



1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

CORSO DI LAUREA IN
INFORMATICA APPLICATA
SCUOLA DI
SCIENZE TECNOLOGIE E FILOSOFIA DELL'INFORMAZIONE

Programmazione ad Oggetti ed Ingegneria del Software

Sessione invernale 2019/2020

Nicolas Lazzari
Matricola: 284721

Matteo Di Lorenzi
Matricola: 284700

21 gennaio 2020

Made with L^AT_EX

Indice

1	Specifica del problema	1
2	Specifica dei requisiti	3
2.1	Diagramma dei casi d'uso	3
2.2	Descrizione casi d'uso	4
3	Analisi e progettazione	8
3.1	Architettura	8
3.1.1	Model	8
3.1.2	View	9
3.1.3	Controller	10
3.1.4	Utils ed Eccezioni	11
3.1.5	Versioning	11
3.1.6	Librerie esterne	11
3.2	Diagramma di robustezza	12
3.3	Diagramma UML delle classi	14
3.4	Model	15
3.5	Controller	18
3.6	View	19
4	Implementazione	21
4.1	Model	21
4.2	Controller	35
4.3	View	39
4.4	Exception	75
4.5	Utils	76
4.6	Program	79
5	Testing	80
5.1	Inizio di una partita	81
5.2	Visualizzazione punteggi	82
5.3	Impostazione di nome, modalità di gioco e BPM iniziali	83
5.4	Effettuazione colpo	84
5.5	Fine della partita	84
5.6	Partita in pausa	85
5.7	Partita ripresa da una pausa	85
5.8	Partita abbandonata da una pausa	86
6	Compilazione ed esecuzione	87

1 Specifica del problema

Si richiede di realizzare un videogioco di tipo musicale ispirato alla saga di successo Guitar Hero in cui l'utente impersona un musicista. Tale videogioco avrà però come strumento utilizzato solamente il rullante (il principale strumento percussivo in una batteria). Tale scelta è guidata dal fatto che il videogioco, oltre a sfidare l'utente così da renderne competitivo l'utilizzo, potrà essere utilizzato per imparare e migliorare la propria tecnica sullo strumento.

Il videogioco sarà composto da una schermata iniziale in cui l'utente potrà decidere se iniziare una nuova partita, visualizzare i risultati ottenuti dagli utenti del videogioco o chiudere il programma.

Nel caso in cui l'utente voglia iniziare una nuova partita, verrà reindirizzato nell'opportuna schermata dove verrà richiesto l'inserimento del suo nome (necessario per poter poi memorizzare il punteggio totalizzato della partita), la modalità alla quale si desidera giocare e la velocità di esecuzione iniziale (espressa in *BPM*, battiti per minuto).

Le modalità implementate nel videogioco sono due:

- nella prima modalità, chiamata *combinazioni casuali*, le note sono generate, molto intuitivamente, in maniera casuale, senza nessuna forzatura sul numero di note standard o speciali o sulla loro posizione;
- nella seconda modalità, chiamata *mani alternate*, le note generate sono alternate, ovvero se una nota è destra, la successiva sarà sinistra, e così via.

Una volta compilati opportunamente i campi della schermata di una nuova partita, verrà mostrata la schermata di gioco con la quale l'utente interagirà. Essa sarà composta da un rullante e due bacchette inizialmente alzate. Alla pressione del tasto **c** e **n** le bacchette (rispettivamente sinistra e destra) reagiranno all'input dell'utente colpendo il rullante. La sequenza di colpi da effettuare verrà rappresentata da immagini rappresentanti note che si muovono lungo due linee che congiungono i due angoli superiori dello schermo sino al rullante. Nel momento in cui il punto raggiunge il rullante e in contemporanea viene effettuato il colpo allora esso verrà conteggiato come corretto.

Il videogioco conterrà un semplice sistema di calcolo del punteggio totalizzato dall'utente. Tale punteggio sarà attribuito in base alla precisione del colpo.

Nel caso l'utente colpisca il rullante esattamente nell'istante in cui viene richiesto allora aumenterà il proprio punteggio di 100 punti (colpo perfetto). Nel caso in cui il colpo non sia precisamente nella posizione segnalata (colpo standard), ogni $2ms$ di ritardo o di anticipo (rispetto al colpo perfetto) risulterà in una penalità di 1 punto alla quantità che verrà aggiunta al punteggio totale. Nel caso l'utente colpisca il rullante quando il colpo non è ancora giunto sul rullante (colpo errato) non verrà attribuito alcun punteggio, dopo 20 errori di questo tipo la partita termina.

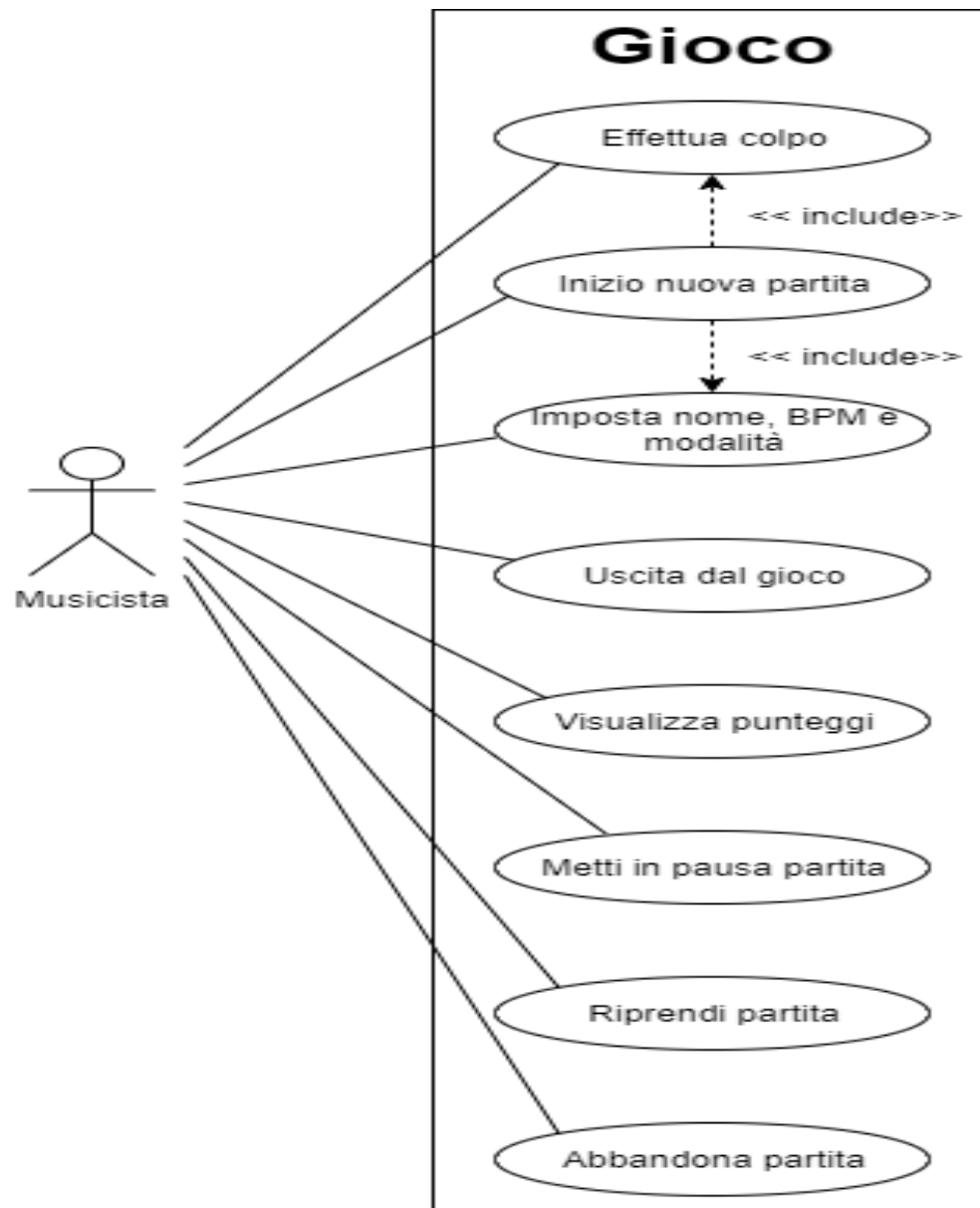
Durante la partita apparirà, in maniera casuale, al posto di un colpo standard un colpo speciale identificato diversamente dagli altri. Il punteggio assegnato nel caso in cui tale colpo venga eseguito in maniera corretta sarà doppio.

La difficoltà del gioco è determinata dal numero di BPM che si sta affrontando. Essi aumentano progressivamente ogni 5 colpi corretti (perfetti o standard) effettuati.

A fine partita il punteggio verrà mostrato all'utente e inserito in un file `csv` contenente tutti i punteggi effettuati con il gioco. Tali punteggi saranno visualizzabili a partire dalla schermata principale tramite un'apposita schermata raggiungibile premendo il tasto dedicato.

2 Specifica dei requisiti

2.1 Diagramma dei casi d'uso



2.2 Descrizione casi d'uso

Caso d'uso	Inizia partita
Id	1
Attori	Musicista
Pre-condizioni	N/A
Eventi base	L'utente avvia la partita cliccando sull'apposito pulsante
Post-condizioni	Viene mostrata la finestra di inserimento nome, inserimento BPM iniziali e selezione della modalità
Percorsi alternativi	N/A

Caso d'uso	Visualizza punteggi
Id	2
Attori	Musicista
Pre-condizioni	N/A
Eventi base	L'utente dopo aver cliccato sull'apposito pulsante, consulta i punteggi di gioco descritti dal punteggio numerico e dal nome dell'utente che ha totalizzato tale punteggio
Post-condizioni	N/A
Percorsi alternativi	N/A

Caso d'uso	Imposta nome, modalità e BPM iniziali
Id	3
Attori	Musicista
Pre-condizioni	L'utente deve aver avviato la partita
Eventi base	L'utente inserisce il proprio nome, i BPM iniziali e la modalità di gioco tra quelle proposte
Post-condizioni	Il gioco inizia
Percorsi alternativi	N/A

Caso d'uso	Effettua colpo
Id	4
Attori	Musicista
Pre-condizioni	La partita deve essere iniziata e devono essere state impostate le condizioni di gioco (nome, BPM iniziali e modalità)
Eventi base	L'utente effettua i colpi che appaiono sullo schermo tramite i due tasti impostati
Post-condizioni	N/A
Percorsi alternativi	N/A

Caso d'uso	Uscita dal gioco
Id	5
Attori	Musicista
Pre-condizioni	N/A
Eventi base	L'utente esce dal gioco premendo l'apposito pulsante proposto nel menù iniziale
Post-condizioni	Il gioco viene chiuso e il processo terminato
Percorsi alternativi	Abbandonare la partita in corso tramite il menù di pausa e poi uscire dalla partita con l'apposito pulsante

Caso d'uso	Metti in pausa partita
Id	6
Attori	Musicista
Pre-condizioni	La partita deve essere iniziata
Eventi base	L'utente, durante la partita, ha la possibilità di mettere in pausa il videogioco premendo il tasto <i>ESC</i> della tastiera. A quel punto avrà la possibilità di riprendere la partita oppure di abbandonarla, tornando al menù iniziale
Post-condizioni	Viene mostrato il menù di pausa del gioco
Percorsi alternativi	N/A

Caso d'uso	Riprendi partita
Id	7
Attori	Musicista
Pre-condizioni	Il gioco deve essere in pausa
Eventi base	L'utente, dopo aver messo in pausa la partita, premendo l'apposito tasto riprende la partita, che ricomincia nello stesso stato in cui era precedentemente
Post-condizioni	La partita riprende
Percorsi alternativi	N/A

Caso d'uso	Abbandona partita
Id	8
Attori	Musicista
Pre-condizioni	Il gioco deve essere in pausa
Eventi base	L'utente, dopo aver messo in pausa la partita, schiacciando l'apposito tasto abbandona la partita. In questo caso il punteggio della partita non viene salvato.
Post-condizioni	La partita termina e viene presentato il menù iniziale
Percorsi alternativi	N/A

3 Analisi e progettazione

3.1 Architettura

Per lo sviluppo del gioco è stato utilizzato il design pattern MVC che permette la separazione della logica del gioco dal codice che si occupa della presentazione all'utente.

Il pattern consiste in 3 componenti principali:

- **Model** che implementa la logica di business del programma tramite le classi che espongono metodi utili ad accedere e manipolare i dati
- **View** che implementa l'interfaccia con cui l'utente interagirà. Ogni modifica che richiede l'intervento sui dati del `model` passerà attraverso il `controller`.
- **Controller** che si occupa di veicolare i messaggi inseriti dall'utente nella `view` per manipolare i dati contenuti nel `model`. Funge da tramite da `model` e `view` in quanto, sebbene la `view` sia in grado di leggere dati dal `model`, essa non è in grado di manipolarli.

Nell'architettura del progetto è stato fatto largo uso di quelle tecniche che contraddistinguono un linguaggio ad oggetti come **C#**: l'*incapsulamento*, il *polimorfismo*, l'*ereditarietà*, l'utilizzo di *eccezioni* e l'utilizzo di *classi generiche*.

3.1.1 Model

La struttura del `model` ruota interamente attorno alle *note* (ossi i colpi che deve effettuare l'utente), come esse vengono generate e il mantenimento dei punti quando esse vengono colpite.

Ogni nota estende la classe astratta `INote` così che, tramite il meccanismo di *upcasting* sia possibile riferirsi ad una generica classe che rappresenta una nota, la quale implementerà i metodi che la contraddistinguono. Ogni nota in particolare è dotata di un punteggio che viene assegnato all'utente qualora esso la colpisce in modo perfetto, un'immagine da mostrare a schermo all'utente e la posizione a schermo (*destra* o *sinistra*).

La sequenza di note che l'utente deve colpire viene generata da un *generatore di note*. Ogni generatore di note estende la classe astratta `INoteGenerator` che utilizza il design pattern **Observer**. Tale design pattern permette la comunicazione agli *observer* delle nuove note generate. La generazione di note

avviene tramite l'utilizzo del metodo astratto `NextNote` la cui implementazione viene delegata all'utente. Il metodo `NextNote` viene utilizzato da una routine interna che contiene il codice eseguito da un thread separato rispetto al flusso di esecuzione principale. Tale thread si occupa di generare le note con la giusta cadenza temporale, basandosi sui *BPM* raggiunti dall'utente.

Il punteggio dell'utente viene mantenuto nella classe `Game` che implementa l'interfaccia `IGame`. Tale classe espone un metodo utilizzato per comunicare che l'utente ha effettuato un colpo a vuoto ed un metodo per comunicare che l'utente ha colpito una certa nota con un determinato ritardo rispetto al colpo perfetto. Tale classe inoltre si occupa di calcolare il punteggio totalizzato dall'utente e di generare un'eccezione di tipo `GameEndedException` nel caso l'utente giunga alla condizione di fine gioco (20 colpi effettuati troppo presto o troppo tardi). La classe `Game` si occupa inoltre di serializzare il punteggio dell'utente nell'apposito file `record.csv` e di deserializzare i record precedenti.

3.1.2 View

La struttura base della componente `View` è implementata interamente nella classe `MainView`. Tale classe, che implementa l'interfaccia `IMainView`, espone i metodi necessari per mostrare all'utente le schermate che compongono il gioco.

Ogni schermata di gioco è implementata tramite la classe `Panel` fornita dal framework `WinForms` e si occupa di gestire in modo autonomo i propri elementi grafici ed il proprio stato. Nel caso sia necessario, tale schermata può comunicare con `MainView` contenendone un riferimento e, nel caso di `PlayingPanel` anche con `MainController`.

Particolare enfasi va posta sulla classe `PlayingPanel`: tale classe infatti si occupa di implementare in modo vero e proprio la schermata di gioco vista dall'utente. Essa, implementando l'interfaccia `IObserver`, utilizza in modo diretto la classe `INoteGenerator` e mostra le nuove note generate all'utente inserendole in una lista generica di tipo `LinkedList` cosicchè sia possibile accedere all'elemento di testa e di coda.

Nel momento in cui un tasto viene premuto comunica in modo diretto a `MainController` dell'avvenuto colpimento di una nota o del colpo a vuoto.

