm_Message(0x83, note, 100);} //noteoff channel 4
        if (type == 105){buffer[index].message = Pm_Message(0x94, note, 100);} //noteon channel 5
        if (type == 205){buffer[index].message = Pm_Message(0x84, note, 100);} //noteoff channel 5
        if (type == 106){buffer[index].message = Pm_Message(0x95, note, 100);} //noteon channel 6
        if (type == 206){buffer[index].message = Pm_Message(0x85, note, 100);} //noteoff channel 6
        if (type == 107){buffer[index].message = Pm_Message(0x96, note, 100);} //noteon channel 7
        if (type == 207){buffer[index].message = Pm_Message(0x86, note, 100);} //noteoff channel 7
```

```
    if (type == 108){buffer[index].message = Pm_Message(0x97, note, 100);} //noteon channel 8
    if (type == 208){buffer[index].message = Pm_Message(0x87, note, 100);} //noteoff channel 8
    if (type == 109){buffer[index].message = Pm_Message(0x98, note, 100);} //noteon channel 9
    if (type == 209){buffer[index].message = Pm_Message(0x88, note, 100);} //noteoff channel 9
    if (type == 110){buffer[index].message = Pm_Message(0x99, note, 100);} //noteon channel 10
    if (type == 210){buffer[index].message = Pm_Message(0x89, note, 100);} //noteoff channel 10
    if (type == 111){buffer[index].message = Pm_Message(0x9A, note, 100);} //noteon channel 11
    if (type == 211){buffer[index].message = Pm_Message(0x8A, note, 100);} //noteoff channel 11
    if (type == 112){buffer[index].message = Pm_Message(0x9B, note, 100);} //noteon channel 12
    if (type == 212){buffer[index].message = Pm_Message(0x8B, note, 100);} //noteoff channel 12
    if (type == 113){buffer[index].message = Pm_Message(0x9C, note, 100);} //noteon channel 13
    if (type == 213){buffer[index].message = Pm_Message(0x8C, note, 100);} //noteoff channel 13
    if (type == 114){buffer[index].message = Pm_Message(0x9D, note, 100);} //noteon channel 14
    if (type == 214){buffer[index].message = Pm_Message(0x8D, note, 100);} //noteoff channel 14
    if (type == 115){buffer[index].message = Pm_Message(0x9E, note, 100);} //noteon channel 15
    if (type == 215){buffer[index].message = Pm_Message(0x8E, note, 100);} //noteoff channel 15
    if (type == 116){buffer[index].message = Pm_Message(0x9F, note, 100);} //noteon channel 16
    if (type == 216){buffer[index].message = Pm_Message(0x8F, note, 100);} //noteoff channel 16

    if (type == 901){buffer[index].message = Pm_Message(0xC0, note, 0);} //program change channel 1
    if (type == 902){buffer[index].message = Pm_Message(0xC1, note, 0);} //program change channel 2
    if (type == 903){buffer[index].message = Pm_Message(0xC2, note, 0);} //program change channel 3
    if (type == 904){buffer[index].message = Pm_Message(0xC3, note, 0);} //program change channel 4
    if (type == 905){buffer[index].message = Pm_Message(0xC4, note, 0);} //program change channel 5
    if (type == 906){buffer[index].message = Pm_Message(0xC5, note, 0);} //program change channel 6
    if (type == 907){buffer[index].message = Pm_Message(0xC6, note, 0);} //program change channel 7
    if (type == 908){buffer[index].message = Pm_Message(0xC7, note, 0);} //program change channel 8
    if (type == 909){buffer[index].message = Pm_Message(0xC8, note, 0);} //program change channel 9
    if (type == 910){buffer[index].message = Pm_Message(0xC9, note, 0);} //program change channel 10
    if (type == 911){buffer[index].message = Pm_Message(0xCA, note, 0);} //program change channel 11
    if (type == 912){buffer[index].message = Pm_Message(0xCB, note, 0);} //program change channel 12
    if (type == 913){buffer[index].message = Pm_Message(0xCC, note, 0);} //program change channel 13
    if (type == 914){buffer[index].message = Pm_Message(0xCD, note, 0);} //program change channel 14
    if (type == 915){buffer[index].message = Pm_Message(0xCE, note, 0);} //program change channel 15
    if (type == 916){buffer[index].message = Pm_Message(0xCF, note, 0);} //program change channel 16

    if (type == 89){device_number = note; index--;} //set device number

    if (type == 0 && note == 0 && timing == 0){break;};
    index++;

