

Relazione:
“DUKEMANIA”

Gianluca Migliarini,
Serafino Pandolfini,
Sofia Tosi,
Laura Leonardi,
Alessandro Brasini

25 Agosto 2021

Indice

1 Analisi	3
1.1 Requisiti	3
1.2 Analisi e modello del dominio	4
2 Design	5
2.1 Architettura	5
2.2 Design dettagliato	7
2.2.1 Gianluca Migliarini	7
2.2.2 Serafino Pandolfini	11
2.2.3 Sofia Tosi	16
2.2.4 Laura Leonardi	18
2.2.5 Alessandro Brasini	21
3 Sviluppo	27
3.1 Testing	27
3.2 Metodologia di lavoro	28
3.2.1 Gestione del lavoro	28
3.2.2 Gianluca Migliarini	29
3.2.3 Serafino Pandolfini	30
3.2.4 Sofia Tosi	31
3.2.5 Laura Leonardi	32
3.2.6 Alessandro Brasini	33
3.3 Note di sviluppo	34
3.3.1 Gianluca Migliarini	34
3.3.2 Serafino Pandolfini	35
3.3.3 Sofia Tosi	36
3.3.4 Laura Leonardi	37
3.3.5 Alessandro Brasini	38

4	Commenti Finali	39
4.1	Gianluca Migliarini	39
4.1.1	commenti	39
4.1.2	difficoltà riscontrate	39
4.2	Serafino Pandolfini	40
4.2.1	commenti	40
4.2.2	difficoltà riscontrate	40
4.3	Sofia Tosi	41
4.3.1	commenti	41
4.3.2	difficoltà riscontrate	41
4.4	Laura Leonardi	42
4.4.1	difficoltà riscontrate	42
4.5	Alessandro Brasini	43
4.5.1	commenti	43
4.5.2	difficoltà riscontrate	43
5	Appendice - Guida Utente	44

Capitolo 1

Analisi

1.1 Requisiti

Il gruppo si pone come obiettivo quello di realizzare un clone del videogioco musicale "BeatmaniaIIDX". Lo sviluppo mira alla completa funzionalità a livello di gameplay, e, in aggiunta all'opera originale dalla quale viene presa ispirazione, offrire all'utente la possibilità di importare i propri livelli giocabili tramite file Midi (Musical Instrument Digital Interface). Viene inoltre considerato come requisito la sostituzione della riproduzione di una traccia audio con la sintesi sonora in tempo reale.

Requisiti funzionali

- L'utente è in grado di scegliere quale canzone giocare a partire dalla scelta di un file Midi.
- L'applicativo deve riprodurre fedelmente tutte le tracce componenti la canzone, con un timbro musicale simile a quello di un GameBoy.
- L'applicativo deve offrire un gameplay su varie colonne, il giocatore dovrà premere le note sulla rispettiva colonna a ritmo di musica per ottenere punti.
- L'applicativo deve memorizzare una scoreboard con tutti i punteggi delle varie partite relative alla canzone selezionata.
- La traccia giocabile viene scelta dall'utente dalla lista delle tracce componenti la canzone.
- L'applicativo deve "semplificare" le tracce musicali complesse rendendole giocabili, segnalando un grado di difficoltà

Requisiti non funzionali

- L'applicativo deve funzionare correttamente sia su Windows sia su sistemi basati su Unix
- La fluidità del gameplay e la qualità dell'audio non devono essere compromesse dalla pesantezza del file midi (ovviamente se non troppo complesso)

1.2 Analisi e modello del dominio

Dukemania dovrà essere in grado di fare selezionare all'utente il proprio nickname e una canzone in formato Midi. Dovrà inoltre permettere la configurazione dei nomi di tracce e strumenti associati alla canzone selezionata. Inoltre, la traccia da giocare, sara' anch'essa selezionabile dall'utente. I vari strumenti saranno caricati a partire da un file di configurazione in locale all'avvio del gioco e saranno detti sintetizzatori. Ci sarà una entità responsabile della gestione e riproduzione del tono e del volume in entrata e in uscita. Le tracce della canzone, con le relative note, verranno individuate da un parser. A ogni traccia verrà associato un solo sintetizzatore che riprodurrà uno strumento. Il gameplay si svolgerà su varie colonne, in ognuna delle quali cadranno note relative alla traccia selezionata, opportunamente semplificata per garantirne la giocabilità. Le difficoltà primarie riguarderanno il calcolo del punteggio in base alla precisione del giocatore e la sincronizzazione tra la caduta delle note e il suono riprodotto. Alla fine della partita, dovrà essere visualizzata una schermata contenente il punteggio migliore di ogni singolo giocatore ottenuto sulla canzone selezionata in precedenza.

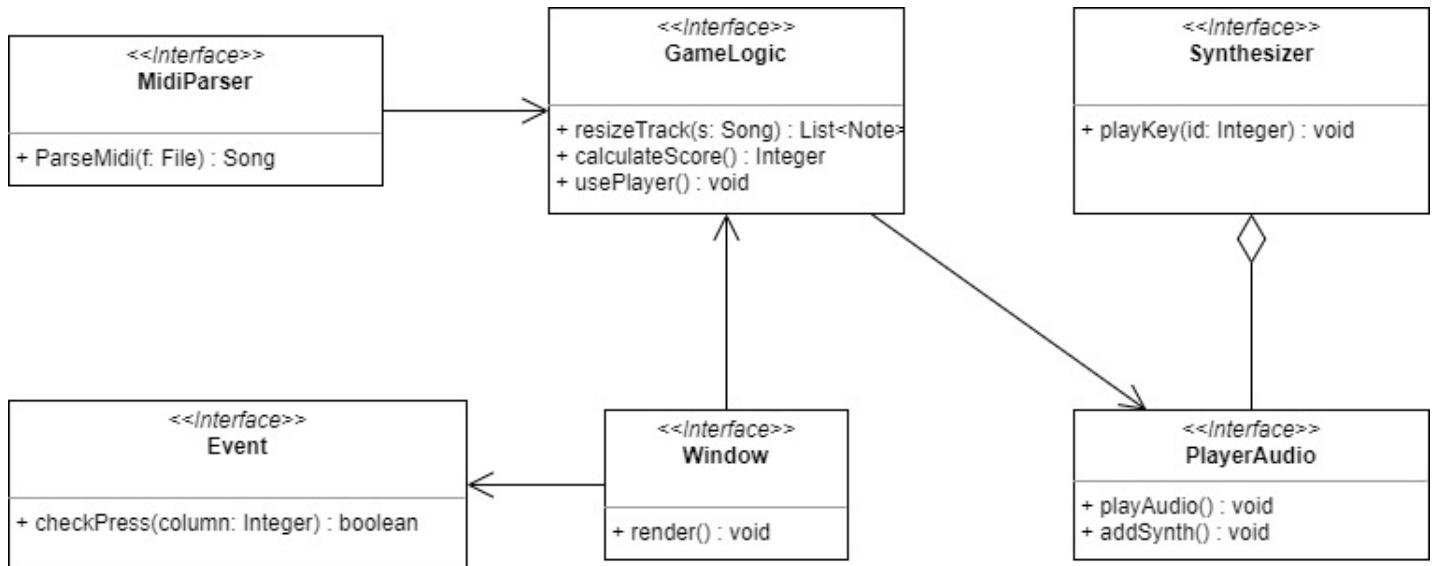


Figura 1.1: Schema UML della fase di analisi.