Tale schermata inoltre mostra il punteggio raggiunto dall'utente, i colpi che l'utente può mancare prima che il gioco termini ed i *BPM* raggiunto.

3.1.3 Controller

La funzione di `controller` viene implementata dalla classe `MainController` che implementa l'interfaccia `IController`. Tale classe espone tutti i metodi necessari per conoscere le informazioni contenute nella classe `Game` ed interagire con i metodi presenti nell'interfaccia `IGame`.

3.1.4 Utils ed Eccezioni

Durante lo sviluppo del gioco si sono rese necessarie lo sviluppo di classi che svolgono funzioni che non sono direttamente imputabili a nessuna delle componenti del pattern MVC. Tali classi sono quelle necessarie allo sviluppo del pattern `Observable`, la rotazione di immagini, l'utilizzo di una tupla di 3 elementi editabile ecc.

Discorso del tutto analogo per le eccezioni che sono state create *ad-hoc*.

3.1.5 Versioning

Lavorando in un team di 2 persone si è reso necessario l'utilizzo di un tool di versioning per permettere la sincronizzazione del codice ed evitare problemi di conflitti.

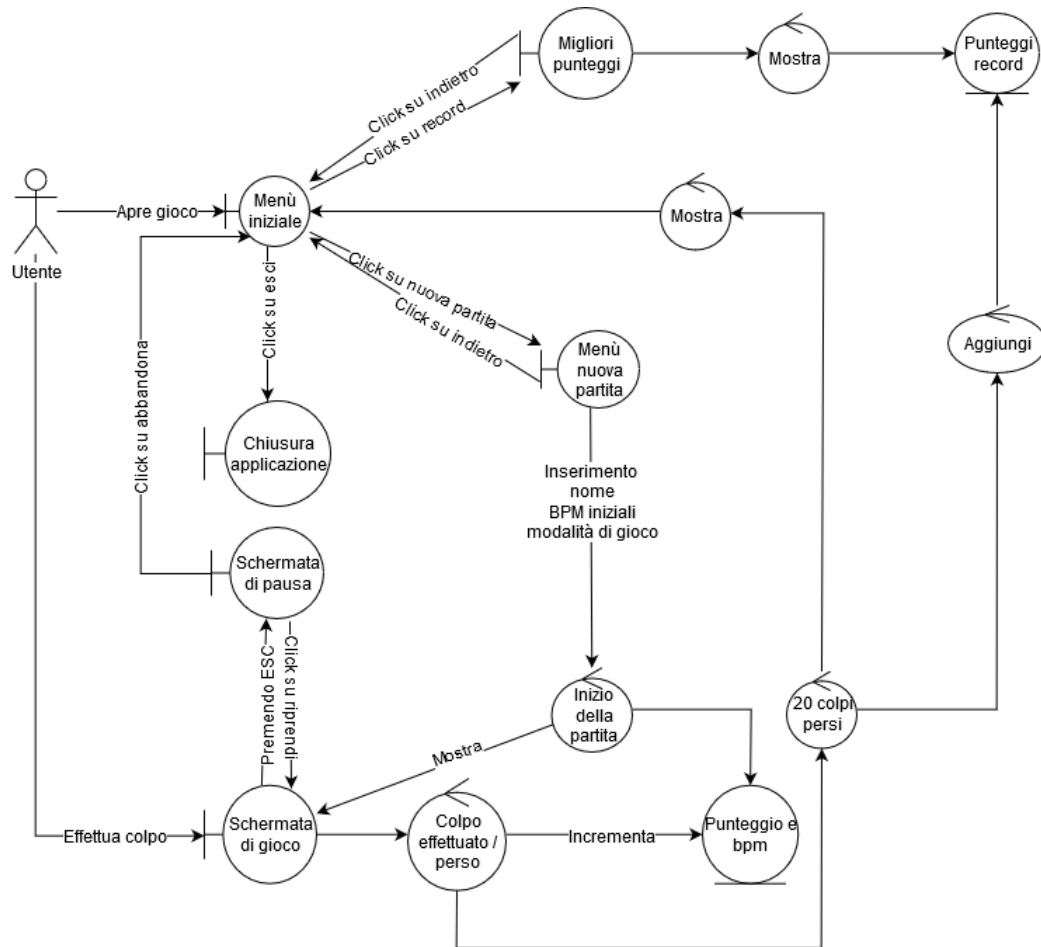
Si è fatto quindi largo uso del software di versioning `git` e della piattaforma github soprattutto grazie all'eccellente integrazione in *Visual Studio* tramite l'apposita estensione. L'intero software è reperibile all'indirizzo `n28div/masterdrums`.

3.1.6 Librerie esterne

Durante l'implementazione del gioco è stata utilizzata la libreria `NAudio` per riprodurre il suono del colpo sul rullante. Tale libreria è opensource ed installabile tramite `nuget`.

3.2 Diagramma di robustezza

Per illustrare il meccanismo di interazione tra i vari elementi del software, viene illustrato un diagramma di robustezza.



All'avvio del gioco l'utente potrà eseguire una delle seguenti operazioni:

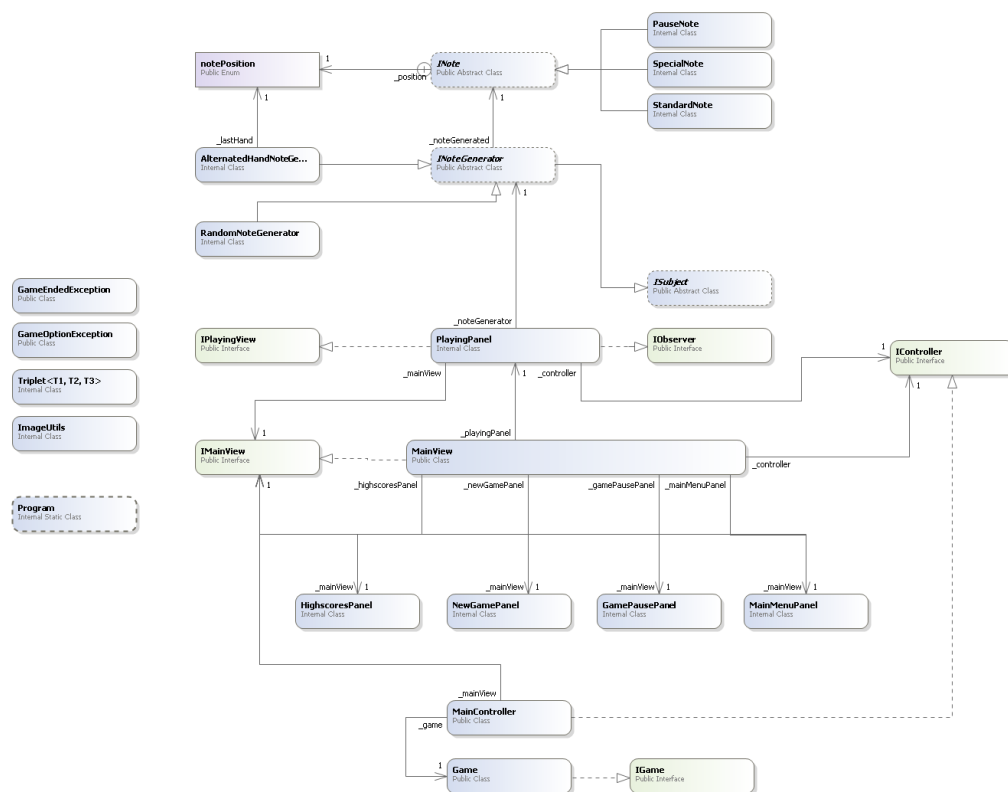
- Visualizzare i record totalizzati dagli utenti nel gioco
- Chiudere il gioco
- Iniziare una nuova partita

Nel caso in cui l'utente decida di iniziare una nuova partita, dopo aver opportunamente inserito il nome, i BPM iniziali e scelto la modalità di gioco, verrà mostrata la schermata di gioco effettiva. Da tale schermata di gioco, durante l'esecuzione della partita premendo il tasto **ESC** della tastiera, l'utente avrà la possibilità di mettere in pausa la partita e di conseguenza di riprendere la partita o di abbandonarla, tornando al menù iniziale.

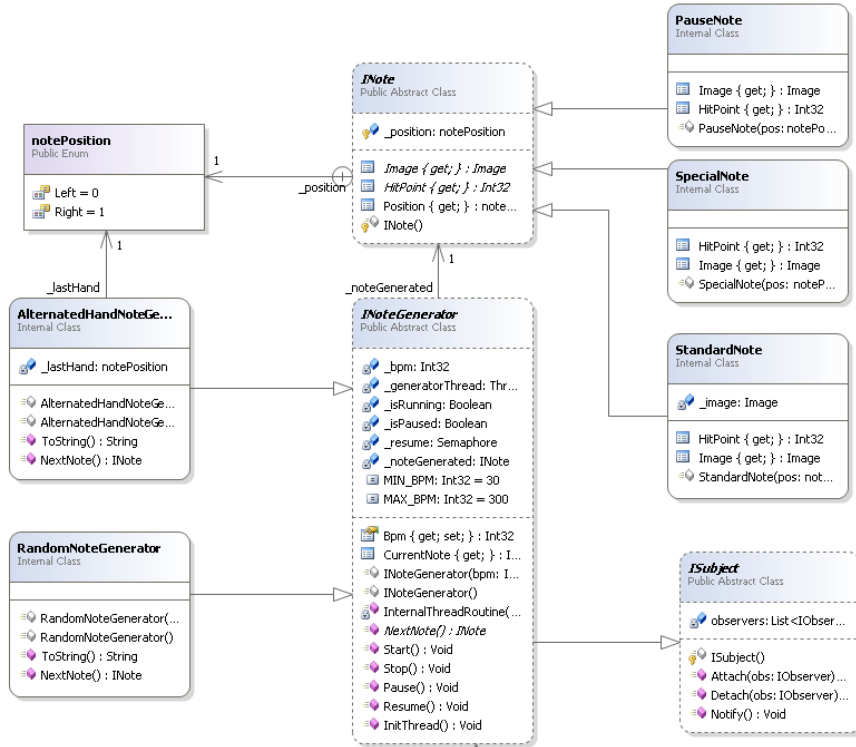
Se l'utente non abbandona la partita, ma essa termina tramite la condizione di fine gioco, il punteggio totalizzato viene memorizzato nell'apposito file `.csv`.

Nel caso in cui l'utente voglia visualizzare i record del gioco, verrà mostrata una schermata contenente i punteggi totalizzati dagli utenti, nel formato **Nome Punteggio**. Nel caso in cui non sia ancora stata effettuata la prima partita del gioco, la schermata non verrà mostrata e l'utente verrà opportunamente avvisato della mancanza di risultati.

3.3 Diagramma UML delle classi



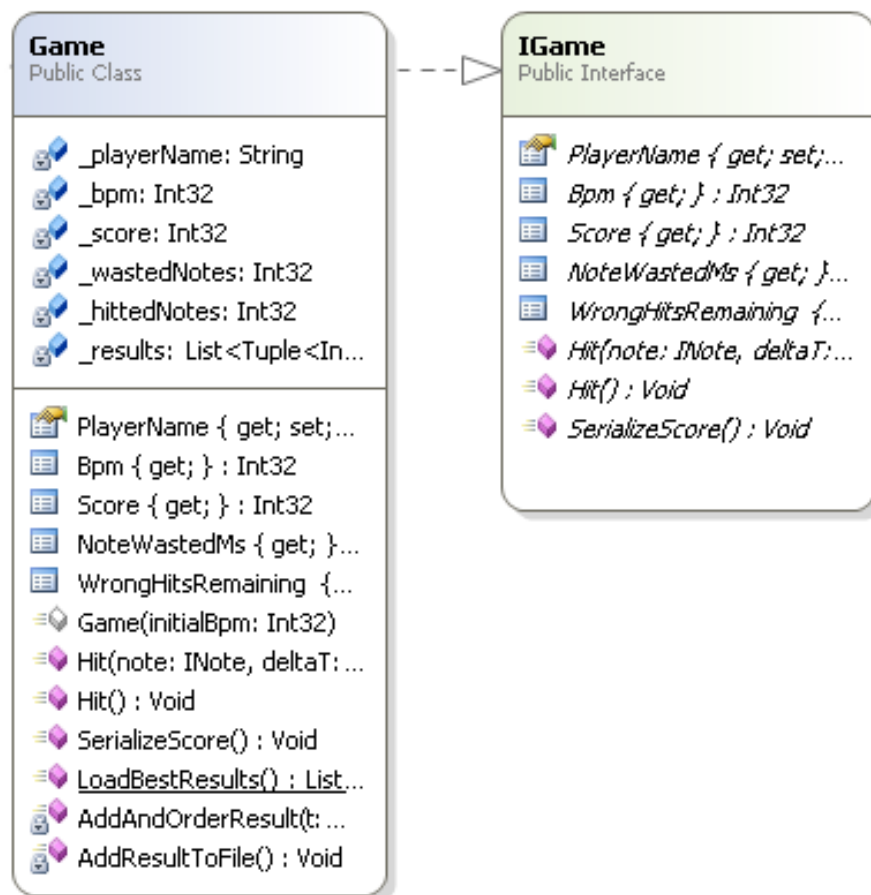
3.4 Model



Nell'immagine posta sopra, viene descritta l'architettura mediante schema UML della parte riguardante le note e il generatore di note. Come visto nella specifica del problema, le modalità proposte dal gioco sono due, e dipendono dal modo in cui vengono generate le note. Nella modalità *combinazioni casuali* le note vengono generate in maniera puramente casuale, senza nessun controllo sulla loro posizione, mentre nella modalità *mani alternate* vengono generate note con posizione (destra e sinistra) alternate tra loro. Questo avviene grazie ai due generatori di note descritti nell'immagine sopra riportata: entrambi i generatori di note implementano la classe astratta `INoteGenerator`, ed eseguono l'override dei due metodi `ToString()` e `NextNote()`. Molta attenzione va posta in particolare nel metodo `NextNote()`, in quanto nella modalità *mani alternate* la prossima nota dovrà per forza essere nella posizione opposta rispetto all'ultima nota generata.

È importante inoltre notare come `INoteGenerator` estende la classe astratta `ISubject`, necessaria per rendere il generatore di note come l'oggetto osservato dai vari observer, come descritto nell'architettura del design pattern `Observer`.

Per quanto riguarda le note invece, si osservi come esse risultino tutte derivanti dalla classe astratta `INote`, della quale eseguono l'override dei metodi `HitPoint()` e `Image()`. Ciascuna tipologia di nota avrà infatti una sua caratterizzazione per quello che riguarda il punteggio e l'immagine che la rappresenterà nel pannello di gioco.

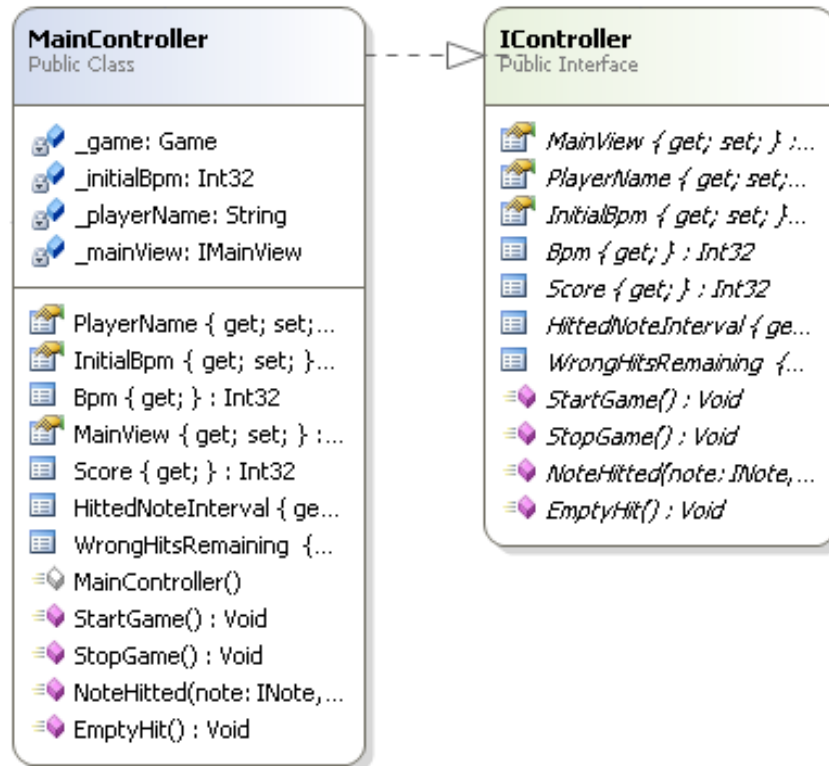


La classe **Game** è la classe che si occupa di mantenere il punteggio dell'utente, di aumentare i BPM e di salvare il punteggio nel file `csv`. Come si può notare dal diagramma UML tale classe implementa l'interfaccia **IGame** e ne

effettua l'*override* dei metodi `Hit()`, che rappresenta il metodo che comunica al gioco un colpo vuoto, `Hit(note, deltaT)` che rappresenta il colpo andato a segno sulla nota `note` dopo un tempo di `deltaT ms` dal tempo di colpo perfetto ed infine il metodo `SerializeScore` che si occupa di salvare il punteggio nel file.