}
```

Come si può vedere dal codice, il primo parametro “letto” attraverso la funzione get_number è “type”. Type è una traduzione dello status byte in un numero di tre cifre, in cui la prima cifra indica il tipo del messaggio (1 = note on, 2 = noteoff), e le ultime due cifre indicano il canale a cui il messaggio si riferisce (da 01 a 16). Dopo aver “scritto” le prime tre cifre, il codice richiede la scrittura del notenumber, sempre costituito da tra cifre, e in seguito il passaggio della marcatura temporale (in millisecondi) da assegnare a quel messaggio.

Un'altra tipologia di messaggio, il program change, potrà essere creata utilizzando come prima cifra del parametro “type” il 9, e indicando successivamente il canale a cui il program change si riferisce. Nel caso di messaggio di tipo program change, il parametro corrispondente al notenumber

rappresenterà il numero corrispondente al timbro general MIDI da assegnare a quel canale, mentre il parametro timing sarà 0.
L'inserimento di un parametro type con valore “89”, avrà invece la funzione di impostare il “device number” (portmidi analizza tutte le periferiche MIDI disponibili e le identifica assegnandogli un numero progressivo). In questo caso, il parametro notenumber rappresenterà il numero che identifica la periferica scelta, mentre il parametro timestamp dovrà essere impostato a 0.

Dopo l'inserimento dei tre parametri, il codice successivo si occupa di valutare attraverso degli “if” il valore del parametro “type”, e in base a questo di riempire i campi “message” e “timing” dell'istanza di tipo PmEvent.
Il campo message conterrà un messaggio creato con la funzione Pm_Message, passando tre parametri: un esadecimale che rappresenta il tipo (noteon o noteoff) e il canale, il parametro notenumber e la velocity, che abbiamo deciso di impostare a 100 per tutti i messaggi, mentre il campo timing conterrà semplicamente la marcatura temporale in millisecondi ricevuta tramite la scrittura dal codice principale.
Messaggio per messaggio dovrà essere ripetuta questa procedura di inserimento dei tre parametri, che riempirà una per una le istanze della struttura, ciascuna corrispondente ad un messaggio. Al termine dell'inserimento di tutti i parametri corrispondenti ai messaggi da riprodurre, sarà possibile interrompere il ciclo while (invocando la funzione di break) e passare al codice successivo, che apre l'output e invia la struttura precedentemente creata (l'array buffer) alla libreria di output, dando di fatto inizio alla riproduzione, impostando tutti e tre i parametri a 0.

Metodi del file Main.cs

Ritornando al commento del file main.cs, possiamo notare che questo contiene l'implementazione di una serie di metodi, richiamabili dal file che contiene l'implementazione delle funzionalità dell'interfaccia grafica, che richiamano a loro volta i metodi per la lettura dell'xml e gestiscono il passaggio dei parametri dei messaggi MIDI al processo che gestisce l'output verso la periferica.

La prima parte di codice crea delle variabili utilizzate per la gestione della riproduzione, oltre che una nuova istanza della classe xml_reading, chiamata xml, sulla quale sarà possibile invocare tutti i metodi per la lettura dell'xml visti sopra. Inoltre, il codice iniziale crea un nuovo processo partendo dall'eseguibile generato dalla compilazione del progetto di test di Portmidi, opportunamente modificato secondo le nostre esigenze:

```
public static ProcessStartInfo portmidi = new ProcessStartInfo
                        {
                                FileName = "",
                                RedirectStandardInput = true,
                                UseShellExecute = false,
                                CreateNoWindow = true

                        } ;

                        public static Process p = Process.Start(portmidi);
```

"p" sarà il nome del processo all'interno del codice, mentre il comando RedirectStandardInput = true ci consentirà di dare in input delle stringhe di testo all'altro processo, utilizzando il comando

```
p.StandardInput.WriteLine();
```

METODO initialize()

questo metodo non fa altro che invocare il metodo Xml_initialize() del file contenente il codice per la lettura dell'xml, passando come parametro filename, ovvero il percorso del file da aprire.

METODO play_song()