Capitolo 2

Design

2.1 Architettura

L'architettura di DukeMania è basata sul pattern MVC. La classe DukeMania è l'entry point dell'applicazione, al suo interno vengono inizializzate tutte le View e il WindowManager, la classe che fungerà da Controller principale. Il compito principale di WindowManager è quello di cambiare, tra le varie View, la finestra da renderizzare e visualizzare, occupandosi inoltre di passare il Model alle varie schermate, che a loro volta lo passeranno al relativo Controller. Il Model principale, rappresentato da GameModel, conterrà le configurazioni di gioco, rendendole così accessibili a tutte le View. La classe WindowManager implementa l'interfaccia SwitchWindowNotifier, il cui compito principale è quello di notificare il WindowManager, segnalando gli che un controller di una View vuole sostituire la finestra attiva. Tutte le View implementano l'interfaccia Window, la quale le rende predisposte a ricevere i dati del Model, e ad utilizzare lo SwitchWindowNotifier tramite il proprio controller. Utilizzando questa strategia è semplice aggiungere nuove View e definirne il comportamento. Un sistema di window-switching è già implementato in LibGDX ma, al fine di rendere il più riutilizzabile possibile e platform-indipendent questa meccanica, è stato deciso di implementare una versione generale che può essere applicata a qualsiasi tipo di libreria grafica.

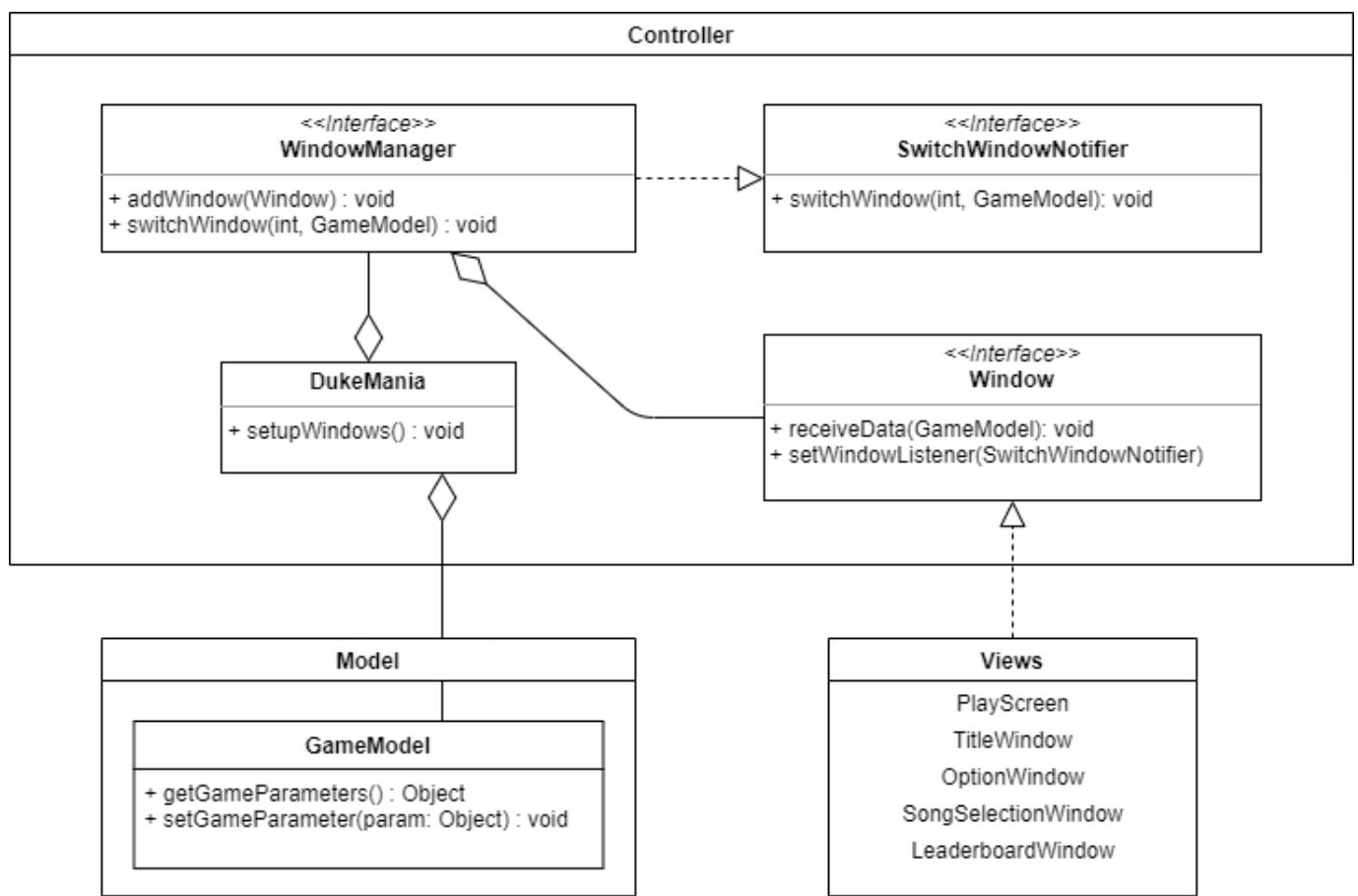


Figura 2.1: Schema UML dell'architettura MVC.

2.2 Design dettagliato

2.2.1 Gianluca Migliarini

Nella mia parte di design dettagliato viene descritto il funzionamento del motore audio. La classe Engine si occupa di aggiornare e valutare lo stato dei vari sintetizzatori (tastiere simulate), ad ogni chiamata del metodo render viene riempito un buffer di samples (campioni audio). Ogni campione audio viene calcolato sommando i campioni audio di tutti i sintetizzatori al momento della chiamata. Una volta che il buffer e' pieno, viene mandato al dispositivo audio che si occupa di suonare i vari campioni. Ogni traccia della canzone viene affidata ad un sintetizzatore di tastiera o di percussioni. Per differenziarli, e' stata creata l' interfaccia Synth, contenente i metodi comuni tra i due, per calcolare quante note/percussioni stanno suonando e per ottenere un campione audio. Le variabili di impostazioni che garantiscono il corretto funzionamento dell'audio sono scritte nella classe Settings.java, come campi finali statici. Per evitare costi computazionali troppo alti, i vari campioni delle varie note di tutti i sintetizzatori vengono pre-caricati in istanze anonime dell'interfaccia BufferManager. Lo scopo di quest'ultima e' quello di rendere una nota risuonabile in ogni momento tramite il metodo refresh, e, tramite i meotodi di iterator, ottenere il prossimo campione suonabile e controllarne l'esistenza.

I sintetizzatori di tastiera e di percussioni si differenziano nel seguente modo:

KeyboardSynth

Per i vari sintetizzatori di tastiere, oltre ai metodi dell'interaccia Synth, e' presente il metodo playTimeNote, per suonare una nota ad una determinata frequenza per un determinato tempo espresso in microsecondi.

Per aumentare le performance, i vari campioni delle note sono caricati all'istanziamento della classe, a partire da una lista contenente gli indici delle note utilizzate e il relativo tempo di suonata.

I BufferManager vengono creati a partire da una classe Enveloper, che si occupa di gestire il volume della nota in entrata e in uscita. Per impostare correttamente tutti i parametri dei sintetizzatori di tastiera ho deciso di utilizzare il pattern "builder", in quanto alcune impostazioni non sono sempre necessarie, ad esempio gli LFO, visti in dettaglio piu' avanti.