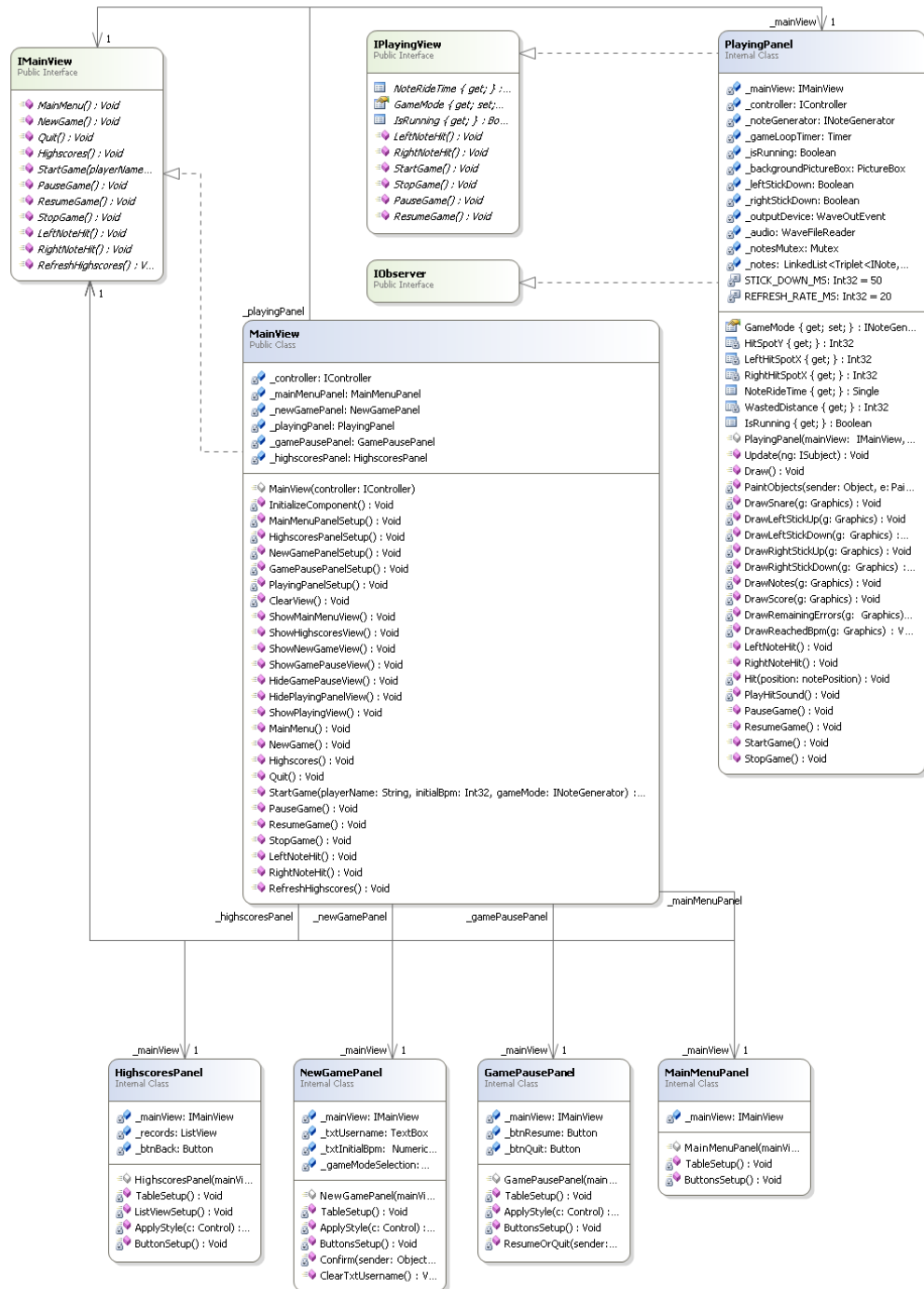
La classe `Game` inoltre espone il metodo statico `LoadBestResults` che si occupa di leggere dal file `csv` i punteggi effettuati precedentemente.

3.5 Controller



La struttura del controller è piuttosto semplice: la classe concreta **MainController** implementa l'interfaccia **IController** che espone metodi utili ad interagire con le classi che implementano l'interfaccia **IGame**. La classe **MainController** contiene inoltre al suo interno un riferimento alla *view principale* (**MainView**) così da poter comunicare di reagire a determinati eventi.

3.6 View



La componente *view* del progetto è principalmente gestita dalla classe concreta **MainView** che implementa l'interfaccia **IMainView** e funge da "*controller*" per le altre viste, implementate sotto forma di **Panel**. La classe **MainView** espone metodi per attivare o nascondere le viste che compongono il gioco (menù iniziale, schermata di pausa ecc.) e contiene metodi privati per creare ed inizializzare tali viste di modo che le dimensioni di esse siano dipendenti dalla dimensione dello schermo dell'utente (il gioco viene presentato a schermo intero).

Particolare enfasi va posta sulla classe concreta **PlayingPanel**: tale classe infatti rappresenta un *Observer* della classe **INoteGenerator**. Al suo interno inoltre è presente il metodo **PaintObjects(object sender, PaintEventArgs e)** che viene chiamato periodicamente e si occupa di disegnare a schermo gli *sprite* che compongono il gioco. Il pannello di gioco infatti gestisce tutti gli elementi grafici disegnandoli in una **PictureBox** che copre l'intero schermo dell'utente. Così facendo è possibile gestire in maniera semplice ed efficiente la sovrapposizione di oggetti sullo schermo.

Di particolare importanza è anche il metodo **DrawNotes** che tramite trasformazioni geometriche e trigonometriche si occupa di disegnare a schermo le note e di calcolare la loro posizione futura.

4 Implementazione

4.1 Model

```
1  using System.Drawing;
2
3  namespace MasterDrums.Model
4  {
5      /// <summary>
6      /// Abstract class that represents a base note
7      /// </summary>
8      public abstract class INote
9      {
10         /// <summary>
11         /// Enumerator for the note position (left or right).
12         /// </summary>
13         public enum notePosition
14         {
15             Left,
16             Right
17         }
18
19         protected notePosition _position;
20
21         /// <summary>
22         /// The image displayed for the note
23         /// </summary>
24         /// <returns>
25         /// The image instance
26         /// </returns>
27         public abstract Image Image {
28             get;
29         }
30
31         /// <summary>
32         /// The points gained by the user when a perfect hit is
33         ///   performed
34         /// </summary>
35         /// <returns>
36         /// The hit points
37         /// </returns>
38         public abstract int HitPoint
39         {
40             get;
41         }
42         /// <summary>
```



```

43     /// The note position on the screen (left or right)
44     /// </summary>
45     /// <returns>
46     /// The note position
47     /// </returns>
48     public notePosition Position
49     {
50         get => this._position;
51     }
52
53 }
54 }

```

```

1  using System;
2  using System.Drawing;
3
4  namespace MasterDrums.Model
5  {
6      /// <summary>
7      /// Class that extends the INote abstract class and represents a
8      /// standard note
9      /// </summary>
10     class StandardNote : INote
11     {
12         private Image _image = null;
13
14         public StandardNote(notePosition pos)
15         {
16             this._position = pos;
17         }
18
19         public override int HitPoint => 100;
20
21         /// <summary>
22         /// Random from green, blue and yellow
23         /// </summary>
24         public override Image Image {
25             get
26             {
27                 Image im;
28
29                 if (this._image == null)
30                 {
31                     Random rnd = new Random();
32                     int r = rnd.Next(0, 3);

```

```

33         if (r == 0)
34             im = Resource.yellow;
35         else if (r == 1)
36             im = Resource.blue;
37         else
38             im = Resource.green;
39
40         this._image = im;
41     }
42     else
43         im = this._image;
44
45     return im;
46 }
47 }
48 }
49 }

```

```

1  using System.Drawing;
2
3  namespace MasterDrums.Model
4  {
5      /// <summary>
6      /// Class that represents a special note
7      /// </summary>
8      class SpecialNote : INote
9      {
10         public SpecialNote(notePosition pos)
11         {
12             this._position = pos;
13         }
14
15         public override int HitPoint => 200;
16
17         public override Image Image => Resource.special;
18     }
19 }

```

```

1  using System.Drawing;
2
3  namespace MasterDrums.Model
4  {
5      /// <summary>

```

```

6  /// Class that represents a pause note
7  /// </summary>
8  class PauseNote : INote
9  {
10     public PauseNote(notePosition pos)
11     {
12         this._position = pos;
13     }
14
15     public override Image Image => null;
16
17     /// <summary>
18     /// The pause note cannot be hit and therefore has 0 point if
19         ↳ theoretically hit
20     /// </summary>
21     public override int HitPoint => 0;
22 }

```

```

1  using System.Threading;
2  using MasterDrums.Utils;
3
4  namespace MasterDrums.Model
5  {
6      /// <summary>
7      /// Abstract class for the note generator objects.
8      /// The class implements the Observer design pattern in order to
9          ↳ communicate the generated notes
10     /// </summary>
11     public abstract class INoteGenerator : ISubject
12     {
13         public const int MIN_BPM = 30;
14         public const int MAX_BPM = 300;
15         private int _bpm;
16
17         private Thread _generatorThread;
18         private bool _isRunning;
19         private bool _isPaused;
20         private Semaphore _resume;
21
22         private INote _noteGenerated;
23
24         /// <summary>
25         /// Constructor that sets the initial bpm value and create the
26             ↳ internal thread
27         /// used to infinitely generate notes.

```

```

26     /// </summary>
27     /// <param name="bpm">The bpm value</param>
28     public INoteGenerator(int bpm)
29     {
30         this._bpm = bpm;
31         this._resume = new Semaphore(0, 1);
32         this._isPaused = false;
33         this._isRunning = false;
34     }
35
36     /// <summary>
37     /// Constructor that sets the initial bpm to 50 and create the
38     /// ↪ internal thread used to infinitely generate notes.
39     /// </summary>
40     public INoteGenerator(): this(50) { }
41
42     /// <summary>
43     /// Bpm property.
44     /// If the value is less than 30 or more than 300 the bpm are set
45     /// ↪ to 50bpm.
46     /// </summary>
47     public int Bpm
48     {
49         get => this._bpm;
50         set => this._bpm = (value < MIN_BPM || value > MAX_BPM) ?
51             ↪ MIN_BPM : value;
52     }
53
54     /// <summary>
55     /// Method used to get the current generated note.
56     /// </summary>
57     /// <returns>The generated note</returns>
58     public INote CurrentNote
59     {
60         get => this._noteGenerated;
61     }
62
63     /// <summary>
64     /// The private internal routine used to generate notes based on
65     /// ↪ bpm.
66     /// 2 notes are generated in each beat.
67     /// </summary>
68     private void InternalThreadRoutine()
69     {
70         bool isResumed = true;
71
72         while (this._isRunning)
73         {

```

```

70         if (this._isPaused)
71             isResumed = this._resume.WaitOne(500);
72
73         if (isResumed)
74         {
75             this._noteGenerated = this.NextNote();
76             this.Notify();
77
78             // to calculate the time needed to sleep we simply
79                 need to divide
80             // 60 * 1000 (the milliseconds in 1 minute) by the
81                 bpm
82             int timeToSleep = ((60 * 1000) / this._bpm) / 2;
83             Thread.Sleep(timeToSleep);
84         }
85     }
86
87     /// <summary>
88     /// Method used to get the next note.
89     /// </summary>
90     public abstract INote NextNote();
91
92     /// <summary>
93     /// Starts the internal thread
94     /// </summary>
95     public void Start()
96     {
97         this._isPaused = false;
98         this._isRunning = true;
99         this.InitThread();
100        this._generatorThread.Start();
101    }
102
103    /// <summary>
104    /// Stops the internal thread
105    /// </summary>
106    public void Stop()
107    {
108        this._isRunning = false;
109        this._generatorThread.Abort();
110    }
111
112    /// <summary>
113    /// Pause the internal thread
114    /// </summary>
115    public void Pause()
116    {

```

```

116         this._isPaused = true;
117     }
118
119     /// <summary>
120     /// Resume the game
121     /// </summary>
122     public void Resume()
123     {
124         this._resume.Release();
125         this._isPaused = false;
126     }
127
128     /// <summary>
129     /// Prepare the thread which will generate notes
130     /// </summary>
131     public void InitThread()
132     {
133         this._generatorThread = new Thread(new
134             ↳ ThreadStart(this.InternalThreadRoutine));
135     }
136 }

```

```

1  using System;
2
3  namespace MasterDrums.Model
4  {
5      /// <summary>
6      /// Class that produces alternated notes (starting from right hand)
7      /// </summary>
8      class AlternatedHandNoteGenerator : INoteGenerator
9      {
10         /// <summary>
11         /// Initial position is right hand
12         /// </summary>
13         private INote.notePosition _lastHand = INote.notePosition.Right;
14
15         public AlternatedHandNoteGenerator(int bpm) : base(bpm) { }
16
17         public AlternatedHandNoteGenerator() : base() { }
18
19         public override string ToString() => "Mani alternate";
20
21         /// <summary>
22         /// Generate the next note. The probability of it being special
23         ↳ is 20%.