```
public static void play_song(double initial_timing)
                {
                        playing = true;
                        p.StandardInput.WriteLine(89); //set device number
                        p.StandardInput.WriteLine(device_number); //device number = 0
                        p.StandardInput.WriteLine(0);

                        foreach (part part in xml.part_list)
                        {
                                p.StandardInput.WriteLine(900+part.number);
                                p.StandardInput.WriteLine(part.gm);
                                p.StandardInput.WriteLine(0);

                        }


                        foreach (midi_message message in xml.midi_message
                                .OrderBy(x => x.timing)
                                .Where (a => a.timing > initial_timing)
                                .Select (a => a))

                        {

                                if (message.type == "noteon") {p.StandardInput.WriteLine(100 +
message.part_number);};
                                if (message.type == "noteoff") {p.StandardInput.WriteLine(200 +
message.part_number);};

                                p.StandardInput.WriteLine(message.note_number);
```

```
                    p.StandardInput.WriteLine(message.timing + 100 - initial_timing);
                }

                p.StandardInput.WriteLine("0");
                p.StandardInput.WriteLine("0");
                p.StandardInput.WriteLine("0");

            }
```

Il metodo play_song si occupa del passaggio dei parametri al processo Portmidi e si può dividere in tre parti, che differiscono per la tipologia di messaggi che vanno a creare. Come già detto, la funzione utilizzata per la scrittura dei parametri è StandardInput.WriteLine(), e per creare un messaggio è necessario passare tre parametri:

- type;
- notenumber;
- timing;

La prima parte di codice serve a impostare il device number, impostando type a 89 e passando il device number anziché il notenumber, con timing 0.

La seconda parte, invece, attraverso un ciclo foreach, per ogni parte contenuta nella struttura part_list crata durante la lettura dell'XML, invia i tre parametri che permettono al codice di Portmidi di aggiungere alla struttura un messaggio di program change. Il secondo parametro scritto, in questo caso, sarà il numero di timbro Generalmidi da associare al canale specificato, contenuto nella variabile “gm”, il cui valore viene impostato nella parte dell'applicazione che gestisce l'interfaccia grafica, in base alla scelta operata dall'utente.

La terza parte di codice, infine, per ogni istanza (che rappresenta un messaggio midi) contenuta nella struttura midi_message, scriverà i tre parametri corrispondenti (type, notenumber e timing), nelle modalità supportate dal codice di Portmidi, in modo da aggiungere un messaggio alla struttura. La lettura della struttura midi_message attraverso il ciclo foreach avviene dopo aver ordinato le istanze in essa contenute attraverso un'istruzione di OrderBy, e rispettando una condizione:
verranno lette soltanto le istanze che hanno un valore del campo timing superiore al valore del parametro initial_timing passato al momento dell'invocazione della funzione play_song(). Questo permette di poter avviare la riproduzione dal punto desiderato, ed è necessario al fine di gestire correttamente la funzione di messa in pausa, in quanto ogni volta che viene avviata la riproduzione

(e quindi, anche dopo aver messo in pausa) sarà necessario riaprire il processo Portmidi, e ripassare tutti i parametri, creando una nuova struttura da passare alla libreria di output.

E' necessario sottolineare che al parametro timing contenuto nella struttura midi_message viene sottratto, nel caso in cui il valore di initial_timing sia diverso da 0, e quindi ne caso in cui il timing iniziale non sia 0, il valore di initial_timing, in modo che, pur iniziando la riproduzione da istanze con timing maggiore di 0 (ad esempio, dalla metà del brano), il timing dei messaggi inviati a Portmidi sia ricondotto a valori cha partono da 0, per garantire che la riproduzione inizi sempre all'istante attuale.

Terminata la scrittura di tutti i parametri relativi ai messaggi, vengono scritti tre “0” per segnalare che la scrittura di tutti i messaggi è stata completata e dare inizio all'output MIDI.

METODO play()

```
public static void play()
                {

                        if (stopped == false && paused == false && playing == false)
                        {
                        play_song(0);
                        play_ms = DateTime.Now;
                        }

                        if (stopped == true && paused == false && playing == false)
                        {
                        p.Start();
                        timing = 0;
                        play_ms = DateTime.Now;
                        stopped = false;
                        play_song(0);
                        }