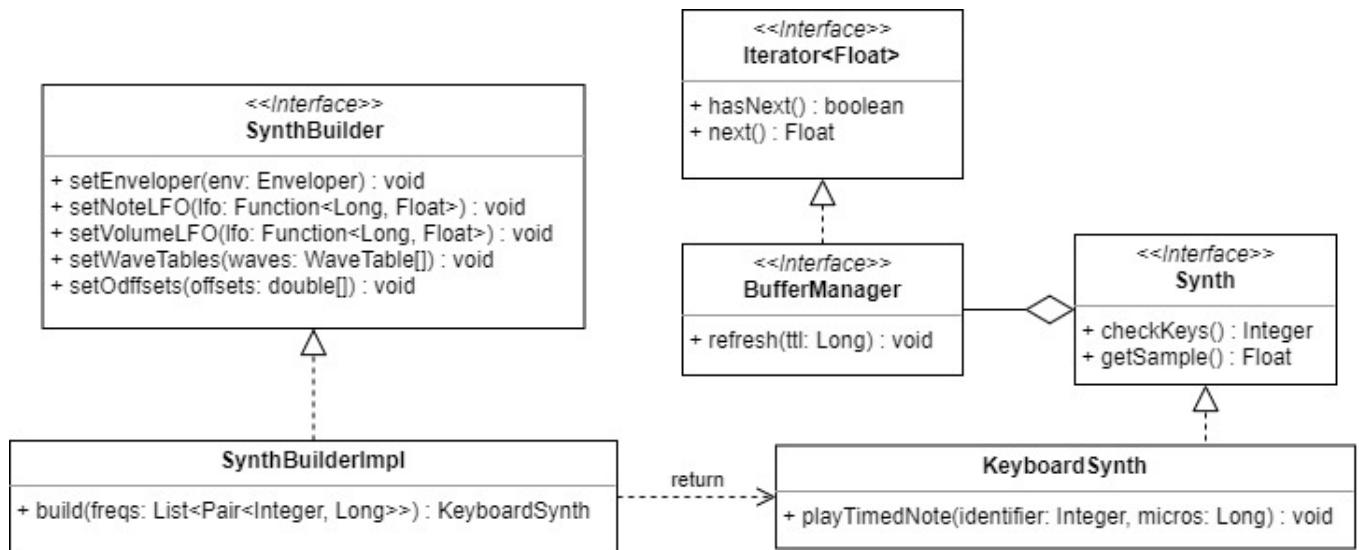


Figura 2.2: Schema UML del Builder dei sintetizzatori di tastiere.

DrumSynth

Nel caso del sintetizzatore per le percussioni, le varie istanze di BufferManager sono presenti come campo read only dell'enum DrumSamples. Ho scelto di utilizzare questo enum come pattern "singleton" in quanto non richiede l'istanziazione di oggetto, e perchè le percussioni non cambiano tra i diversi file Midi, a differenza delle note le quali hanno lunghezze diverse. L'unico metodo esterno all'interfaccia Synth è playPercussion, il quale si occupa di ri-iniziare la riproduzione di un elemento della batteria.

Per aggiungere nuove percussioni, in futuro, basterà aggiungere un nuovo valore all'enum e caricare il relativo BufferManager, questo verrà assegnato automaticamente alle percussioni Midi idonee.

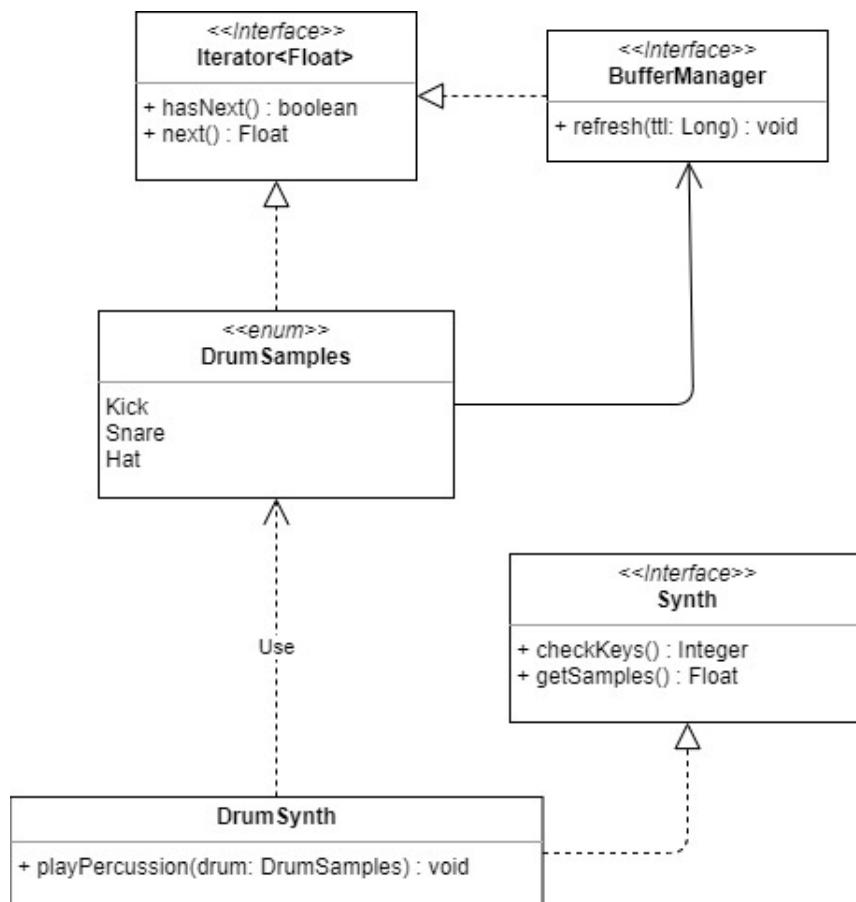


Figura 2.3: Schema UML del Singleton Drum Samples.

Creazione delle note e timbri sonori

Durante il caricamento, i sintetizzatori utilizzano l'enum Wavetables, contenente le varie forme audio utilizzabili assumibili dalle note; queste forme sono caricate all'avvio in array, i quali possono essere letti in intervalli più o meno distaccati per simulare le varie frequenze. Per differenziare i timbri sonori faccio uso degli LFO, funzioni adibite al cambio del tono e del volume nel tempo. Ho deciso di utilizzare il pattern "static factory" a causa della necessità di essere utilizzata senza istanze. Per aggiungere funzioni LFO in futuro, e avere ancora più varietà del suono, SynthBuilder utilizza una strategy con "Function Long,Float" come interfaccia funzionale, la funzione che rappresenta l'LFO.