```

```

23     /// </summary>
24     /// <returns>The note instance</returns>
25     public override INote NextNote()
26     {
27         Random rnd = new Random();
28         INote outNote;
29
30         this._lastHand = (this._lastHand == INote.notePosition.Right)
31             ?
32             INote.notePosition.Left :
33             INote.notePosition.Right;
34
35         // if note is special is determined with a random number from
36         // 1 to 5
37         int noteType = rnd.Next(1, 5);
38
39         // if the random number is in the range [1, 4] a standard
40         // note is generated
41         if (noteType <= 4)
42             outNote = new StandardNote(this._lastHand);
43         else
44             outNote = new SpecialNote(this._lastHand);
45
46         return outNote;
47     }
48 }

```

```

1  using System;
2
3  namespace MasterDrums.Model
4  {
5      /// <summary>
6      /// Class that produces the notes to be played randomly
7      /// </summary>
8      class RandomNoteGenerator : INoteGenerator
9      {
10         public RandomNoteGenerator(int bpm) : base(bpm) { }
11
12         public RandomNoteGenerator() : base() { }
13
14         public override string ToString() => "Combinazioni casuali";
15
16         /// <summary>
17         /// Generate a random note.
18         /// Same probability of being left or right (50% left 50% right)

```

```

19      /// 60% of probability of being a standard note, 30% of being a
    ↪ pause
20      /// 10% of being special.
21      /// </summary>
22      /// <returns>The note instance</returns>
23      public override INote NextNote()
24      {
25          Random rnd = new Random();
26          INote.notePosition pos;
27          INote outNote;
28
29          // right or left position is determined with a random
    ↪ boolean
30          if (rnd.Next(0, 2) == 0)
31              pos = INote.notePosition.Left;
32          else
33              pos = INote.notePosition.Right;
34
35          // note type is determined with a random number from 1 to 10
36          int noteType = rnd.Next(1, 11);
37
38          // if the random number is in the range [1, 6] a standard
    ↪ note is generated
39          // if in the range [7, 9] a pause note is generated
40          // if in the range (9, 10] a special note is generated
41          if (noteType <= 6)
42              outNote = new StandardNote(pos);
43          else if (noteType <= 9)
44              outNote = new PauseNote(pos);
45          else {
46              outNote = new SpecialNote(pos);
47          }
48
49          return outNote;
50      }
51  }
52  }

```

```

1  namespace MasterDrums.Model
2  {
3      /// <summary>
4      /// Interface for a class tha will take care of implementing the game
    ↪ mechanics
5      /// such as score keeping and bpm increasing in time.
6      /// </summary>
7      public interface IGame

```



```

8      {
9          /// <summary>
10         /// The player name
11         /// </summary>
12         string PlayerName { get; set; }
13
14         /// <summary>
15         /// The bpm at witch the user is currently playing
16         /// </summary>
17         int Bpm { get; }
18
19         /// <summary>
20         /// The score totalized by the user
21         /// </summary>
22         int Score { get; }
23
24         /// <summary>
25         /// The user has hit a note, the score is augmented based on bpm
26         /// ↪ and note type.
27         /// </summary>
28         /// <param name="note">The note that has been hitted</param>
29         /// <param name="deltaT">The delay in ms from the perfect hit
30         /// ↪ spot</param>
31         void Hit(INote note, double deltaT);
32
33         /// <summary>
34         /// Called when an empty hit is performed
35         /// </summary>
36         void Hit();
37
38         /// <summary>
39         /// Saves the score of the user in the .csv file
40         /// </summary>
41         void SerializeScore();
42
43         /// <summary>
44         /// Ms after which a note is considered as wasted
45         /// </summary>
46         int NoteWastedMs
47         {
48             get;
49         }
50
51         /// <summary>
52         /// Wrong hits remaining until the game ends
53         /// </summary>
54         int WrongHitsRemaining
55         {

```

```

54         get;
55     }
56 }
57 }

```

```

1  using MasterDrums.Exception;
2  using System;
3  using System.Collections.Generic;
4  using System.IO;
5
6  namespace MasterDrums.Model
7  {
8      /// <summary>
9      /// The game class contains the game state informations.
10     /// </summary>
11     public class Game : IGame
12     {
13         private string _playerName = null;
14         private int _bpm = -1;
15
16         private int _score = 0;
17         private int _wastedNotes = 0;
18         private int _hittedNotes = 0;
19
20         private List<Tuple<int, String>> _results = new List<Tuple<int,
21             String>>();
22
23         /// <summary>
24         /// Creates the game instance, sets the initial bpm and loads the
25         /// records internally
26         /// </summary>
27         /// <param name="initialBpm">The initial bpm</param>
28         public Game(int initialBpm) : base()
29         {
30             this._results = LoadBestResults();
31             this._bpm = initialBpm;
32         }
33
34         /// <summary>
35         /// Sets the player's name.
36         /// null is used if the player name is an empty string
37         /// </summary>
38         public string PlayerName {
39             get => this._playerName;
40             set => this._playerName = string.IsNullOrEmpty(value) ?
41                 null : value;
42         }
43     }
44 }

```

```

39     }
40
41     /// <summary>
42     /// The BPM at which the user is playing
43     /// </summary>
44     public int Bpm => this._bpm;
45
46     /// <summary>
47     /// The user score
48     /// </summary>
49     public int Score => this._score;
50
51     /// <summary>
52     /// A note is considered as wasted if it's performed 50ms before
53         ↳ or after it would naturally occur
54     /// </summary>
55     public int NoteWastedMs
56     {
57         get => 50;
58     }
59
60     /// <summary>
61     /// Method called when a note has been hit.
62     /// Every 5 notes hitted the bpm is increased by 1,
63     /// Every 2ms of delay from the perfect hit represents a 1 point
64         ↳ penalty
65     /// </summary>
66     /// <param name="note">The note hitted</param>
67     /// <param name="deltaT">The distance in time from the perfect
68         ↳ hit time</param>
69     public void Hit(INote note, double deltaT)
70     {
71         // every 2ms a 1 point penalty is added
72         int penalty = (int)Math.Round(deltaT / 2.0);
73         this._score += (note.HitPoint - penalty);
74
75         this._hittedNotes++;
76         if ((this._hittedNotes % 5) == 0)
77             this._bpm++;
78     }
79
80     /// <summary>
81     /// Called when an empty hit has been performed
82     /// </summary>
83     public void Hit()
84     {
85         this._wastedNotes++;
86     }

```

```

84         if (this._wastedNotes >= 20)
85             throw new GameEndedException();
86     }
87
88     /// <summary>
89     /// Serialize the user score to a file
90     /// </summary>
91     public void SerializeScore()
92     {
93         this._results = Game.LoadBestResults();
94
95         Tuple<int, String> t = new Tuple<int, String>(this._score,
96             ↪ this._playerName);
97         if (this._score != 0)
98         {
99             this.AddAndOrderResult(t);
100             this.AddResultToFile();
101         }
102     }
103
104     /// <summary>
105     /// Wrong hits remaining until the game ends
106     /// </summary>
107     /// <returns>The number of wrong hits that can be
108         ↪ executed</returns>
109     public int WrongHitsRemaining
110     {
111         get => (20 - this._wastedNotes);
112     }
113
114     /// <summary>
115     /// Loads the highscores from the file containg them
116     /// </summary>
117     /// <returns>A list of record in the format score -
118         ↪ name</returns>
119     public static List<Tuple<int, String>> LoadBestResults()
120     {
121         try
122         {
123             List<Tuple<int, String>> results = new List<Tuple<int,
124                 ↪ String>>();
125             StreamReader sr = new StreamReader("../record.csv");
126             while (!sr.EndOfStream)
127             {
128                 string line = sr.ReadLine();
129                 string[] values = line.Split(';');
130                 if (!string.IsNullOrEmpty(line))

```

```

128         {
129             results.Add(new Tuple<int,
130                 ↪ String>(int.Parse(values[1]), values[0]));
131         }
132     }
133     results.Sort();
134     results.Reverse();
135     sr.Close();
136     return results;
137 }
138 catch(FileNotFoundException)
139 {
140     return null;
141 }
142 }
143
144 /// <summary>
145 /// Add a new score to the internal list
146 /// </summary>
147 /// <param name="t">The new result</param>
148 private void AddAndOrderResult(Tuple<int, String> t)
149 {
150     if (this._results != null)
151     {
152         this._results.Add(t);
153         this._results.Sort();
154         this._results.Reverse();
155     }
156     else
157     {
158         this._results = new List<Tuple<int, String>>();
159         this._results.Add(t);
160     }
161 }
162
163 /// <summary>
164 /// Write results to csv file
165 /// </summary>
166 private void AddResultToFile()
167 {
168     StreamWriter sw = new StreamWriter("../..record.csv",
169         ↪ false);
170     foreach (Tuple<int, String> t in this._results)
171     {
172         sw.WriteLine(t.Item2 + ";" + t.Item1);
173     }
    sw.Close();

```

```
174     }
175 }
176 }
```

4.2 Controller

```
1  using MasterDrums.Model;
2  using MasterDrums.View;
3
4  namespace MasterDrums.Controller
5  {
6      public interface IController
7      {
8          /// <summary>
9          /// Starts the game
10         /// </summary>
11         void StartGame();
12
13         /// <summary>
14         /// Stops the game
15         /// </summary>
16         void StopGame();
17
18         /// <summary>
19         /// The main view reference
20         /// </summary>
21         IMainView MainView
22         {
23             get;
24             set;
25         }
26
27         /// <summary>
28         /// The name of the player who's playing
29         /// </summary>
30         string PlayerName
31         {
32             set;
33             get;
34         }
35
36         /// <summary>
37         /// The initial bpm of the game
38         /// </summary>
```

```

39     int InitialBpm
40     {
41         set;
42         get;
43     }
44
45     /// <summary>
46     /// The current bpm
47     /// </summary>
48     int Bpm
49     {
50         get;
51     }
52
53     /// <summary>
54     /// The players score
55     /// </summary>
56     int Score
57     {
58         get;
59     }
60
61     /// <summary>
62     /// A note has been hitted by the user
63     /// </summary>
64     /// <param name="note">The note hitted by the user</param>
65     /// <param name="delay">The time in ms from the perfect hit
66     ///     time</param>
67     void NoteHitted(INote note, int delay);
68
69     /// <summary>
70     /// Called when an empty hit is performed
71     /// </summary>
72     void EmptyHit();
73
74     /// <summary>
75     /// Distance in ms from perfect note hit time in order to
76     ///     consider a note as hittable
77     /// </summary>
78     int HittedNoteInterval
79     {
80         get;
81     }
82
83     /// <summary>
84     /// The remaining wrong hits until the game end
85     /// </summary>
86     int WrongHitsRemaining {

```

```

85         get;
86     }
87 }
88 }

```

```

1  using MasterDrums.Model;
2  using MasterDrums.View;
3
4  namespace MasterDrums.Controller
5  {
6      /// <summary>
7      /// Main controller, used to reflect the view events in the model.
8      /// </summary>
9      public class MainController : IController
10     {
11         private Game _game;
12
13         private int _initialBpm = -1;
14         private string _playerName = null;
15         private IMainView _mainView = null;
16
17         /// <summary>
18         /// If the initial BPM and the player name are set the game is
19         /// ↪ started
20         /// otherwise an exception of type <c>GameOptionException</c> it
21         /// ↪ raised
22         /// </summary>
23         public void StartGame()
24         {
25             if (this._initialBpm == -1 && this._playerName == null)
26                 throw new GameOptionException("Game options not set");
27
28             this._game = new Game(this._initialBpm);
29             this._game.PlayerName = this._playerName;
30         }
31
32         /// <summary>
33         /// Stops the game and serializes the player's score.
34         /// </summary>
35         public void StopGame() {
36             this._mainView.StopGame();
37             this._game.SerializeScore();
38             this._mainView.RefreshHighscores();
39         }
40
41         /// <summary>

```



```

40     /// Communicate to the game model that a note has been hitted
41     /// </summary>
42     /// <param name="note">The note hitted</param>
43     /// <param name="delay">Delay from the perfect time</param>
44     public void NoteHitted(INote note, int delay) =>
45         => this._game.Hit(note, delay);
46
47     /// <summary>
48     /// Communicate to the model that an empty hit has been
49     /// </summary>
50     /// performed
51     public void EmptyHit() => this._game.Hit();
52
53     /// <summary>
54     /// Sets and gets the player's name
55     /// </summary>
56     public string PlayerName
57     {
58         set => this._playerName = string.IsNullOrEmpty(value) ?
59             null : value;
60         get => this._playerName;
61     }
62
63     /// <summary>
64     /// Sets and gets the initial bpm.
65     /// </summary>
66     public int InitialBpm
67     {
68         set => this._initialBpm = value;
69         get => this._initialBpm;
70     }
71
72     /// <summary>
73     /// Gets the bpm at which the user is playing
74     /// </summary>
75     public int Bpm {
76         get => this._game.Bpm;
77     }
78
79     /// <summary>
80     /// Main view instance
81     /// </summary>
82     public IMainView MainView
83     {
84         get => this._mainView;
85         set => this._mainView = value;
86     }

```

```

85     /// <summary>
86     /// The score totalized by the user
87     /// </summary>
88     public int Score
89     {
90         get => this._game.Score;
91     }
92
93     /// <summary>
94     /// Time after or before an hit is considered as wasted
95     /// </summary>
96     public int HittedNoteInterval
97     {
98         get => this._game.NoteWastedMs;
99     }
100
101     /// <summary>
102     /// The remaining wrong hits until the game ends
103     /// </summary>
104     public int WrongHitsRemaining
105     {
106         get => this._game.WrongHitsRemaining;
107     }
108
109     }
110 }

```

4.3 View

```

1  using MasterDrums.Model;
2  using System;
3  using System.Collections.Generic;
4
5  namespace MasterDrums.View
6  {
7      /// <summary>
8      /// Interface implemented by the main view
9      /// </summary>
10     public interface IMainView
11     {
12         /// <summary>
13         /// Shows the main menu view
14         /// </summary>
15         void MainMenu();

```

```

16
17     /// <summary>
18     /// Shows the new game view
19     /// </summary>
20     void NewGame();
21
22     /// <summary>
23     /// Close the application
24     /// </summary>
25     void Quit();
26
27     /// <summary>
28     /// Shows the highscores view
29     /// </summary>
30     void Highscores();
31
32     /// <summary>
33     /// Starts a new game
34     /// </summary>
35     void StartGame(string playerName, int initialBpm, INoteGenerator
        ↪ gameMode);
36
37     /// <summary>
38     /// Pause the game
39     /// </summary>
40     void PauseGame();
41
42     /// <summary>
43     /// Resume the game
44     /// </summary>
45     void ResumeGame();
46
47     /// <summary>
48     /// Stop the current game.
49     /// </summary>
50     void StopGame();
51
52     /// <summary>
53     /// Left note trigger
54     /// </summary>
55     void LeftNoteHit();
56
57     /// <summary>
58     /// Right note trigger
59     /// </summary>
60     void RightNoteHit();
61
62     /// <summary>

```

```

63     /// Refresh the highscores panel after new record has been added
64     /// </summary>
65     void RefreshHighscores();
66 }
67 }

```

```

1  using MasterDrums.Model;
2  using MasterDrums.Controller;
3  using System;
4  using System.Windows.Forms;
5  using System.Drawing;
6
7  namespace MasterDrums.View
8  {
9      public class MainView : Form, IMainView
10     {
11         private IController _controller;
12
13         private MainMenuPanel _mainMenuPanel;
14         private NewGamePanel _newGamePanel;
15         private PlayingPanel _playingPanel;
16         private GamePausePanel _gamePausePanel;
17         private HighscoresPanel _highscoresPanel;
18
19         /// <summary>
20         /// Constructor that sets the controller to interact with the
21             ↪ application model
22         /// </summary>
23         /// <param name="controller">The instance of the
24             ↪ controller</param>
25         public MainView(IController controller)
26         {
27             Application.EnableVisualStyles();
28
29             this._controller = controller;
30             this._controller.MainView = this;
31
32             // Create panels
33             this._mainMenuPanel = new MainMenuPanel(this);
34             this._newGamePanel = new NewGamePanel(this);
35             this._playingPanel = new PlayingPanel(this,
36                 ↪ this._controller);
37             this._gamePausePanel = new GamePausePanel(this);
38             this._highscoresPanel = new HighscoresPanel(this);
39
40             this.KeyPreview = true;

```

```

38     this.KeyUp += (s, e) =>
39     {
40         switch (e.KeyCode)
41         {
42             case Keys.C:
43                 if (this._playingPanel.IsRunning)
44                     this.LeftNoteHit();
45                 break;
46             case Keys.N:
47                 if (this._playingPanel.IsRunning)
48                     this.RightNoteHit();
49                 break;
50
51             case Keys.Escape:
52                 if (this._playingPanel.IsRunning)
53                     this.PauseGame();
54                 break;
55
56         }
57     };
58
59     this.InitializeComponent();
60     this.ShowMainMenuView();
61
62     Application.Run(this);
63 }
64
65 #region Panels setup methods
66 /// <summary>
67 /// Method used to initialize form components
68 /// </summary>
69 private void InitializeComponent()
70 {
71     // Main form setup
72     this.SuspendLayout();
73
74     // fullscreen
75     int screenWidth = Screen.PrimaryScreen.Bounds.Width;
76     int screenHeight = Screen.PrimaryScreen.Bounds.Height;
77     this.ClientSize = new Size(screenWidth, screenHeight);
78     this.ControlBox = false;
79     this.FormBorderStyle = FormBorderStyle.FixedSingle;
80     this.MaximizeBox = false;
81     this.StartPosition = FormStartPosition.CenterScreen;
82     this.ResumeLayout(false);
83
84     // Formatting each panel with right dimension
85     this.MainMenuPanelSetup();