                        if (paused == true && playing == false)
                        {
                                p.Start();
                                paused = false;
                                stopped = false;
                                play_song(timing);
                        }

                }
```

Il codice che si occupa della gestione delle funzionalità dell'interfaccia grafica (contenuto nel file MainWindow.cs), per avviare la riproduzione, non richiamerà direttamente la funzione play_song, ma invocherà il metodo play, che si occuperà di invocare a sua volta la funzione play_song, ma

passando il valore di initial_timing appropriato, dopo aver valutato se la riproduzione è stata avviata in seguito ad una messa in pausa, o ad uno stop. Possiamo notare che se la riproduzione non è successiva ad una messa in pausa, viene invocato play_song con initial_timing pari a 0, e viene impostato il valore di start della riproduzione (play_ms) con l'orario attuale (DateTime.Now), mentre se la riproduzione è successiva ad una messa in pausa, verrà invocato play_song con initial_timing pari al valore della variabile timing, che contiene il riferimento temporale dell'istante di messa in pausa.

METODO stop()

```
public static void stop()
        {
            if(stopped == false && paused != true)
            {
            p.Kill();
            stopped = true;
            }

            if(stopped == false && paused == true)
            {
            stopped = true;
            paused = false;
            }

            playing = false;
        }
```

Il metodo stop(), quando invocato, si limita a valutare le condizioni precedenti, e quindi a bloccare il processo portmidi con un comando di Kill (l'unico modo per arrestare la riproduzione una volta passata la struttura).

METODO pause()

```
public static void pause()
        {
            if(stopped != true)
            {
            p.Kill();
            paused = true;
            pause_ms = DateTime.Now;
            timing = (pause_ms.Subtract(play_ms).TotalMilliseconds);
            }