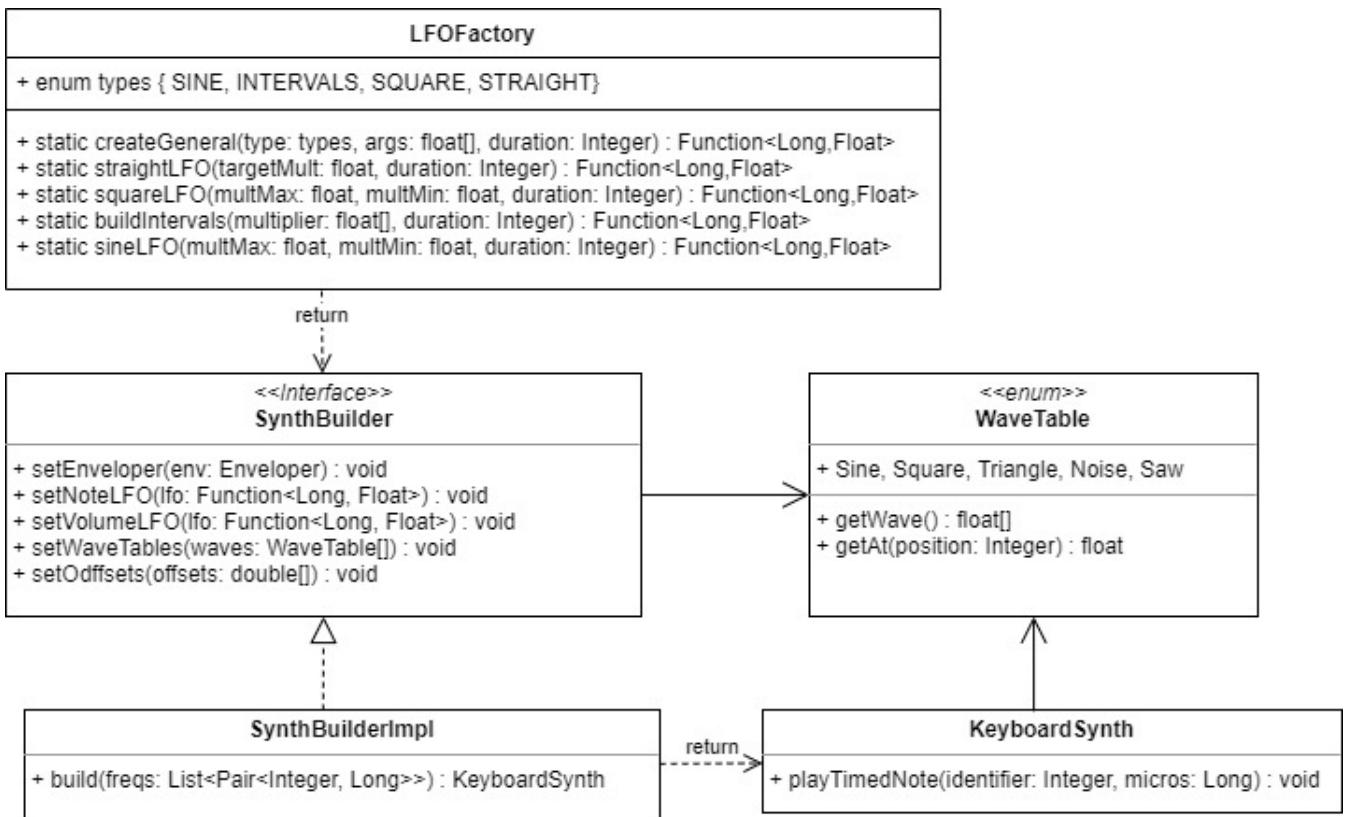


Figura 2.4: Schema UML dell'interazione tra LFOFactory, keyboardSynth e WaveTable.

2.2.2 Serafino Pandolfini

Il package logic si occupa di ricevere Song fornite dal MidiParser e di elaborare le tracce e le note che la compongono. Vi sono inoltre presenti metodi e classi per il calcolo del punteggio e per l'avvio della traccia audio. Il tutto si può dividere nelle seguenti sezioni:

Filtraggio delle note e difficoltà tracce

Il primo passaggio svolto è la riduzione del numero di note per ogni traccia. Prima vengono eliminate tracce non giocabili, generalmente tracce di percussioni identificate da un canale; poi da ogni traccia vengono eliminate le note troppo piccole per poter essere giocate con successo; infine, basandosi sul numero di note, vengono eliminate ulteriori note in modo da portare il numero di note di ogni traccia sotto una soglia predeterminata. Questo ultimo filtro è basato sull'ordine temporale delle note, in modo da preservare l'uniformità della traccia e garantire una certa diversità di numero di note tra le tracce di una Song. Si precisa che ogni volta che è necessario eliminare note dalla traccia invece che eliminarle si preferisce ricreare la traccia stessa tramite una factory in modo da utilizzare gli stream più agilmente. Dopo che ogni traccia è stata filtrata queste vengono mostrate a schermo con l'aggiunta di un livello di difficoltà per ognuna di esse. Questo livello è basato sul numero di note della traccia dopo essere stata filtrata. I vari livelli sono contenuti all'interno dell'enum DifficultyLevel.

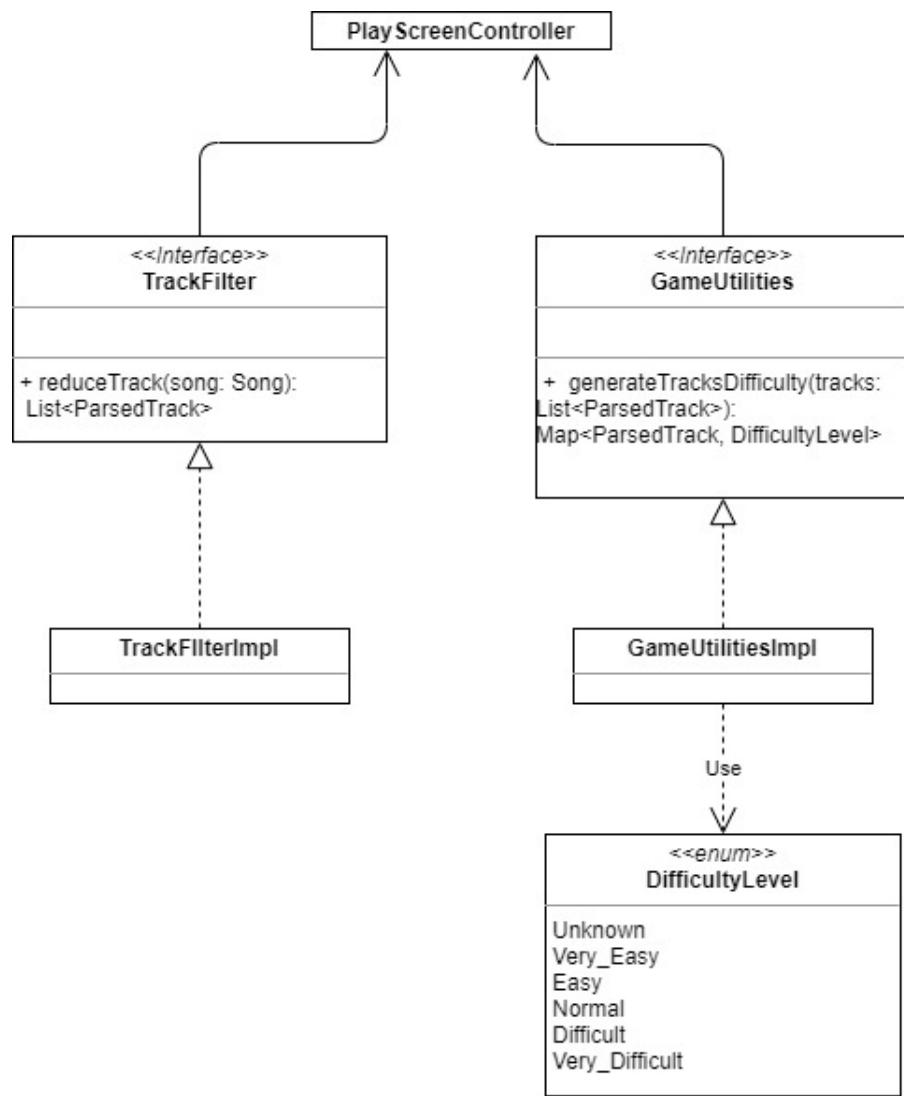


Figura 2.5: Schema UML del Track Filter.

Incolonramento delle note e passaggio alla grafica

Dopo che viene scelta una traccia da giocare è necessario posizionare le note nelle colonne presenti nell'enum Columns. In questo processo vengono anche generati i range che poi saranno sfruttati nel calcolo del punteggio. Dopo aver incolumnato le colonne queste vengono convertite tramite un pattern adapter. In questo caso il pattern, invece che connettere due interfacce, si occupa di rendere compatibili una classe astratta AbstractNote e un'interfaccia LogicNote. Tramite questo adattamento è possibile connettere gli aspetti logici e grafici del progetto.