```

```

86         this.NewGamePanelSetup();
87         this.PlayingPanelSetup();
88         this.GamePausePanelSetup();
89         this.HighscoresPanelSetup();
90     }
91
92     /// <summary>
93     /// Sets up the main menu panel in the form and hides it
94     /// </summary>
95     private void MainMenuPanelSetup()
96     {
97         int mainMenuPanelWidth = this.ClientSize.Width / 2;
98         int mainMenuPanelHeight = this.ClientSize.Height / 2;
99         int mainMenuPanelX = (this.ClientSize.Width / 2) -
100             ↳ (mainMenuPanelWidth / 2);
101         int mainMenuPanelY = (this.ClientSize.Height / 2) -
102             ↳ (mainMenuPanelHeight / 2);
103         this._mainMenuPanel.Size = new Size(mainMenuPanelWidth,
104             ↳ mainMenuPanelHeight);
105         this._mainMenuPanel.Location = new Point(mainMenuPanelX,
106             ↳ mainMenuPanelY);
107         this.Controls.Add(this._mainMenuPanel);
108         this._mainMenuPanel.Hide();
109     }
110
111     /// <summary>
112     /// Sets up the highscores panel in the form and hides it
113     /// </summary>
114     private void HighscoresPanelSetup()
115     {
116         int highscoresPanelWidth = this.ClientSize.Width / 2;
117         int highscoresPanelHeight = this.ClientSize.Height / 2;
118         int highscoresPanelX = (this.ClientSize.Width / 2) -
119             ↳ (highscoresPanelWidth / 2);
120         int highscoresPanelY = (this.ClientSize.Height / 2) -
121             ↳ (highscoresPanelHeight / 2);
122         this._highscoresPanel.Size = new Size(highscoresPanelWidth,
123             ↳ highscoresPanelHeight);
124         this._highscoresPanel.Location = new Point(highscoresPanelX,
125             ↳ highscoresPanelY);
126         this.Controls.Add(this._highscoresPanel);
127         this._highscoresPanel.Hide();
128     }
129
130     /// <summary>
131     /// Sets up the player name panel in the form and hides it
132     /// </summary>
133     private void NewGamePanelSetup()

```

```

126     {
127         int panelWidth = this.ClientSize.Width / 2;
128         int panelHeight = this.ClientSize.Height / 2;
129         int panelX = (this.ClientSize.Width / 2) - (panelWidth / 2);
130         int panelY = (this.ClientSize.Height / 2) - (panelHeight /
131             ↪ 2);
132         this._newGamePanel.Size = new Size(panelWidth, panelHeight);
133         this._newGamePanel.Location = new Point(panelX, panelY);
134         this.Controls.Add(this._newGamePanel);
135         this._newGamePanel.Hide();
136     }
137
138     /// <summary>
139     /// Sets up the game pause panel in the form and hides it
140     /// </summary>
141     private void GamePausePanelSetup()
142     {
143         int panelWidth = this.ClientSize.Width / 2;
144         int panelHeight = this.ClientSize.Height / 2;
145         int panelX = (this.ClientSize.Width / 2) - (panelWidth / 2);
146         int panelY = (this.ClientSize.Height / 2) - (panelHeight /
147             ↪ 2);
148         this._gamePausePanel.Size = new Size(panelWidth,
149             ↪ panelHeight);
150         this._gamePausePanel.Location = new Point(panelX, panelY);
151         this.Controls.Add(this._gamePausePanel);
152         this._gamePausePanel.Hide();
153     }
154
155     /// <summary>
156     /// Sets up the playing panel in the form and hides it
157     /// </summary>
158     private void PlayingPanelSetup()
159     {
160         int panelWidth = this.ClientSize.Width;
161         int panelHeight = this.ClientSize.Height;
162         int panelX = 0;
163         int panelY = 0;
164         this._playingPanel.Size = new Size(panelWidth, panelHeight);
165         this._playingPanel.Location = new Point(panelX, panelY);
166         this.Controls.Add(this._playingPanel);
167         this._playingPanel.Hide();
168     }
169     #endregion
170
171     #region Panels display methods
172     /// <summary>
173     /// Remove all controls from the main panel

```

```

171     /// </summary>
172     private void ClearView()
173     {
174         this._mainMenuPanel.Hide();
175         this._highscoresPanel.Hide();
176         this._newGamePanel.Hide();
177         this._playingPanel.Hide();
178         this._gamePausePanel.Hide();
179     }
180
181     /// <summary>
182     /// Shows the main menu
183     /// </summary>
184     public void ShowMainMenuView() => this._mainMenuPanel.Show();
185
186     /// <summary>
187     /// Shows the highscores view
188     /// </summary>
189     public void ShowHighscoresView() => this._highscoresPanel.Show();
190
191     /// <summary>
192     /// Shows the new game option menu
193     /// </summary>
194     public void ShowNewGameView() => this._newGamePanel.Show();
195
196     /// <summary>
197     /// Shows the game pause menu
198     /// </summary>
199     public void ShowGamePauseView() {
200         this._gamePausePanel.Show();
201         this._gamePausePanel.BringToFront();
202     }
203
204     /// <summary>
205     /// Hides the game pause menu
206     /// </summary>
207     public void HideGamePauseView() => this._gamePausePanel.Hide();
208
209     /// <summary>
210     /// Hides the playing panel
211     /// </summary>
212     public void HidePlayingPanelView() => this._playingPanel.Hide();
213
214     /// <summary>
215     /// Shows the playing view
216     /// </summary>
217     public void ShowPlayingView()
218     {

```



```

219         this._playingPanel.Draw();
220         this._playingPanel.Show();
221     }
222     #endregion
223
224     /// <summary>
225     /// Shows the main menu panel
226     /// </summary>
227     public void MainMenu()
228     {
229         this.ClearView();
230         this.ShowMainMenuView();
231     }
232
233     /// <summary>
234     /// Called when the user clicks on the new game button in the
235         ↪ main menu panel.
236     /// Shows the new game panel which will ask the user to insert
237         ↪ his name, initial bpm and game mode
238     /// </summary>
239     public void NewGame()
240     {
241         this.ClearView();
242         this._newGamePanel.ClearTxtUsername();
243         this.ShowNewGameView();
244     }
245
246     /// <summary>
247     /// Shows the highscores view
248     /// </summary>
249     public void Highscores()
250     {
251         if(Game.LoadBestResults() != null)
252         {
253             this.ClearView();
254             this.ShowHighscoresView();
255         }
256         else
257         {
258             MessageBox.Show("Nessun risultato presente!");
259         }
260     }
261
262     /// <summary>
263     /// Close the application
264     /// </summary>
265     public void Quit()
266     {

```

```

265         Environment.Exit(0);
266     }
267
268     /// <summary>
269     /// Hide all the panels and shows the gaming panel.
270     /// </summary>
271     /// <param name="playerName">The player name</param>
272     /// <param name="initialBpm">The initial bpm</param>
273     /// <param name="gameMode">The game mode selected</param>
274     public void StartGame(string playerName, int initialBpm,
275         ↪ INoteGenerator gameMode)
276     {
277         this.ClearView();
278         this.ShowPlayingView();
279
280         this._controller.PlayerName = playerName;
281         this._controller.InitialBpm = initialBpm;
282
283         this._playingPanel.GameMode = gameMode;
284         this._playingPanel.StartGame();
285     }
286
287     /// <summary>
288     /// Show the pause panel (on top of the playing panel).
289     /// Communicate to the playing panel that the user wants to pause
290     /// ↪ the game
291     /// </summary>
292     public void PauseGame()
293     {
294         this.ShowGamePauseView();
295         this._playingPanel.PauseGame();
296     }
297
298     /// <summary>
299     /// Hide the pause panel.
300     /// Communicate to the playing panel that the user wants to
301     /// ↪ resume the game
302     /// </summary>
303     public void ResumeGame()
304     {
305         this.HideGamePauseView();
306         this._playingPanel.ResumeGame();
307     }
308
309     /// <summary>
310     /// Stop the current game and return to the initial menu.
311     /// </summary>
312     public void StopGame()

```

```

310     {
311         this.HidePlayingPanelView();
312         this.HideGamePauseView();
313         this.ShowMainMenuView();
314     }
315
316     /// <summary>
317     /// Left note has been hit
318     /// </summary>
319     public void LeftNoteHit()
320     {
321         this._playingPanel.LeftNoteHit();
322     }
323
324     /// <summary>
325     /// Right note has been hit
326     /// </summary>
327     public void RightNoteHit()
328     {
329         this._playingPanel.RightNoteHit();
330     }
331
332     /// <summary>
333     /// Refresh the highscores panel after new records are added
334     /// </summary>
335     public void RefreshHighscores()
336     {
337         this._highscoresPanel = new HighscoresPanel(this);
338         this.HighscoresPanelSetup();
339     }
340
341     }
342 }

```

```

1  using MasterDrums.Model;
2
3  namespace MasterDrums.View
4  {
5      /// <summary>
6      /// Interface implemented by the view that takes care of showing the
7      /// ↪ game elements
8      /// </summary>
9      public interface IPlayingView
10     {
11         /// <summary>
12         /// The time that a note takes in order to go from the top of the
13         /// ↪ screen to the perfect hit spot.

```

```

12     /// </summary>
13     float NoteRideTime
14     {
15         get;
16     }
17
18     /// <summary>
19     /// Method called when a left note is hitted.
20     /// </summary>
21     void LeftNoteHit();
22
23     /// <summary>
24     /// Method called when a right note is hitted.
25     /// </summary>
26     void RightNoteHit();
27
28     /// <summary>
29     /// The mode played by the user
30     /// </summary>
31     INoteGenerator GameMode
32     {
33         get;
34         set;
35     }
36
37     /// <summary>
38     /// Starts the game
39     /// </summary>
40     void StartGame();
41
42     /// <summary>
43     /// Stops the game
44     /// </summary>
45     void StopGame();
46
47     /// <summary>
48     /// Puts the note generator in pause
49     /// </summary>
50     void PauseGame();
51
52     /// <summary>
53     /// Resumes the game from pause
54     /// </summary>
55     void ResumeGame();
56
57     /// <summary>
58     /// States if the game is running or not
59     /// </summary>

```

```

60         bool IsRunning
61         {
62             get;
63         }
64     }
65 }

```

```

1  using System;
2  using System.Drawing;
3  using System.Windows.Forms;
4  using System.Collections.Generic;
5  using MasterDrums.Model;
6  using MasterDrums.Utills;
7  using MasterDrums.Controller;
8  using MasterDrums.Exception;
9  using NAudio.Wave;
10
11 namespace MasterDrums.View
12 {
13     /// <summary>
14     /// View used to play the game.
15     /// </summary>
16     class PlayingPanel : Panel, IPlayingView, IObservable
17     {
18         private const int STICK_DOWN_MS = 50;
19         private const int REFRESH_RATE_MS = 20;
20
21         private IMainView _mainView;
22         private IController _controller;
23
24         private INoteGenerator _noteGenerator = null;
25         private Timer _gameLoopTimer;
26         private bool _isRunning;
27
28         private PictureBox _backgroundPictureBox;
29
30         private bool _leftStickDown = false;
31         private bool _rightStickDown = false;
32         private WaveOutEvent _outputDevice;
33         private WaveFileReader _audio;
34
35         private System.Threading.Mutex _notesMutex;
36         private LinkedList<Triplet<INote, Point, int>> _notes;
37
38         /// <summary>

```

```

39     /// Constructor method
40     /// </summary>
41     /// <param name="mainView">The main view instance that created
    ↪ the panel</param>
42     /// <param name="controller">The controller instance used to
    ↪ communicate with the model</param>
43     public PlayingPanel(IMainView mainView, IController controller)
44     {
45         this._mainView = mainView;
46         this._controller = controller;
47
48         // create the note generator and subscribe as listener
49         this._isRunning = false;
50
51         // the queue containg the notes viewed by the user
52         this._notes = new LinkedList<Triplet<INote, Point, int>>();
53
54         // the game loop timer takes care of drawing the required
    ↪ objects in the panel's main picture box
55         this._gameLoopTimer = new Timer();
56         _gameLoopTimer.Interval = REFRESH_RATE_MS;
57
58         // the output sound configuration using NAudio lib
59         this._outputDevice = new WaveOutEvent();
60         this._audio = new WaveFileReader(Resource.snare_hit);
61         this._outputDevice.Init(this._audio);
62     }
63
64     /// <summary>
65     /// Called by the notegenerator when a new note has been
    ↪ generated
66     /// Thread safe
67     /// </summary>
68     /// <param name="ng">The note generator instance</param>
69     public void Update(ISubject ng)
70     {
71         try
72         {
73             this._notesMutex.WaitOne();
74
75             INote note = this._noteGenerator.CurrentNote;
76
77             if (!(note is PauseNote))
78             {
79                 int timestamp = (int)(DateTime.UtcNow.Subtract(new
    ↪ DateTime(1970, 1, 1))).TotalSeconds;
80
81                 Point point;

```

```

82         if (note.Position == INote.notePosition.Left)
83             point = new Point(0, 0);
84         else
85             point = new Point(this.Size.Width, 0);
86
87         this._notes.AddLast(new Triplet<INote, Point,
88             ↳ int>(note, point, timestamp));
89     }
90     finally
91     {
92         this._notesMutex.ReleaseMutex();
93     }
94 }
95
96 /// <summary>
97 /// The mode played by the user
98 /// </summary>
99 public INoteGenerator GameMode
100 {
101     get => this._noteGenerator;
102     set {
103         if (this._noteGenerator != null)
104             this._noteGenerator.Detach(this);
105
106         this._noteGenerator = value;
107         this._noteGenerator.Attach(this);
108     }
109 }
110
111 /// <summary>
112 /// Draw the background main picture box and setup the game loop
113 /// ↳ to update it
114 /// </summary>
115 public void Draw()
116 {
117     this.SuspendLayout();
118     this.Controls.Clear();
119
120     this._backgroundPictureBox = new PictureBox();
121     this._backgroundPictureBox.Width = this.Size.Width;
122     this._backgroundPictureBox.Height = this.Size.Height;
123     this._backgroundPictureBox.Location = new Point(0, 0);
124
125     this._backgroundPictureBox.Paint += this.PaintObjects;
126     this._gameLoopTimer.Tick += (s, e) =>
        ↳ this._backgroundPictureBox.Invalidate();

```

```

127         this.Controls.Add(this._backgroundPictureBox);
128         this.ResumeLayout();
129     }
130
131     /// <summary>
132     /// Draw the object composing the game scene
133     /// </summary>
134     private void PaintObjects(object sender, PaintEventArgs e)
135     {
136         e.Graphics.Clear(Color.White);
137
138         this.DrawSnare(e.Graphics);
139         this.DrawNotes(e.Graphics);
140         this.DrawScore(e.Graphics);
141         this.DrawRemainingErrors(e.Graphics);
142         this.DrawReachedBpm(e.Graphics);
143
144         if (this._leftStickDown)
145             this.DrawLeftStickDown(e.Graphics);
146         else
147             this.DrawLeftStickUp(e.Graphics);
148
149         if (this._rightStickDown)
150             this.DrawRightStickDown(e.Graphics);
151         else
152             this.DrawRightStickUp(e.Graphics);
153     }
154
155     /// <summary>
156     /// Draws the snare on the center of the screen
157     /// </summary>
158     /// <param name="g">The graphics object where the image is
159         ↪ drawn</param>
160     private void DrawSnare(Graphics g)
161     {
162         int snareSide = (int)Math.Round(this.Size.Width / 3.0);
163         int snareX = (this.Width - snareSide) / 2;
164         int snareY = (int)Math.Round(this.Size.Height * 0.6);
165         Image snare = Resource.snare;
166         g.DrawImage(snare, snareX, snareY, snareSide, snareSide);
167
168         // draw right and left hit spot
169         int ellipseHeight = (int)Math.Round(this.Size.Height * 0.05);
170         int ellipseWidth = (int)Math.Round(this.Size.Width * 0.05);
171
172         int leftEllipseX = (int)Math.Round(this.LeftHitSpotX -
173             ↪ (ellipseWidth / 2.0));
174         int rightEllipseX = (int)Math.Round(this.RightHitSpotX -
175             ↪ (ellipseWidth / 2.0));