            playing = false;
        }
```

All'invocazione del metodo pause, il processo di output dei messaggi midi viene bloccato con un comando di kill, e viene impostato il valore della variabile timing al timing a cui la riproduzione è

giunta, ottenuto sottraendo all'orario attuale quello di inizio della riproduzione (play_ms), e ricavandone i millisecondi relativi. In questo modo, se dopo la messa in pausa verrà riavviata la riproduzione, sarà possibile invocare il metodo play_song passando come parametro initial_timing il valore della variabile timing, ossia il timing da cui far ripartire la riproduzione.

3.4: Il file MainWindow.cs

Il codice contenuto nel file MainWindow.cs si occupa di gestire tutti gli eventi generati dalle operazioni che l'utente effettua attraverso le funzionalità dell'interfaccia grafica. Non si ritiene necessario addentrarsi nella descrizione dettagliata di tutto il codice, anche perchè questa parte non contiene l'implementazione di algoritmi fondamentali per la lettura dell'xml o la creazione e l'output dei messaggi, ma si limita a richiamare con una certa logica le funzioni implementate in Xml_reading.cs e in Main.cs. Procediamo quindi ad un'essenziale descrizione dei gestori degli eventi contenuti in questa parte di codice:

OnDeleteEvent()

Contiene la descrizione delle operazioni da eseguire alla chiusura della finestra. In caso di chiusura, ad esempio, un comando di Kill chiuderà il processo relativo a Portmidi, che altrimenti continuerebbe a suonare.

OnFilechooserSelectionChanged()

Contiene il codice che gestisce la scelta e l'apertura del file xml, in seguito alla quale viene cancellato l'eventuale contenuto delle strutture e viene invocato il metodo initialize() di Xml_reading. Oltre a questo, dopo l'apertura del file, viene popolata la combobox con l'elenco di tutte le parti del brano (come da lettura dell'xml), per dare la possibilità all'utente di associare ad ognuna di esse un timbro Generalmidi (vedi gm_combobox_changed).

OnPlayButtonClicked, OnStopButtonClicked, OnPauseButtonClicked

Questi tre metodi si occupano di gestire rispettivamente l'avvio, l'arresto e la messa in pausa della riproduzione, invocando opportunamente i metodi play(), stop() e pause() del file Main.cs e gestendo l'avvio e l'arresto di un timer (label_timer) interno, che ha come scopo quello di dare

all'utente un'indicazione dell'avanzamento temporale della riproduzione. Se si tratta della prima volta che la riproduzione viene avviata, viene invocato il metodo xml_read di xml_reading, che si occupa di assegnare il timing alle note e popolare la struttura midi_message. L'invocazione di questo metodo non avviene contestualmente all'apertura del file, in quanto potrebbe essere necessario reinvocarlo per riassegnare i valori di timing, nel caso in cui l'utente, dopo una riproduzione, effettui un cambio del valore del bpm.

gm_combobox_changed, part_list_combobox_changed, assign_button_clicked

Questi metodi si occupano di gestire l'assegnamento, da parte dell'utente, del timbro Generalmidi ad ogni parte del brano, attraverso la scelta dei valori da due combobox e la pressione del bottone “assign”. Per ogni parte della struttura part_list, viene semplicemente assegnato un valore che va da 0 a 127 al campo gm dell'istanza part relativa a quella parte.

output_device_combobox_changed

L'interfaccia grafica dispone di una combobox che permette di selezionare la periferica su cui dare in output i messaggi midi, scegliendo un numero da 0 (default) a 9. All'atto della selezione, viene aggiornato il valore della variabile device_number di Main.cs.

timing_entry_changed

Questo evento viene invocato al cambiamento del valore di bpm da parte dell'utente, e provvede ad aggiornare il valore della variabile string_bpm di Xml_reading. Inoltre, la variabile first_read viene impostata a true, in modo che alla pressione del tasto play in seguito al cambiamento di bpm venga reinvocato il metodo xml_read, per calcolare nuovamente tutti i valori di timing in base al nuovo valore di bpm.

Tick

Tick è il gestore dell'evento che viene invocato allo scadere dell'intervallo del timing interno a questo file label_timer. Ogni 100 millisecondi viene aggiornato il valore della label del timer per dare un riscontro, seppur abbastanza approssimativo in quanto non derivato direttamente da Portmidi, dell'istante attuale a cui è giunta la riproduzione.

Inoltre, quando il tempo del timer raggiunge il valore di timing massimo del brano (max_timing) più un secondo, il processo (che altrimenti resterebbe aperto) viene chiuso, e tutte le variabili per la gestione della riproduzione vengono reimpostate ai valori iniziali, per consentire eventualmente di avviare nuovamente la riproduzione, attraverso la pressione del pulsante play.

PARTE 4: CONCLUSIONI E SVILUPPI FUTURI

L'applicazione ottenuta tramite la scrittura del codice illustrato precedentemente è funzionante e stabile, sia in ambiente Windows, che in ambiente Mac. Mentre il codice principale può essere indifferentemente compilato con Mono su qualsiasi piattaforma, un discorso a parte vale per la libreria Portmidi, che va compilata per intero, ovviamente sostituendo il file test.c della directory Portmidi/pm_test originale con quello modificato, utilizzando sistemi di compilazione diversi a seconda della piattaforma (visual studio per Windows, cmake/xcode per Mac, cmake per Linux). Mentre la compilazione di Pormidi su Windows e Mac non ha creato particolari problemi (se non errori dovuti alla localizzazione dei file, risolti cambiando i relativi percorsi all'interno del progetto principale), non è stato fino ad ora possibile compilare Portmidi in ambiente Linux (la compilazione, anche se effettuata seguendo le istruzioni date dal creatore del codice, genera un errore generico), e questo è l'unico motivo per il quale non è possibile utilizzare il player in ambiente Linux.

Oltre a questa limitazione, il fatto di dover utilizzare una libreria esterna al framework .NET per gestire l'output dei messaggi MIDI, controllato da un processo esterno a quello principale, è la ragione dell'utilizzo di alcuni “artifici” per gestire lo stop e la ripresa in seguito alla messa in pausa (lo stop e la messa in pausa vengono ottenuti interrompendo il processo, e la ripresa dopo la pausa avviene riaprendo il processo e ripassando nuovamente tutti i parametri). Questo, ovviamente, può avvenire a scapito della fluidità e della precisione dell'esecuzione.

Per i motivi sopracitati, oltre che per dare la possibilità di integrare quanto realizzato all'interno di un software più completo, la parte di lettura dell'xml è stata volutamente mantenuta separata da tutto il resto: tramite il codice contenuto nel file xml_reanding.cs è possibile aprire il file xml, leggere e convertire il suo contenuto ed ottenere come risultato la compilazione di una struttura contenente delle istanze, che corrispondono ai messaggi MIDI necessari, e i cui campi corrispondono ai parametri necessari per la creazione di questi messaggi (tipo, note number, part number e marcatura temporale assoluta in millisecondi). Avendo a disposizione questa parte che realizza il lavoro di lettura e conversione in formato midi-alike dell'xml, un futuro sviluppo potrebbe essere quello di realizzare una libreria .NET per l'output dei messaggi MIDI, che sia utilizzabile in multipiattaforma (non faccia quindi utilizzo delle dll di windows), in modo da creare un'unica applicazione, le cui parti di lettura dell'xml e gestione dell'output dei messaggi MIDI siano perfettamente integrate in un unico processo.

Un ulteriore futuro sviluppo potrebbe riguardare la gestione dell'assegnazione della corrispondenza parte – timbro GM. Nell'applicazione realizzata, la scelta è lasciata all'utente, ma si potrebbe realizzare del codice che provi ad impostare la corrispondenza in automatico, magari esaminando il nome della parte nello staff_list del file xml.

Ovviamente, l'unico scopo del codice di lettura e conversione dell'xml creato non è quello di dare in output dei messaggi MIDI. E' infatti possibile sfruttare il contenuto della struttura creata dal codice xml_reading per compilare, seguendo le specifiche, il layer performance del file xml. A tale scopo, sarà necessario poter riferire ogni messaggio a un tag event dello spine, e per fare questo sarà necessario modificare il codice del file xml_reading aggiungendo un campo "event_ref" al tipo midi_message, a cui verrà assegnato, all'atto della creazione della struttura midi_message tramite la lettura del contenuto della stuttura "note" nel metodo Create_midimessagearray, il valore del campo chord_eventref dell'istanza nota, corrispondente all'id dell'evento a cui la nota è collegata. Ecco come potrebbe essere modificato il codice (in grassetto le parti aggiunte):