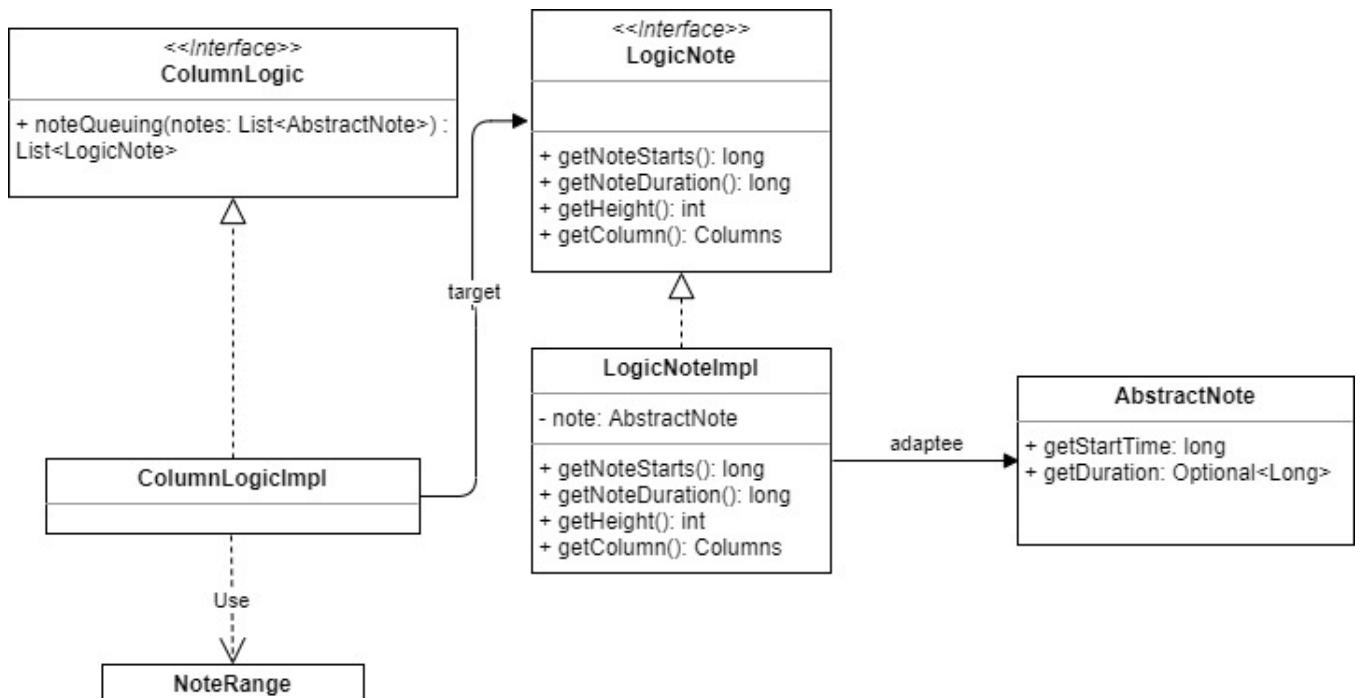


Figura 2.6: Schema UML dell'adapter.

Calcolo del punteggio

Per ogni nota premuta dal giocatore viene calcolato un punteggio. Viene prima deciso il range della nota premuta su quelli creati precedentemente, poi si passa al calcolo effettivo. L'algoritmo utilizzato per questo calcolo è strutturato usando il pattern strategy in modo da permettere in futuro l'aggiunta di ulteriori algoritmi per il calcolo del punteggio.

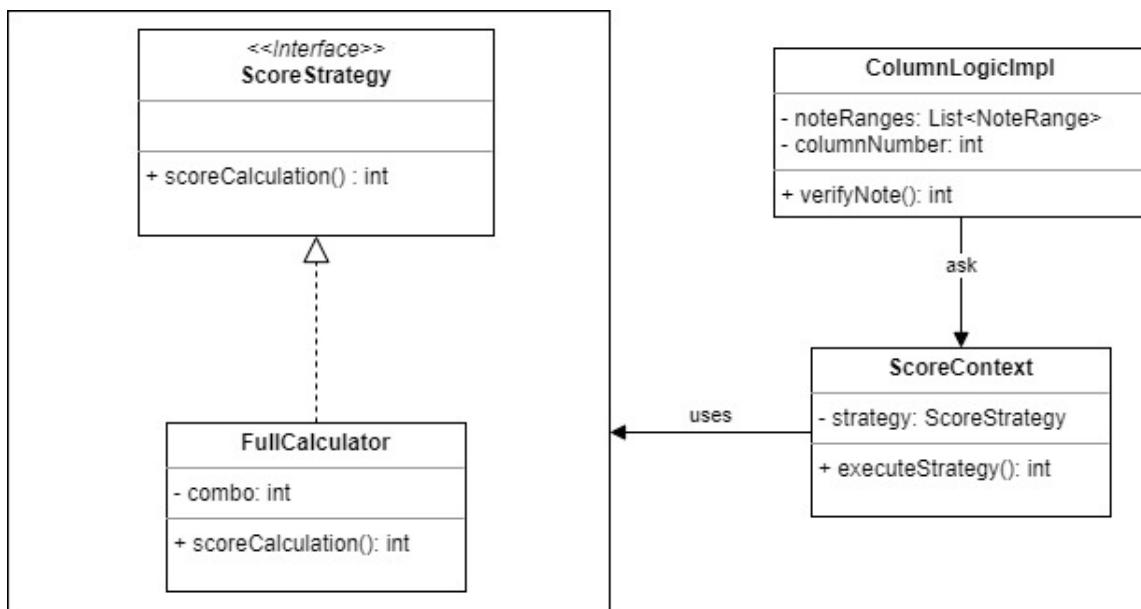


Figura 2.7: Schema UML della strategy per il calcolo del punteggio.

Observer - PlayerAudio

Per garantire una corretta gestione delle tracce ho deciso di implementare, nella classe PlayerAudio, il pattern observer. Ho scelto questo pattern in quanto, a causa del framerate variabile sui vari dispositivi, risulta comodo avere un singolo metodo per notificare tutte le tracce (gli Observer) di suonare le note il cui tempo di inizio è inferiore o uguale a quello scandito dalla chiamata di Player, l'Observable. La lista di observer "playableTracks" viene riempita dal costruttore una volta passata la canzone. Gli observer vengono caricati come istanze anonime di PlayableTrack, interfaccia contenente il metodo per essere notificata dall'Observable e per controllare se un'altra nota puo' essere suonata. Ad ogni chiamata del metodo playNotes, verranno rimosse le tracce non più utilizzabili e tutte quelle rimanenti verranno notificate di suonare, se possibile, delle note.

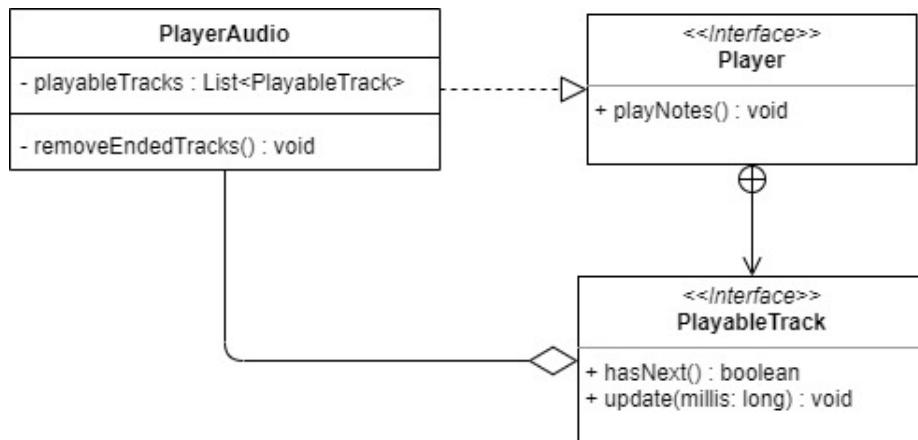


Figura 2.8: Schema UML per la gestione del player audio tramite observer.

2.2.3 Sofia Tosi

Questo paragrafo illustra il design relativo alla gestione dell'input e degli assets. In seguito ad un'attenta valutazione, si è deciso di far interfacciare l'utente all'applicazione DukeMania tramite l'utilizzo di una tastiera per pc. Durante la progettazione della sezione concernente l'input, ho ritenuto opportuno utilizzare il pattern Adapter. La classe EventsFromKeyboardImpl rappresenta l'Adapter in sè che, implementando l'interfaccia omonima, riesce a semplificare e a permettere l'interazione tra la tastiera, che sfrutta la libreria com.badlogic.gdx.Input, e la classe relativa alla grafica PlayScreen. Per non superare eccessivamente l'ammontare di ore, l'applicazione attuale è stata implementata per consentire solo l'utilizzo della tastiera del pc. Laddove si volesse, in futuro, cambiare la periferica, l'applicazione non esclude questa possibilità. Si potrebbe infatti implementare EventsFromKeyboard e sostituire la classe dell'adapter con un altro adapter che interagisca con una diversa periferica (ad esempio un gamepad, una SG Guitar Hero o una tastiera midi).