```



```

173         int ellipseY = (int)Math.Round(this.HitSpotY - (ellipseHeight
           ↪ / 2.0));
174
175         g.DrawEllipse(new Pen(Color.LightGreen, 5F), leftEllipseX,
           ↪ ellipseY, ellipseWidth, ellipseHeight);
176         g.DrawEllipse(new Pen(Color.LightGreen, 5F), rightEllipseX,
           ↪ ellipseY, ellipseWidth, ellipseHeight);
177     }
178
179     /// <summary>
180     /// The Y of the perfect hit spot
181     /// </summary>
182     private int HitSpotY
183     {
184         get => (int)Math.Round(this.Size.Height * 0.75);
185     }
186
187     /// <summary>
188     /// The X of the left perfect hit spot
189     /// </summary>
190     private int LeftHitSpotX
191     {
192         // size of the snare + 30% of the snare width from left
193         get
194         {
195             int snareSize = (int)Math.Round(this.Size.Width / 3.0);
196             return (int)Math.Round(snareSize + (snareSize * 0.3));
197         }
198     }
199
200     /// <summary>
201     /// The X of the right perfect hit spot
202     /// </summary>
203     private int RightHitSpotX
204     {
205         // size of the snare + 70% size of the snare
206         get
207         {
208             int snareSize = (int)Math.Round(this.Size.Width / 3.0);
209             return (int)Math.Round(snareSize + (snareSize * 0.7));
210         }
211     }
212
213     /// <summary>
214     /// Draws the left stick up on the screen
215     /// </summary>
216     /// <param name="g">The graphics object where the image is
           ↪ drawed</param>

```

```

217     private void DrawLeftStickUp(Graphics g)
218     {
219         int w = (int)Math.Round(this.Size.Width * 0.25);
220         int h = w;
221         int y = (int)Math.Round(this.Size.Height * 0.55);
222         int x = (int)Math.Round((this.Size.Width * 0.30) - (w / 2));
223
224         Image leftStick = ImageUtils.RotateImage(Resource.left_stick,
225             ↪ 30);
226         g.DrawImage(leftStick, x, y, w, h);
227     }
228
229     /// <summary>
230     /// Draws the left stick down on the screen
231     /// </summary>
232     /// <param name="g">The graphics object where the image is
233         ↪ drawn</param>
234     private void DrawLeftStickDown(Graphics g)
235     {
236         int w = (int)Math.Round(this.Size.Width * 0.25);
237         int h = w;
238         int y = (int)Math.Round(this.Size.Height * 0.50);
239         int x = (int)Math.Round((this.Size.Width * 0.45) - w);
240
241         Image leftStick = ImageUtils.RotateImage(Resource.left_stick,
242             ↪ 100);
243         g.DrawImage(leftStick, x, y, w, h);
244     }
245
246     /// <summary>
247     /// Draws the right stick up on the screen
248     /// </summary>
249     /// <param name="g">The graphics object where the image is
250         ↪ drawn</param>
251     private void DrawRightStickUp(Graphics g)
252     {
253         int w = (int)Math.Round(this.Size.Width * 0.25);
254         int h = w;
255         int y = (int)Math.Round(this.Size.Height * 0.55);
256         int x = (int)Math.Round((this.Size.Width * 0.60));
257
258         Image leftStick =
259             ↪ ImageUtils.RotateImage(Resource.right_stick, -30);
260         g.DrawImage(leftStick, x, y, w, h);
261     }
262
263     /// <summary>
264     /// Draws the right stick down on the screen

```

```

260     /// </summary>
261     /// <param name="g">The graphics object where the image is
    ↪ drawn</param>
262     private void DrawRightStickDown(Graphics g)
263     {
264         int w = (int)Math.Round(this.Size.Width * 0.25);
265         int h = w;
266         int y = (int)Math.Round(this.Size.Height * 0.50);
267         int x = (int)Math.Round((this.Size.Width * 0.55));
268
269         Image leftStick =
    ↪ ImageUtils.RotateImage(Resource.right_stick, -100);
270         g.DrawImage(leftStick, x, y, w, h);
271     }
272
273     /// <summary>
274     /// Draw the notes on the screen
275     /// Thread safe
276     /// </summary>
277     /// <param name="g">The graphic object where the drawing is
    ↪ performed</param>
278     private void DrawNotes(Graphics g)
279     {
280         try
281         {
282             this._notesMutex.WaitOne();
283             Triplet<INote, Point, int>[] notesCopy = new
    ↪ Triplet<INote, Point, int>[this._notes.Count];
284             this._notes.CopyTo(notesCopy, 0);
285
286             foreach (Triplet<INote, Point, int> x in notesCopy)
287             {
288                 INote note = x.Item1;
289                 Point curPoint = x.Item2;
290
291                 // draw note
292                 int noteSide = (int)Math.Round(this.Width * 0.03);
293                 if (note is SpecialNote)
294                     noteSide *= 2;
295
296                 g.DrawImage(note.Image,
297                             (curPoint.X - (noteSide / 2)),
298                             (curPoint.Y - (noteSide / 2)),
299                             noteSide,
300                             noteSide);
301
302                 /*                /\-----> angle
303                 *                / \

```

```

304         * HitSpotY ->      / \      <- diagonalSpace
305         *                  /___\
306         *                  ^-- HitSpotX
307         */
308     double diagonalSpace =
309         ↪ Math.Sqrt(Math.Pow(this.LeftHitSpotX, 2) +
310         ↪ Math.Pow(this.HitSpotY, 2));
311     double angle = Math.Acos(this.HitSpotY /
312         ↪ diagonalSpace);
313     double speed = diagonalSpace / this.NoteRideTime;
314
315     double sy = REFRESH_RATE_MS * speed *
316         ↪ Math.Cos(angle);
317     double sx = REFRESH_RATE_MS * speed *
318         ↪ Math.Sin(angle);
319
320     int newX;
321     int newY = (int)Math.Round(curPoint.Y + sy);
322     if (note.Position == INote.notePosition.Left)
323         newX = (int)Math.Round(curPoint.X + sx);
324     else
325         newX = (int)Math.Round(curPoint.X - sx);
326
327     Point newPoint = new Point(newX, newY);
328     x.Item2 = newPoint;
329     if (newPoint.Y > this.Size.Height)
330         this._notes.Remove(x);
331     else
332         x.Item2 = newPoint;
333     }
334 } finally
335 {
336     this._notesMutex.ReleaseMutex();
337 }
338
339 /// <summary>
340 /// Draw the score on the screen
341 /// </summary>
342 /// <param name="g">The graphic object where the drawing is
343 /// ↪ performed</param>
344 private void DrawScore(Graphics g)
345 {
346     string labelScore = "Punteggio";
347     Font labelScoreFont = new Font("Arial", 35);
348     SizeF labelScoreSize = new SizeF();
349     labelScoreSize = g.MeasureString(labelScore, labelScoreFont);

```

```

346
347     SizeF scoreSize = new SizeF();
348     Font scoreFont = new Font("Arial", 25);
349     scoreSize =
        ↳ g.MeasureString(this._controller.Score.ToString(),
        ↳ scoreFont);
350
351     int labelScoreX = (int)Math.Round((this.Size.Width -
        ↳ Size.Round(labelScoreSize).Width) / 2.0);
352     int scoreX = (int)Math.Round((this.Size.Width -
        ↳ Size.Round(scoreSize).Width) / 2.0);
353     int labelScoreY = (int)Math.Round(this.Size.Height * 0.1);
354     int scoreY = (int)Math.Round(this.Size.Height * 0.2);
355     g.DrawString(labelScore, labelScoreFont, new
        ↳ SolidBrush(Color.Black), new Point(labelScoreX,
        ↳ labelScoreY));
356     g.DrawString(this._controller.Score.ToString(), scoreFont,
        ↳ new SolidBrush(Color.Black), new Point(scoreX, scoreY));
357 }
358
359 /// <summary>
360 /// Draw the number of hit that can be missed before the game
361 /// </summary>
362 /// <param name="g">The graphic object where the drawing is
363 /// </param>
364 private void DrawRemainingErrors(Graphics g)
365 {
366     string labelErrors = "Colpi errati rimasti";
367     Font labelErrorsFont = new Font("Arial", 28);
368     SizeF labelErrorsSize = new SizeF();
369     labelErrorsSize = g.MeasureString(labelErrors,
        ↳ labelErrorsFont);
370
371     int labelErrorsX = (int)Math.Round((this.Size.Width -
        ↳ Size.Round(labelErrorsSize).Width) / 2.0);
372     int labelErrorsY = (int)Math.Round(this.Size.Height * 0.3);
373     int errorsY = (int)Math.Round(this.Size.Height * 0.35);
374
375     g.DrawString(labelErrors, labelErrorsFont, new
        ↳ SolidBrush(Color.Black), new Point(labelErrorsX,
        ↳ labelErrorsY));
376     g.DrawString(this._controller.WrongHitsRemaining.ToString(),
        ↳ new Font("Arial", 20), new SolidBrush(Color.Black), new
        ↳ Point((this.Width / 2) - 10, errorsY));
377 }
378

```

```

379     /// <summary>
380     /// Draw the Bpm reached
381     /// </summary>
382     /// <param name="g">The graphic object where the drawing is
383         ↳ performed</param>
384     private void DrawReachedBpm(Graphics g)
385     {
386         string label = "BPM: " + this._controller.Bpm;
387         Font labelFont = new Font("Arial", 28);
388         SizeF labelSize = new SizeF();
389         labelSize = g.MeasureString(label, labelFont);
390
391         int labelX = (int)(this.Size.Width -
392             ↳ Size.Round(labelSize).Width);
393         int labelY = (int)Math.Round(this.Size.Height * 0.9);
394
395         g.DrawString(label, labelFont, new SolidBrush(Color.Black),
396             ↳ new Point(labelX, labelY));
397     }
398
399     /// <summary>
400     /// Time that a note takes going from the top of the screen to
401         ↳ the bottom in ms
402     /// </summary>
403     public float NoteRideTime
404     {
405         get => ((60000 / this._controller.Bpm) / 2);
406     }
407
408     /// <summary>
409     /// Returns the distance from HitSpotY in which a note is
410         ↳ considered as not hit
411     /// (therefore the hit is wasted)
412     /// </summary>
413     private int WastedDistance
414     {
415         get {
416             double diagonalSpace =
417                 ↳ Math.Sqrt(Math.Pow(this.LeftHitSpotX, 2) +
418                 ↳ Math.Pow(this.HitSpotY, 2));
419             double speed = diagonalSpace / this.NoteRideTime;
420             return (int)Math.Round(speed *
421                 ↳ this._controller.HittedNoteInterval);
422         }
423     }
424
425     /// <summary>
426     /// Put down the left stick

```

```

419     /// </summary>
420     public void LeftNoteHit()
421     {
422         this._leftStickDown = true;
423         this.PlayHitSound();
424         this.Hit(INote.notePosition.Left);
425
426         Timer t = new Timer();
427         t.Interval = STICK_DOWN_MS;
428         t.Tick += (s, e) =>
429         {
430             this._leftStickDown = false;
431             t.Stop();
432         };
433         t.Start();
434     }
435
436     /// <summary>
437     /// Put down the right stick
438     /// </summary>
439     public void RightNoteHit()
440     {
441         this._rightStickDown = true;
442         this.PlayHitSound();
443         this.Hit(INote.notePosition.Right);
444
445         Timer t = new Timer();
446         t.Interval = STICK_DOWN_MS;
447         t.Tick += (s, e) =>
448         {
449             this._rightStickDown = false;
450             t.Stop();
451         };
452         t.Start();
453     }
454
455     /// <summary>
456     /// Method called when an hit is performed
457     /// Check whether the hit is counted or wasted
458     /// </summary>
459     /// <param name="position">The position if the hit</param>
460     private void Hit(INote.notePosition position)
461     {
462         try
463         {
464             this._notesMutex.WaitOne();
465
466             if (this._notes.Count == 0)

```

```

467         this._controller.EmptyHit();
468     else
469     {
470         Triplet<INote, Point, int> bottomNote =
471             ↳ this._notes.First.Value;
472
473         if (bottomNote.Item1.Position == position &&
474             Math.Abs(bottomNote.Item2.Y - this.HitSpotY) <
475             ↳ this.WastedDistance)
476         {
477             int timestamp =
478                 ↳ (int)(DateTime.UtcNow.Subtract(new
479                 ↳ DateTime(1970, 1, 1))).TotalSeconds;
480             int hitDelay = Math.Abs(timestamp -
481                 ↳ bottomNote.Item3);
482             this._controller.NoteHitted(bottomNote.Item1,
483                 ↳ hitDelay);
484             this._notes.RemoveFirst();
485         }
486     else
487         this._controller.EmptyHit();
488     }
489     this._notesMutex.ReleaseMutex();
490 } catch (GameEndedException)
491 {
492     this._notesMutex.ReleaseMutex();
493     MessageBox.Show("La tua partita termina qui! Il tuo
494         ↳ punteggio è di " +
495         ↳ this._controller.Score.ToString());
496
497     if(Game.LoadBestResults() != null)
498     {
499         if (this._controller.Score >
500             ↳ (Game.LoadBestResults().ToArray())[0].Item1)
501             MessageBox.Show("Complimenti
502                 ↳ "+this._controller.PlayerName +", hai
503                 ↳ stabilito il nuovo record!\n"
504                 ↳ + "\n\nNuovo record:
505                 ↳ "+this._controller.PlayerName + " " +
506                 ↳ this._controller.Score.ToString());
507     }
508
509     this.StopGame();
510 }
511
512 /// <summary>
513 /// Plays the sound of a snare hit

```



```

502     /// </summary>
503     private void PlayHitSound()
504     {
505         if (this.IsRunning)
506         {
507             this._outputDevice.Play();
508             this._audio.Position = 0;
509         }
510     }
511
512     /// <summary>
513     /// Return wether the game is running
514     /// </summary>
515     public bool IsRunning
516     {
517         get => this._isRunning;
518     }
519
520     /// <summary>
521     /// Puts the game in pause
522     /// </summary>
523     public void PauseGame()
524     {
525         this._isRunning = false;
526         this._noteGenerator.Pause();
527         this._gameLoopTimer.Stop();
528     }
529
530     /// <summary>
531     /// Resume from pause
532     /// </summary>
533     public void ResumeGame()
534     {
535         this._isRunning = true;
536         this._notes.Clear();
537         this.Draw();
538         this._gameLoopTimer.Start();
539         this._noteGenerator.Resume();
540     }
541
542     /// <summary>
543     /// Start the game
544     /// </summary>
545     public void StartGame()
546     {
547         this._notesMutex = new System.Threading.Mutex();
548         this._notes.Clear();
549         this._isRunning = true;

```

```

550
551         try {
552             this._controller.StartGame();
553             this._noteGenerator.Start();
554             this._gameLoopTimer.Start();
555         } catch (GameOptionException)
556         {
557             MessageBox.Show("Impossibile iniziare il gioco: bpm
558                             → iniziali o nome non settati!");
559         }
560
561         /// <summary>
562         /// Stops the game
563         /// </summary>
564         public void StopGame()
565         {
566             this._isRunning = false;
567             this._notesMutex.Close();
568             this._gameLoopTimer.Stop();
569             this._noteGenerator.Stop();
570             this._controller.StopGame();
571         }
572     }
573 }

```

```

1  using System;
2  using System.Drawing;
3  using System.Windows.Forms;
4
5  namespace MasterDrums.View
6  {
7      /// <summary>
8      /// The panel that shows the paused game options which are resume or
9      /// → quit
10     /// </summary>
11     class GamePausePanel : TableLayoutPanel
12     {
13         private IMainView _mainView;
14         private Button _btnResume;
15         private Button _btnQuit;
16
17         /// <summary>
18         /// The game pause panel is a table with one column and 5 rows.
19         /// The 1st, 2nd and 3rd row are used as a spacing row.
20         /// </summary>