```
public void Create_midimessagearray()
{
                        foreach (nota nota in note)
                        {
                                midi_message.Add(new midi_message("noteon", nota.chord_eventref,
nota.part_number, Convert.ToInt32(nota.ms_timing), Convert.ToInt32(nota.note_number)));
                                midi_message.Add (new midi_message("noteoff", nota.chord_eventref,
nota.part_number, Convert.ToInt32(Convert.ToInt32(nota.ms_timing)+nota.ms_duration), nota.note_number));

                        }

}

public class midi_message
        {

                public string type;
                public string event_ref;
                public int part_number;
                public int timing;
                public int note_number;

                public midi_message(string type, string chord_eventref, int part_number, int timing, int
note_number)
                {
                        this.type = type;
                        this.event_ref = chord_eventref;
                        this.part_number = part_number;
                        this.timing = timing;
                        this.note_number = note_number;
                }
        }
```

In questo modo, ogni istanza della struttura midi_message avrà anche un campo event_ref, corrispondente all'id dell'evento a cui il messaggio MIDI che l'istanza rappresenta si collega. Questo permetterà di leggere la struttura per creare all'interno del layer performance degli elementi corrispondenti ai messaggi MIDI che rappresentano il brano a livello di performance, riferiti agli elementi event dello spine, come vuole la specifica del formato IEEE1599.

PARTE 5: Bibliografia

- Joseph Albahari, Ben Albahari, C# 4.0 in a Nutshell, Fourth Edition - O'Reilly 2010 ISBN: 978-0-596-80095-6;
- Robert Guérin, MIDI. L'interfaccia digitale per gli strumenti musicali - Apogeo 2003 ISBN: 88-503-2078-7;
- IEEE 1599 DTD (standards.ieee.org/downloads/1599/1599-2008/ieee1599.dtd);
- Ludovico, L. A., “Key concepts of the IEEE 1599 Standard.” In Proceedings of the IEEE CS Conference The Use of Symbols To Represent Music And Multimedia Objects, ed. Denis Baggi and Goffredo Haus, 15–26. Lugano: IEEE CS, 2008. ISBN: 88-7595-010-5;
- Eliotte Rusty Harold - W. Scott Means, XML In a Nutshell, the definitive reference – O’Reilly 2004 ISBN: 978-0-596-00058-5.