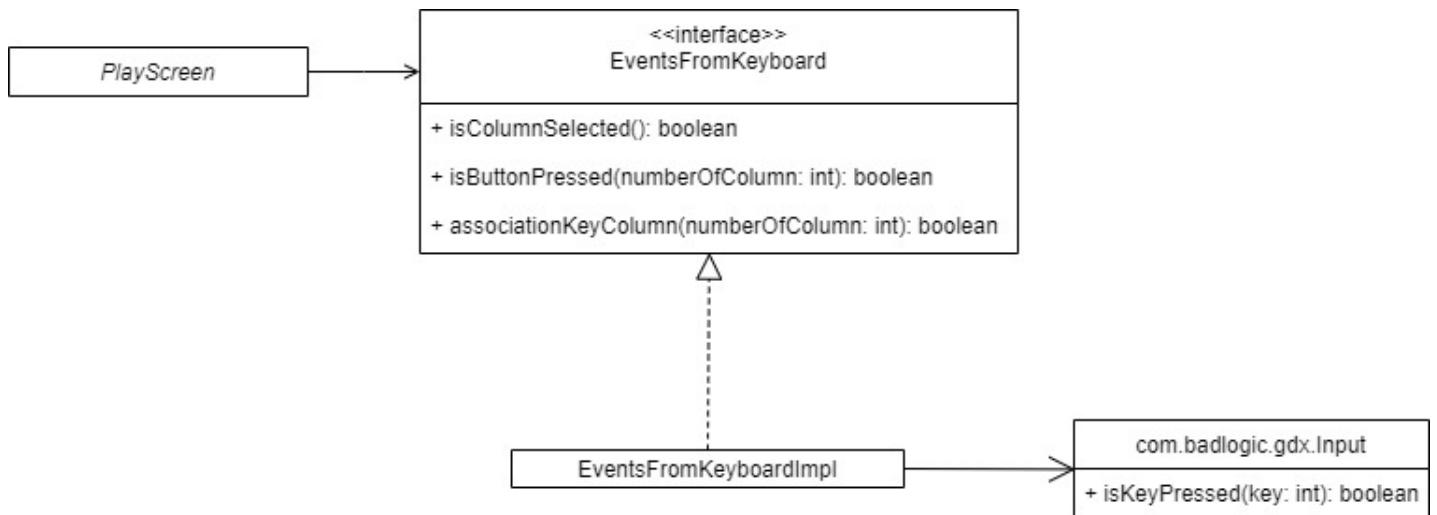


Figura 2.9: Schema UML dell'Adapter.

Un'altra peculiare classe che merita di essere citata è l'AssetsManager che, come dice il nome, si dedica all'organizzazione degli assets, ovvero tutte le risorse grafiche necessarie per il corretto funzionamento dell'applicazione. Queste risorse, che possono essere, ad esempio, le texture degli elementi nella GUI o i file con i font utilizzati, vengono caricate una sola volta in memoria. Ho deciso di sfruttare il pattern Singleton per la realizzazione di tale classe. Difatti è importante che ci sia una sola istanza dell'AssetsManager per evitare che le risorse grafiche vengano istanziate più volte in memoria sprecando spazio inutilmente.

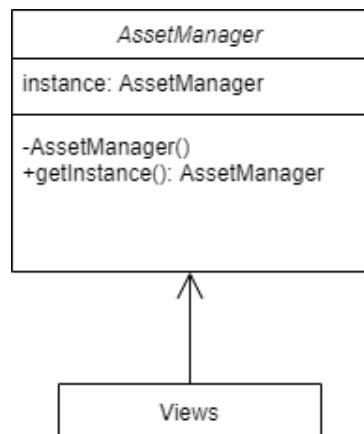


Figura 2.10: Schema UML del Singleton.

2.2.4 Laura Leonardi

Per quanto riguarda il parsing dei file midi è stata realizzata la classe MidiParser che ha lo scopo di ottenere, a partire da un file .mid, un oggetto di classe Song contenente una serie di informazioni utili per sintesi audio e la generazione delle tracce di gioco. Dalla classe Song saranno infatti ottenibili delle tracce (ParsedTrack), una per strumento più quella dedicata alle percussioni, da cui saranno a loro volta ottenibili le AbstractNotes, ovvero le note che le compongono. La classe MidiParser è stata dotata di un'interfaccia Parser per permettere un futura implementazione di un nuovo parser a partire da un tipo di file differente dal midi.

Abstract Factory

Le tracce e le note possono essere di due tipologie: normali e percussioni. Per semplificare la costruzione di questi diversi tipi di tracce e note ho pensato di utilizzare il pattern abstract factory, che permette la creazione di famiglie di oggetti della stessa tipologia. Sebbene infatti i campi e i metodi richiamabili dai differenti tipi di tracce e note coincidano solo in parte, ho ritenuto utile l'implementazione di questo pattern per separare la costruzione di queste tipologie di oggetti dal resto del codice, facilitandone così la lettura e l'utilizzo e portando la complessità di questa distinzione fuori dal MidiParser. Nello specifico questo pattern è stato realizzato tramite l'interfaccia AbstractFactory, che viene implementata dalle Concrete Factory PercussionFactoryImpl e FactoryImpl, che a loro volta si occuperanno di creare prodotti, quali PercussionNote o Note (che estendono la classe astratta AbstractNote, contenente i metodi in comune), e PercussionTrack o KeyboardTrack(che estendono la classe astratta ParsedTrack, contenente i metodi in comune). La scelta di utilizzare classi astratte invece di interfacce è stata effettuata per ottimizzare quanto più possibile il riuso del codice, in particolare le parti riguardanti i campi privati in comune, mantenendo comunque il vincolo della non istanziabilità dei prodotti dell'abstract factory. Inoltre nonstante la classe PercussionTrack non aggiunga nulla alla classe astratta ParsedTrack si è compiuta questa scelta implementativa per favorire una futura estensione della classe PercussionTrack o una proliferazione di altri tipi di track, cosa che sarebbe resa più difficile dalla sostituzione dell'abstract class ParsedTrack con la classe PercussionTrack. Per quanto riguarda la selezione della factory corretta è stata implementata un'utility class FactoryConfigurator che ritorna la concrete factory corretta a partire dal canale midi.