```

```

20     public GamePausePanel(IMainView mainView) : base()
21     {
22         this._mainView = mainView;
23
24         this.TableSetup();
25         this.ButtonsSetup();
26
27         this.BackColor = Color.DarkGray;
28     }
29
30     /// <summary>
31     /// Sets up the table related properties
32     /// </summary>
33     private void TableSetup()
34     {
35         /// Table structure:
36         ///                                20%
37         /// 2   btnResume                20%
38         ///                                20%
39         /// 4   btnQuit                  20%
40         ///                                20%
41
42         this.SuspendLayout();
43         this.ColumnCount = 1;
44         this.ColumnStyles.Add(new ColumnStyle(SizeType.Percent,
45         ↪ 100F));
46
47         this.RowCount = 5;
48
49         this.RowStyles.Add(new RowStyle(SizeType.Percent, 20F));
50         // resume button
51         this.RowStyles.Add(new RowStyle(SizeType.Percent, 20F));
52         // quit button
53         this.RowStyles.Add(new RowStyle(SizeType.Percent, 20F));
54         this.RowStyles.Add(new RowStyle(SizeType.Percent, 20F));
55
56         this.ResumeLayout();
57     }
58
59     /// <summary>
60     /// Apply padding font and docking properties to the control
61     /// </summary>
62     /// <param name="c">The input control where the styles are
63         ↪ applied</param>
64     /// <returns>The modified control</returns>
65     private Control ApplyStyle(Control c)
66     {
67         c.Dock = DockStyle.Fill;

```

```

66         c.Margin = new Padding(10);
67         c.Font = new Font("Microsoft Sans Serif", 20F,
        ↪      FontStyle.Regular, GraphicsUnit.Pixel, 0);
68         return c;
69     }
70
71     /// <summary>
72     /// Sets up the buttons
73     /// </summary>
74     private void ButtonsSetup()
75     {
76         this.SuspendLayout();
77
78         #region Game mode controls
79         // TODO
80         #endregion
81
82         #region Resume button
83         this._btnResume = new Button();
84         this._btnResume.Text = "Riprendi";
85         this._btnResume.UseVisualStyleBackColor = true;
86         this._btnResume.Click += new EventHandler(this.ResumeOrQuit);
87         this.ApplyStyle(this._btnResume);
88
89         this.Controls.Add(this._btnResume, 0, 1);
90         #endregion
91
92         #region Quit button
93         this._btnQuit = new Button();
94         this._btnQuit.Text = "Abbandona";
95         this._btnQuit.UseVisualStyleBackColor = true;
96         this._btnQuit.Click += new EventHandler(this.ResumeOrQuit);
97         this.ApplyStyle(this._btnQuit);
98
99         this.Controls.Add(this._btnQuit, 0, 3);
100        #endregion
101
102
103        this.ResumeLayout();
104    }
105
106    /// <summary>
107    /// Handles when the user clicks on the resume or quit button
108    /// </summary>
109    private void ResumeOrQuit(object sender, EventArgs e)
110    {
111        Button clickedButton = sender as Button;
112        if (clickedButton.Text == "Riprendi")

```

```

113         {
114             this._mainView.ResumeGame();
115         }
116         if (clickedButton.Text == "Abbandona")
117         {
118             DialogResult dialogResult = MessageBox.Show("Vuoi davvero
119                 ↳ abbandonare la partita?", "Attenzione",
120                 ↳ MessageBoxButtons.YesNo);
121             if (dialogResult == DialogResult.Yes)
122             {
123                 this._mainView.StopGame();
124             }
125             else if (dialogResult == DialogResult.No)
126             {
127                 // nothing to do
128             }
129         }
130     }
131 }
132 }

```

```

1  using MasterDrums.Model;
2  using System;
3  using System.Collections.Generic;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  namespace MasterDrums.View
8  {
9      class HighscoresPanel : TableLayoutPanel
10     {
11         private IMainView _mainView;
12         private ListView _records;
13         private Button _btnBack;
14
15         /// <summary>
16         /// Highscores panel is a table with one column and 2 rows.
17         /// The 1st row contains a listview with the game results,
18         /// the 2nd contains a back button to return to the main menu
19         /// </summary>
20         public HighscoresPanel(IMainView mainView) : base()
21         {
22             this._mainView = mainView;

```

```

23         this.TableSetup();
24         this.ListViewSetup();
25         this.ButtonSetup();
26     }
27
28     /// <summary>
29     /// Sets up the table related properties
30     /// </summary>
31     private void TableSetup()
32     {
33         /// Table structure:
34         /// Listview           80%
35         ///                 5%
36         /// btnBack         15%
37
38         this.SuspendLayout();
39         this.ColumnCount = 1;
40         this.ColumnStyles.Add(new ColumnStyle(SizeType.Percent,
41         ↳ 100F));
42         this.RowCount = 3;
43
44         // listview
45         this.RowStyles.Add(new RowStyle(SizeType.Percent, 80F));
46         // blank space
47         this.RowStyles.Add(new RowStyle(SizeType.Percent, 5F));
48         // back button
49         this.RowStyles.Add(new RowStyle(SizeType.Percent, 15F));
50
51         this.ResumeLayout();
52     }
53
54     /// <summary>
55     /// Sets up the listview containing the records
56     /// </summary>
57     private void ListViewSetup()
58     {
59
60         this.SuspendLayout();
61
62         this._records = new ListView();
63         this._records.View = System.Windows.Forms.View.Details;
64         this._records.GridLines = true;
65
66         // Add columns
67         this._records.Columns.Add("Nome", 300);
68         this._records.Columns.Add("Punteggio", 300);
69
70         List<Tuple<int, String>> records = Game.LoadBestResults();
71         String[] items = new string[2];

```

```

70
71     // Add rows
72     if (records != null)
73     {
74         foreach (Tuple<int, String> t in records)
75         {
76             items[0] = t.Item2;
77             items[1] = t.Item1.ToString();
78             this._records.Items.Add(new ListViewItem(items));
79         }
80     }
81     this.ApplyStyle(this._records);
82     this.Controls.Add(this._records, 0, 0);
83
84     this.ResumeLayout();
85 }
86
87
88     /// </summary>
89     /// <param name="c">The input control where the styles are
90     ↪ applied</param>
91     /// <returns>The modified control</returns>
92     private Control ApplyStyle(Control c)
93     {
94         c.Dock = DockStyle.Fill;
95         c.Margin = new Padding(10);
96         c.Font = new Font("Microsoft Sans Serif", 20F,
97             ↪ FontStyle.Regular, GraphicsUnit.Pixel, 0);
98         return c;
99     }
100
101     /// <summary>
102     /// Sets up the back button
103     /// </summary>
104     private void ButtonSetup()
105     {
106         this.SuspendLayout();
107         this._btnBack = new Button();
108         this._btnBack.Text = "Indietro";
109         this._btnBack.Click += new EventHandler((s, e) =>
110             ↪ this._mainView.MainMenu());
111         this.ApplyStyle(this._btnBack);
112         this.Controls.Add(this._btnBack, 0, 2);
113         this.ResumeLayout();
114     }
115 }

```

```

1  using System;
2  using System.Windows.Forms;
3  using System.Drawing;
4
5  namespace MasterDrums.View
6  {
7      class MainMenuPanel: TableLayoutPanel
8      {
9          private IMainView _mainView;
10
11          /// <summary>
12          /// Main menu panel is a table with one column and 4 rows.
13          /// The 3rd row is used as a spacing row.
14          /// </summary>
15          public MainMenuPanel(IMainView mainView) : base()
16          {
17              this._mainView = mainView;
18
19              this.TableSetup();
20              this.ButtonsSetup();
21
22              this.BackColor = Color.DarkGray;
23          }
24
25          /// <summary>
26          /// Sets up the table related properties
27          /// </summary>
28          private void TableSetup()
29          {
30              this.SuspendLayout();
31              this.ColumnCount = 1;
32              this.ColumnStyles.Add(new ColumnStyle(SizeType.Percent,
33              ↪ 100F));
34              this.RowCount = 4;
35              this.RowStyles.Add(new RowStyle(SizeType.Percent, 30F));
36              this.RowStyles.Add(new RowStyle(SizeType.Percent, 30F));
37              this.RowStyles.Add(new RowStyle(SizeType.Percent, 10F));
38              this.RowStyles.Add(new RowStyle(SizeType.Percent, 30F));
39              this.ResumeLayout();
40          }
41
42          /// <summary>
43          /// Sets up the buttons
44          /// </summary>
45          private void ButtonsSetup()
46          {

```



```

46         this.SuspendLayout();
47
48         Button buttonNewGame = new Button();
49         buttonNewGame.Dock = DockStyle.Fill;
50         buttonNewGame.Font = new Font("Microsoft Sans Serif", 20F,
51             ↳ FontStyle.Regular, GraphicsUnit.Pixel, 0);
52         buttonNewGame.Margin = new Padding(10);
53         buttonNewGame.Text = "Nuova partita";
54         buttonNewGame.UseVisualStyleBackColor = true;
55         buttonNewGame.Click += new EventHandler((s, e) =>
56             ↳ this._mainView.NewGame());
57         this.Controls.Add(buttonNewGame, 0, 0);
58
59         Button buttonHighscores = new Button();
60         buttonHighscores.Dock = DockStyle.Fill;
61         buttonHighscores.Font = new Font("Microsoft Sans Serif", 20F,
62             ↳ FontStyle.Regular, GraphicsUnit.Pixel, 0);
63         buttonHighscores.Margin = new Padding(10);
64         buttonHighscores.Text = "Record";
65         buttonHighscores.UseVisualStyleBackColor = true;
66         buttonHighscores.Click += new EventHandler((s, e) =>
67             ↳ this._mainView.Highscores());
68         this.Controls.Add(buttonHighscores, 0, 1);
69
70         Button buttonQuit = new Button();
71         buttonQuit.Dock = DockStyle.Fill;
72         buttonQuit.Font = new Font("Microsoft Sans Serif", 20F,
73             ↳ FontStyle.Regular, GraphicsUnit.Pixel, 0);
74         buttonQuit.Margin = new Padding(10);
75         buttonQuit.Text = "Esci";
76         buttonQuit.UseVisualStyleBackColor = true;
77         buttonQuit.Click += new EventHandler((s, e) =>
78             ↳ this._mainView.Quit());
79         this.Controls.Add(buttonQuit, 0, 3);
80
81         this.ResumeLayout();
82     }
83 }
84
85 }
86
87 }

```

```

1  using MasterDrums.Model;
2  using System;
3  using System.Windows.Forms;
4  using System.Drawing;
5

```

```

6 namespace MasterDrums.View
7 {
8     class NewGamePanel : TableLayoutPanel
9     {
10         private IMainView _mainView;
11         private TextBox _txtUsername;
12         private NumericUpDown _txtInitialBpm;
13         private ComboBox _gameModeSelection;
14
15         /// <summary>
16         /// New game panel is a table with one column and 8 rows.
17         /// </summary>
18         public NewGamePanel(IMainView mainView) : base()
19         {
20             this._mainView = mainView;
21
22             this.TableSetup();
23             this.ButtonsSetup();
24
25             this.BackColor = Color.DarkGray;
26         }
27
28         /// <summary>
29         /// Sets up the table related properties
30         /// </summary>
31         private void TableSetup()
32         {
33             /// Table structure:
34             ///
35             /// 0    labelName        12%
36             /// 1    txtName          12%
37             ///
38             /// 2    labelMode         12%
39             /// 3    selectMode        12%
40             ///
41             /// 4    labelInitialBpm  12%
42             /// 5    txtInitialBpm    12%
43             ///
44             /// 6    startButton       16%
45             /// 7    backButton        12%
46
47             this.SuspendLayout();
48             this.ColumnCount = 1;
49             this.ColumnStyles.Add(new ColumnStyle(SizeType.Percent,
50             ↪ 100F));
51             this.RowCount = 8;
52
53             // player name

```

```

53         this.RowStyle.Add(new RowStyle(SizeType.Percent, 12F));
54         this.RowStyle.Add(new RowStyle(SizeType.Percent, 12F));
55
56         // game mode
57         this.RowStyle.Add(new RowStyle(SizeType.Percent, 12F));
58         this.RowStyle.Add(new RowStyle(SizeType.Percent, 12F));
59
60         // initial bpm
61         this.RowStyle.Add(new RowStyle(SizeType.Percent, 12F));
62         this.RowStyle.Add(new RowStyle(SizeType.Percent, 12F));
63
64         // start button
65         this.RowStyle.Add(new RowStyle(SizeType.Percent, 16F));
66
67         // back button
68         this.RowStyle.Add(new RowStyle(SizeType.Percent, 12F));
69
70         this.ResumeLayout();
71     }
72
73     /// <summary>
74     /// Apply padding font and docking properties to the control
75     /// </summary>
76     /// <param name="c">The input control where the styles are
77         applied</param>
78     /// <returns>The modified control</returns>
79     private Control ApplyStyle(Control c)
80     {
81         c.Dock = DockStyle.Fill;
82         c.Margin = new Padding(10);
83         c.Font = new Font("Microsoft Sans Serif", 20F,
84             ↪ FontStyle.Regular, GraphicsUnit.Pixel, 0);
85         return c;
86     }
87
88     /// <summary>
89     /// Sets up the buttons
90     /// </summary>
91     private void ButtonsSetup()
92     {
93         this.SuspendLayout();
94
95         #region Username controls
96         Label labelUsername = new Label();
97         labelUsername.Text = "Nome del giocatore";
98         this.ApplyStyle(labelUsername);
99
100        this._txtUsername = new TextBox();

```

```

99         this._txtUsername.Clear();
100         this.ApplyStyle(this._txtUsername);
101
102         this.Controls.Add(labelUsername, 0, 0);
103         this.Controls.Add(this._txtUsername, 0, 1);
104         #endregion
105
106         #region Game mode controls
107         Label labelGameMode = new Label();
108         labelGameMode.Text = "Modalità di gioco";
109         this.ApplyStyle(labelGameMode);
110
111         this._gameModeSelection = new ComboBox();
112         this.ApplyStyle(this._gameModeSelection);
113         this._gameModeSelection.Items.Add(new RandomNoteGenerator());
114         this._gameModeSelection.Items.Add(new
115             ↪ AlternatedHandNoteGenerator());
116         this._gameModeSelection.SelectedIndex = 0;
117
118         this.Controls.Add(labelGameMode, 0, 2);
119         this.Controls.Add(this._gameModeSelection, 0, 3);
120         #endregion
121
122         #region Initial bpm controls
123         Label labelInitialBpm = new Label();
124         labelInitialBpm.Text = "BPM Iniziale";
125         this.ApplyStyle(labelInitialBpm);
126
127         this._txtInitialBpm = new NumericUpDown();
128         this._txtInitialBpm.Minimum = INoteGenerator.MIN_BPM;
129         this._txtInitialBpm.Maximum = INoteGenerator.MAX_BPM;
130         this._txtInitialBpm.ReadOnly = true;
131         this.ApplyStyle(this._txtInitialBpm);
132
133         this.Controls.Add(labelInitialBpm, 0, 4);
134         this.Controls.Add(this._txtInitialBpm, 0, 5);
135         #endregion
136
137         #region Confirm controls
138         Button buttonConfirm = new Button();
139         buttonConfirm.Text = "Inizia";
140         buttonConfirm.UseVisualStyleBackColor = true;
141         buttonConfirm.Click += new EventHandler(this.Confirm);
142         this.ApplyStyle(buttonConfirm);
143
144         this.Controls.Add(buttonConfirm, 0, 6);
145         #endregion

```

```

146         #region Back option
147         Button buttonBack = new Button();
148         buttonBack.Text = "Indietro";
149         buttonBack.UseVisualStyleBackColor = true;
150         buttonBack.Click += new EventHandler((s, e) =>
151             ↪ this._mainView.MainMenu());
152         this.ApplyStyle(buttonBack);
153
154         this.Controls.Add(buttonBack, 0, 7);
155         #endregion
156
157         this.ResumeLayout();
158     }
159
160     /// <summary>
161     /// Used when the user click the confirm button to start new
162     ↪ game
163     /// </summary>
164     /// <param name="sender"></param>
165     /// <param name="e"></param>
166     private void Confirm(object sender, EventArgs e)
167     {
168         if (!(string.IsNullOrEmpty(this._txtUsername.Text)))
169         {
170             string name = this._txtUsername.Text;
171             int initialBpm = (int)this._txtInitialBpm.Value;
172
173             if (this._gameModeSelection.SelectedItem != null)
174             {
175                 INoteGenerator gameMode =
176                 ↪ (INoteGenerator)this._gameModeSelection.SelectedItem;
177                 this._mainView.StartGame(name, initialBpm, gameMode);
178             }
179             else
180             {
181                 MessageBox.Show("E' necessario inserire la modalità
182                 ↪ per iniziare una partita!");
183             }
184             else
185             {
186                 MessageBox.Show("E' necessario inserire il nome del
187                 ↪ giocatore per iniziare una partita!");
188             }
189         }
190     }
191
192     /// <summary>
193     /// Clear the player name text box
194     /// </summary>
195     public void ClearTxtUsername()