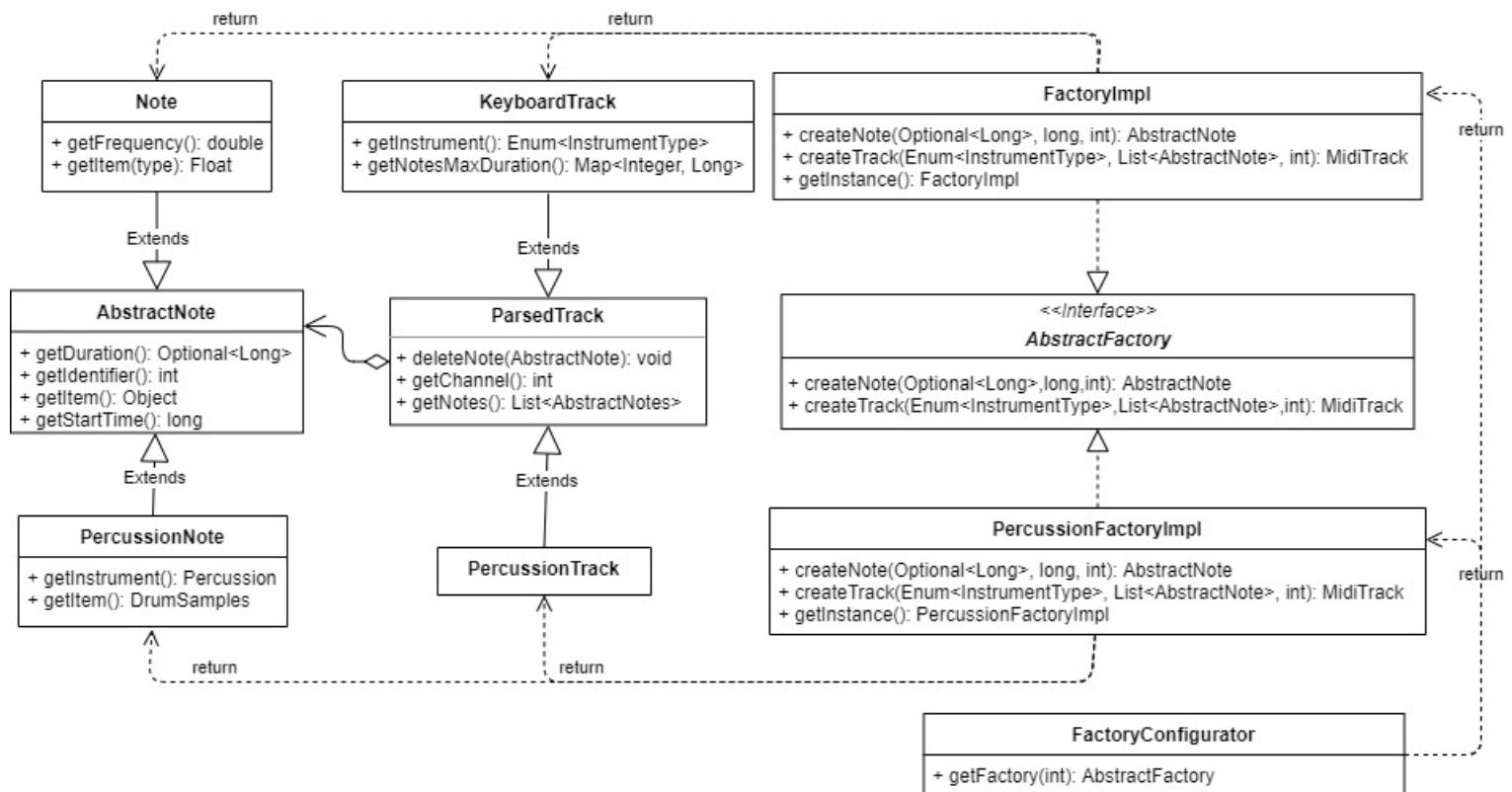


Figura 2.11: Schema UML della Factory.

Singleton

Nell'implementare le concrete factory ho ritenuto opportuno che non si generassero, per ogni creazione di nota o traccia, nuove istanze di factory, di conseguenza ho applicato il pattern Singleton alle classi `PercussionFactoryImpl` e `FactoryImpl`. Il pattern singleton è stato utilizzato anche nella classe `MidiParser` poiché è sufficiente una sola sua istanza per il corretto funzionamento del programma.



Figura 2.12: Schema UML del singleton MidiParser.

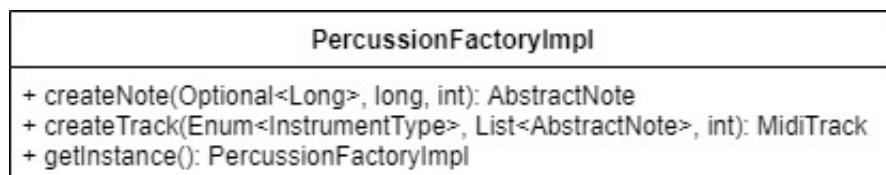


Figura 2.13: Schema UML del singleton PercussionFactoryImpl.

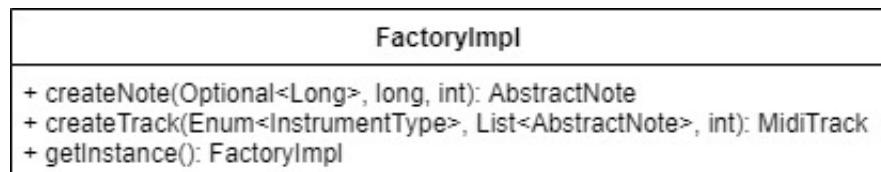


Figura 2.14: Schema UML del singleton FactoryImpl.

2.2.5 Alessandro Brasini

In questa sezione di design si parlerà del sistema di storage, window-switching, serializzazione e deserializzazione di oggetti (configurazioni di canzoni, preset di sintetizzatori, leaderboards).

Storage

All'interno del progetto bisogna inoltre accedere a diverse directory "fisse", tra le quali la cartella di configurazione presente nella home directory dell'utente `.dukemania`, degli asset oppure lavorare semplicemente su singoli file. L'accesso a tali cartelle deve essere pratico e senza dover immettere ogni volta il percorso preciso del file/directory a cui si vuole accedere, bensì vi si vuole accedere introducendo solamente il percorso parziale a partire dalla directory richiesta.

Ad esempio, se si volesse accedere ad un file di configurazione all'interno della cartella delle configurazioni del gioco (come ad esempio `/home/user/.dukemania/configs/synthesizers_config.json`), deve essere sufficiente inserire nel metodo per ottenere il file la stringa `configs/synthesizers_config.json`. A tal proposito ho ritenuto opportuno implementare il pattern **Strategy**. In questo caso l'interfaccia rappresentante la Strategy è l'interfaccia funzionale `Function<String, FileHandle>`, la quale esegue il mapping della path troncata (nell'esempio precedente `configs/synthesizers_config.json`) nella path completa (sempre dall'esempio precedente `/home/user/.dukemania/configs/synthesizers_config.json`) rappresentata dall'oggetto `FileHandle` di LibGDX per questioni di praticità.

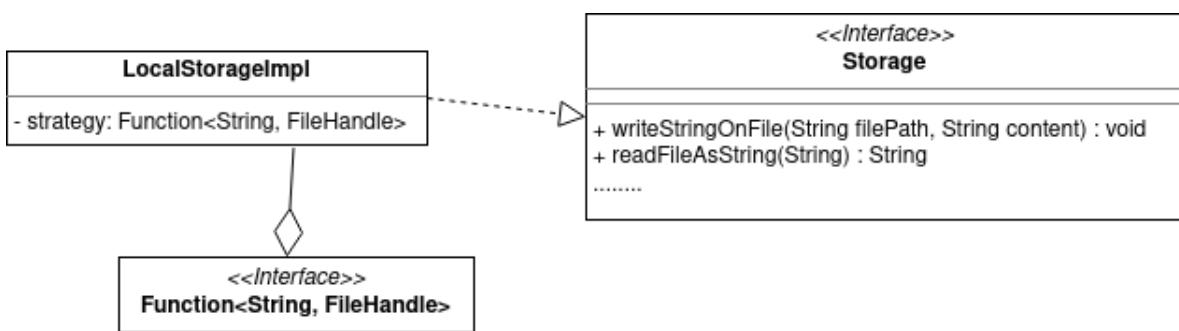


Figura 2.15: Rappresentazione UML dell'applicazione del pattern Strategy per il local storage

E' stato implementato anche il pattern **Factory** che, al momento, permette di ottenere oggetti implementati l'interfaccia **Storage** che lavorano sulla cartella degli asset, delle configurazioni e su un file generico presente nel file-system. Al momento il ruolo di questa factory è solamente quella di passare la corretta Strategy alla classe statica innestata **LocalStorageImpl** ma, in futuro, questa factory potrà implementare anche metodi per la creazione di oggetti che permettono di accedere a dello storage presente, ad esempio, su un server (un possibile metodo potrà essere `getServerStorage()`), rendendo di conseguenza poco efficace la creazione di una specifica Strategy, in quanto il codice per l'accesso ai dati dovrà essere del tutto riadattato.