```

```

189         {
190             this._txtUsername.Clear();
191         }
192     }
193 }

```

4.4 Exception

```

1  using System;
2
3  namespace MasterDrums.Exception
4  {
5      /// <summary>
6      /// Exception launched when the user lose the game and it must end
7      /// </summary>
8      public class GameEndedException : SystemException
9      {
10         public GameEndedException() : base() { }
11     }
12 }

```

```

1  using System;
2
3  namespace MasterDrums.Model
4  {
5      /// <summary>
6      /// Exception raised when the user name or the initial bpms are not
7      /// set
8      /// </summary>
9      public class GameOptionException : SystemException
10     {
11         public GameOptionException(string msg) : base(msg) { }
12     }
13 }

```

4.5 Utils

```
1 namespace MasterDrums.Utils
2 {
3     /// <summary>
4     /// Interface for the Observable actor used for the Observable
5     /// pattern.
6     /// </summary>
7     public interface IObserver
8     {
9         /// <summary>
10        /// The abstract method used to update the internal state based
11        /// on the Observable object.
12        /// </summary>
13        /// <param name="subj">The observed object.</param>
14        void Update(ISubject subj);
15    }
16 }
```

```
1 using System.Collections.Generic;
2
3 namespace MasterDrums.Utils
4 {
5     /// <summary>
6     /// The abstract class that represents the Subject actor in the
7     /// Observer pattern.
8     /// </summary>
9     public abstract class ISubject
10    {
11        /// <summary>
12        /// List of actors that are currently observing.
13        /// </summary>
14        private List<IObserver> observers = new List<IObserver>();
15
16        /// <summary>
17        /// Method used by an actor to start observing the subject.
18        /// </summary>
19        /// <param name="obs">The actor who's going to observe</param>
20        public void Attach(IObserver obs) {
21            observers.Add(obs);
22        }
23
24        /// <summary>
25        /// Method used by an actor to stop observing the subject.
26        /// </summary>
27    }
28 }
```

```

26      /// <param name="obs">The actor who's going to stop to
    ↪ observe</param>
27      public void Detach(IObserver obs) {
28          observers.Remove(obs);
29      }
30
31      /// <summary>
32      /// Method used to notify the observers that the internal state
    ↪ has changed
33      /// </summary>
34      public void Notify()
35      {
36          foreach (IObserver obs in observers)
37              obs.Update(this);
38      }
39  }
40 }

```

```

1  using System.Drawing;
2  using System.Drawing.Drawing2D;
3
4  namespace MasterDrums.Utils
5  {
6      /// <summary>
7      /// Class implementing some static methods useful for image
    ↪ processing
8      /// </summary>
9      class ImageUtils
10     {
11         /// <summary>
12         /// Method to rotate an image either clockwise or
    ↪ counter-clockwise
13         /// taken from
    ↪ https://stackoverflow.com/questions/2163829/how-do-i-rotate-a-picture-in-winforms
14         /// </summary>
15         /// <param name="img">the image to be rotated</param>
16         /// <param name="rotationAngle">the angle (in degrees).
17         /// NOTE:
18         /// Positive values will rotate clockwise
19         /// negative values will rotate counter-clockwise
20         /// </param>
21         /// <returns></returns>
22         public static Image RotateImage(Image img, float rotationAngle)
23         {
24             //create an empty Bitmap image
25             Bitmap bmp = new Bitmap(img.Width, img.Height);

```



```

26
27      //turn the Bitmap into a Graphics object
28      Graphics gfx = Graphics.FromImage(bmp);
29
30      //now we set the rotation point to the center of our image
31      gfx.TranslateTransform((float)bmp.Width / 2,
32          ↪ (float)bmp.Height / 2);
33
34      //now rotate the image
35      gfx.RotateTransform(rotationAngle);
36
37      gfx.TranslateTransform(-(float)bmp.Width / 2,
38          ↪ -(float)bmp.Height / 2);
39
40      //set the InterpolationMode to HighQualityBicubic so to
41      ↪ ensure a high
42      //quality image once it is transformed to the specified size
43      gfx.InterpolationMode = InterpolationMode.HighQualityBicubic;
44
45      //now draw our new image onto the graphics object
46      gfx.DrawImage(img, new Point(0, 0));
47
48      //dispose of our Graphics object
49      gfx.Dispose();
50
51      //return the image
52      return bmp;
53  }
54 }
55 }

```

```

1  namespace MasterDrums.Utils
2  {
3      /// <summary>
4      /// Class representing a tuple of three objects (a triplet)
5      /// The following is used in place of c# Tuple because Tuple's are
6      ↪ immutable,
7      /// Triplet's elements will be mutable.
8      /// </summary>
9      /// <typeparam name="T1">Type of the first object</typeparam>
10     /// <typeparam name="T2">Type of the second object</typeparam>
11     /// <typeparam name="T3">Type of the third object</typeparam>
12     class Triplet<T1, T2, T3>
13     {
14         public Triplet(T1 i1, T2 i2, T3 i3)
15         {

```

```

15         this.Item1 = i1;
16         this.Item2 = i2;
17         this.Item3 = i3;
18     }
19
20     public T1 Item1 { get; set; }
21     public T2 Item2 { get; set; }
22     public T3 Item3 { get; set; }
23 }
24 }

```

4.6 Program

```

1  using System;
2  using MasterDrums.Controller;
3  using MasterDrums.View;
4  using System.Windows.Forms;
5
6  namespace MasterDrums
7  {
8      static class Program
9      {
10         /// <summary>
11         /// Application entry point.
12         /// Creates the view and the controller and connects them.
13         /// The model is created in the controller.
14         /// </summary>
15         [STAThread]
16         static void Main()
17         {
18             MainController controller = new MainController();
19             MainView view = new MainView(controller);
20         }
21     }
22 }

```

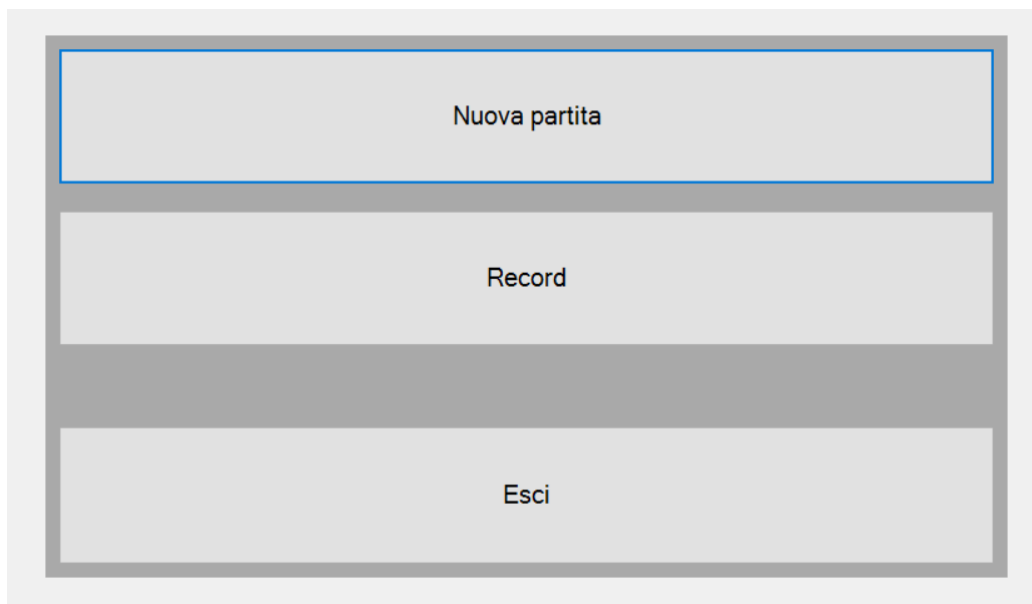
5 Testing

Lo sviluppo del software è stato per gran parte effettuato utilizzando la tecnica del *pair-programming*. Tale tecnica ci ha permesso di analizzare ogni scelta implementativa a fondo senza effettuare nessuna scelta in modo approssimativo; il risultato è un software la cui probabilità di presenza di errori è piuttosto bassa.

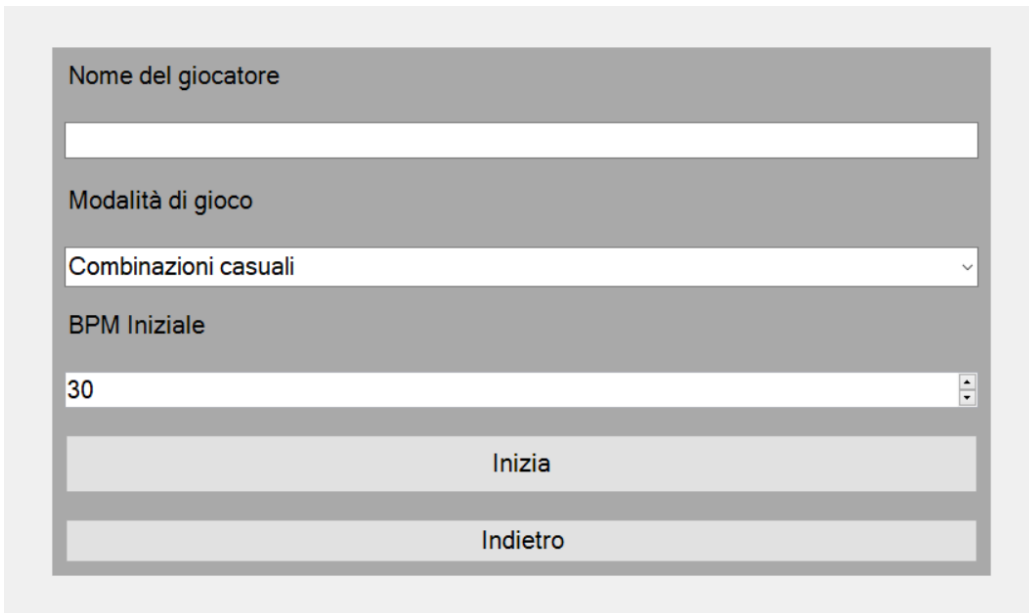
Il testing di tale software è stato effettuato inizialmente in modalità **whitebox** durante la fase di sviluppo, testando ogni condizione limite che si sarebbe potuta presentare. Ciò ha permesso di arginare in fase iniziale la maggior parte degli errori che sarebbero potuti emergere a lavoro completato.

In seguito il software, giunti a quella che è la versione finale, è stato testato con le modalità **blackbox**. Nella sottosezioni seguenti verranno mostrati vari *screenshot* del software corrispondenti ai vari casi d'uso identificati nella sezione corrispondente.

5.1 Inizio di una partita



Cliccando sul pulsante *Nuova partita* viene mostrata la schermata seguente



A screenshot of a game setup screen. It contains the following elements:

- A label "Nome del giocatore" above a text input field.
- A label "Modalità di gioco" above a dropdown menu.
- The dropdown menu is currently set to "Combinazioni casuali" with a downward arrow.
- A label "BPM Iniziale" above a text input field.
- The text input field contains the number "30" and has a small up/down arrow icon on the right.
- At the bottom, there are two buttons: "Inizia" and "Indietro".

5.2 Visualizzazione punteggi

Nel caso l'utente abbia già giocato almeno una partita vengono mostrati i punteggi totalizzati.

Nome	Punteggio	
Marco	4680	

Indietro

Nel caso invece l'utente non abbia ancora giocato nessuna partita viene mostrato un messaggio che comunica l'assenza di punteggi registrati.

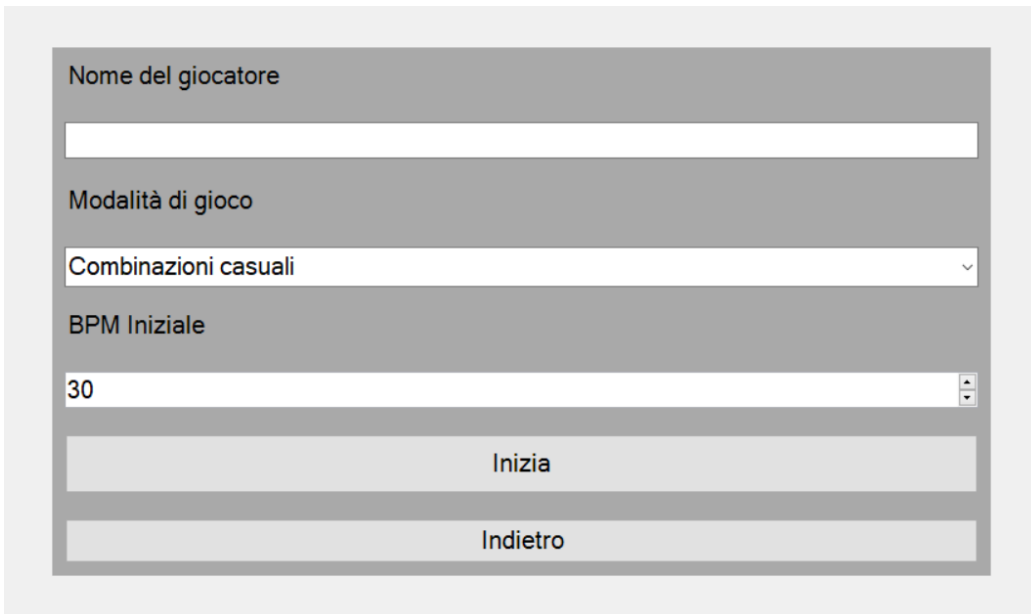
Nuova partita

Nessun risultato presente!

OK

Esci

5.3 Impostazione di nome, modalità di gioco e BPM iniziali



Nome del giocatore

Modalità di gioco

Combinazioni casuali

BPM Iniziale

30

Inizia

Indietro

Nel caso l'utente tenti di procedere senza aver inserito il proprio nome la partita non viene iniziata.



Nome del giocatore

Modalità di gioco

Combinazioni casuali

BPM Iniziale

30

Inizia

Indietro

×

E' necessario inserire il nome del giocatore per iniziare una partita!

OK

5.4 Effettuazione colpo



5.5 Fine della partita



5.6 Partita in pausa



5.7 Partita ripresa da una pausa



5.8 Partita abbandonata da una pausa



BPM: 32

6 Compilazione ed esecuzione

Gli strumenti utilizzati per la compilazione del programma sono i seguenti:

- **Ambiente di sviluppo:** Visual Studio Community 2017
- **Versione:** 15.9.28307.905
- **Framework:** .NET Framework 4.5.2

Per la compilazione della soluzione, è necessario recarsi nel seguente path dell'ambiente di sviluppo: nella barra dei menù selezionare *Compila > Compila soluzione*. In alternativa, se non si compila manualmente la soluzione, all'avvio del programma (tramite apposita icona su Visual Studio), la compilazione avviene in maniera automatica.

Per eseguire l'applicazione si può utilizzare l'ambiente di sviluppo oppure ricorrere all'eseguibile presente nella cartella del progetto al percorso `./MasterDrums/MasterDrums/bin/Debug/MasterDrums.exe`.

I requisiti minimi per l'esecuzione del programma sono i seguenti:

- Sistema operativo: Windows 7 o successivi
- Architettura: 32 o 64 bit
- Framework: .NET Framework 4.0

Non vi sono requisiti di performance particolari, tuttavia si rimanda a consultare i requisiti minimi per il .NET Framework al sito Microsoft tenendo in considerazione che il programma utilizza al più 60 MB di memoria RAM.

Il software è stato testato su un computer con la seguente scheda tecnica:

- CPU: Intel(R) Core(TM) i5-8250U CPU @ 1.80 GHz
- RAM: 8GB DDR3

- GPU: Intel UHD Graphics 620
- SO: Windows 10 Home, Build 1903
- Architettura: 64 bit