Come possibile addizione futura a questa Factory è quella dell' aggiunta di un metodo per creare Storage che permettono solo la lettura e scrittura senza la possibilità di creazione di cartelle e/o file (o comunque di implementare un comportamento diverso). Di questa aggiunta ne andrebbe a beneficiare la classe di lettura/scrittura delle configurazioni (citata nella sottosezione successiva), in quanto non ha necessariamente bisogno di creare file e/o cartelle.

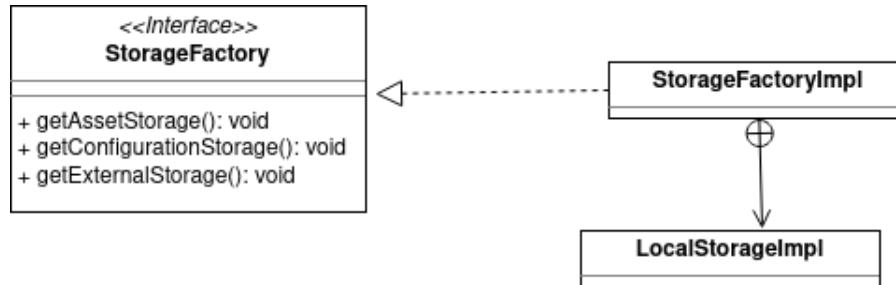


Figura 2.16: Rappresentazione UML dell'applicazione del pattern Factory per la creazione degli oggetti storage

Serializzatori e Deserializzatori

Per leggere e scrivere le varie configurazioni delle canzoni, sintetizzatori e leaderboards, in formato JSON, sono stati implementati serializzatori e relativi deserializzatori. La lettura e scrittura di questi oggetti serializzati è demandata alla classe **ConfigurationsModelImpl**, mentre la serializzazione dei singoli è demandata alle apposite classi presenti nel package model.serializers. Queste ultime sono state implementate sfruttando la libreria Jackson e, per ogni classe da serializzare/deserializzare, è stata aggiunta un'annotazione che indica quale classe usare per deserializzare la stessa. Ritornando alla classe ConfigurationsModelImpl, ho sempre ritenuto opportuno implementare il pattern **Strategy**, dove l'interfaccia rappresentante la Strategy, in questo caso, è **Storage**. In questo modo è possibile scegliere se dover leggere e/o scrivere in locale le configurazioni oppure, nel caso in futuro si voglia leggere/creare configurazioni di canzoni preesistenti e accedere ad una leaderbord globale (quindi entrambe presenti su un server), sarà necessario passare la Strategy opportuna creata dalla Factory citata nella sottosezione precedente come, ad esempio, getServerStorage().

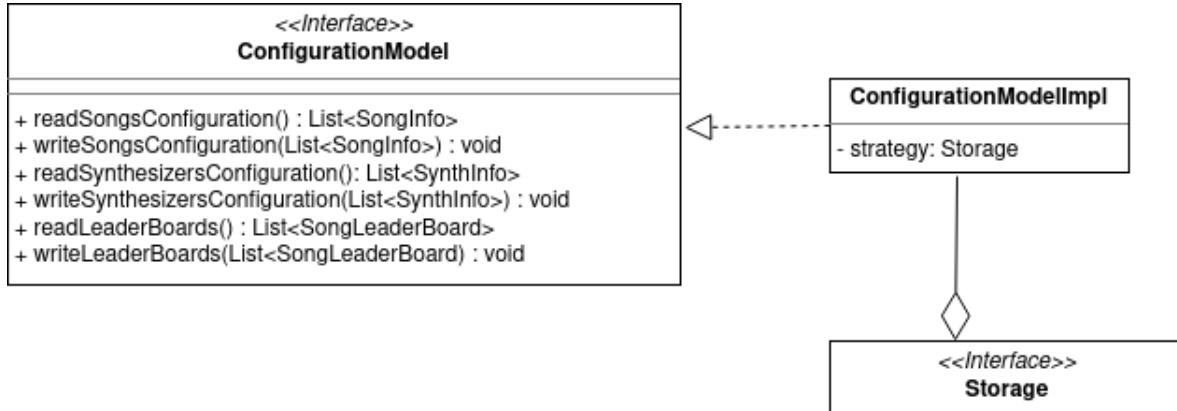


Figura 2.17: Rappresentazione UML dell'applicazione del pattern Strategy per il ConfigurationsModel

Menu

Per quanto riguarda le finestre del menu (e della leaderboard) è stato implementato per ognuno il pattern MVC, dove la View richiama i metodi del Controller per aggiornare a schermo i dati da visualizzare e il Controller richiama gli eventuali Model per leggere e/o modificare i dati.

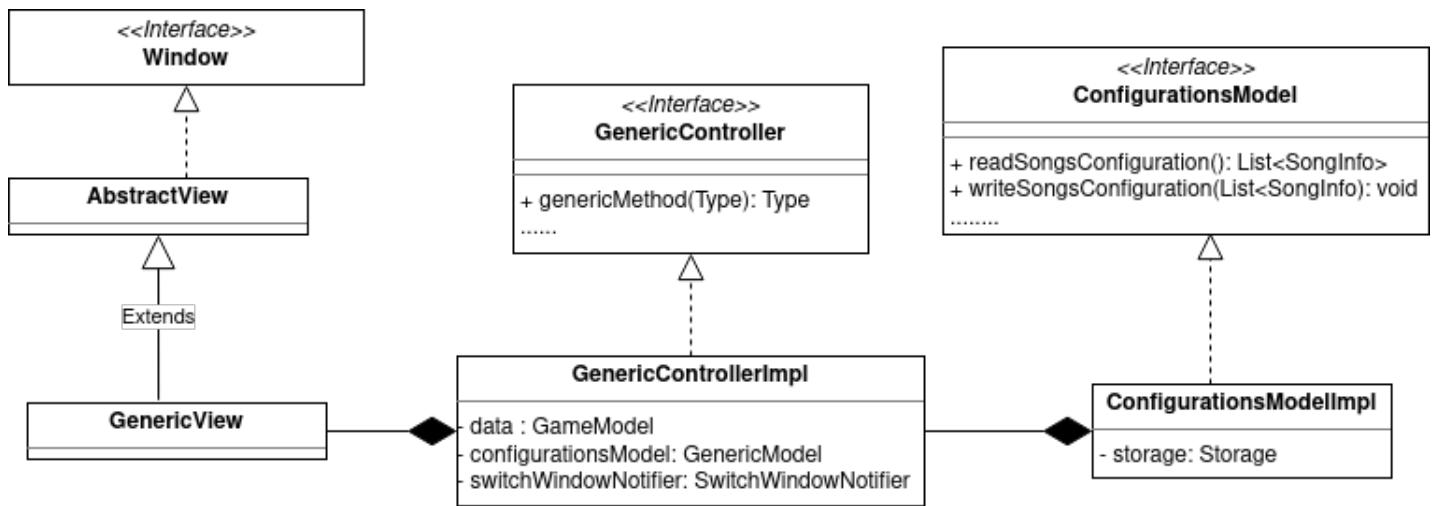


Figura 2.18: Rappresentazione UML dell'applicazione del pattern MVC di una generica schermata di menù

Tutte le view implementano l'interfaccia **Window** che definisce i metodi necessari per permettere la creazione e renderizzazione della stessa. Questi metodi saranno poi richiamati appropriatamente dal WindowManager. In fase di scrittura del codice ho notato che molte righe di codice si ripetevano in tutte le view del menu, per questo motivo ho deciso di implementare il pattern **Template Method**, dove il metodo template è rappresentato dalla classe astratta AbstractMenuView. Le classi che estendono quest'ultima andranno ad eseguire l'override dei soli metodi che cambiano, nel caso delle view del menu viene eseguito in quasi tutte solamente l'override del metodo pubblico **create()**, richiamando ovviamente prima la relativa **super.create()** che contiene codice simile a tutte le view del menu.

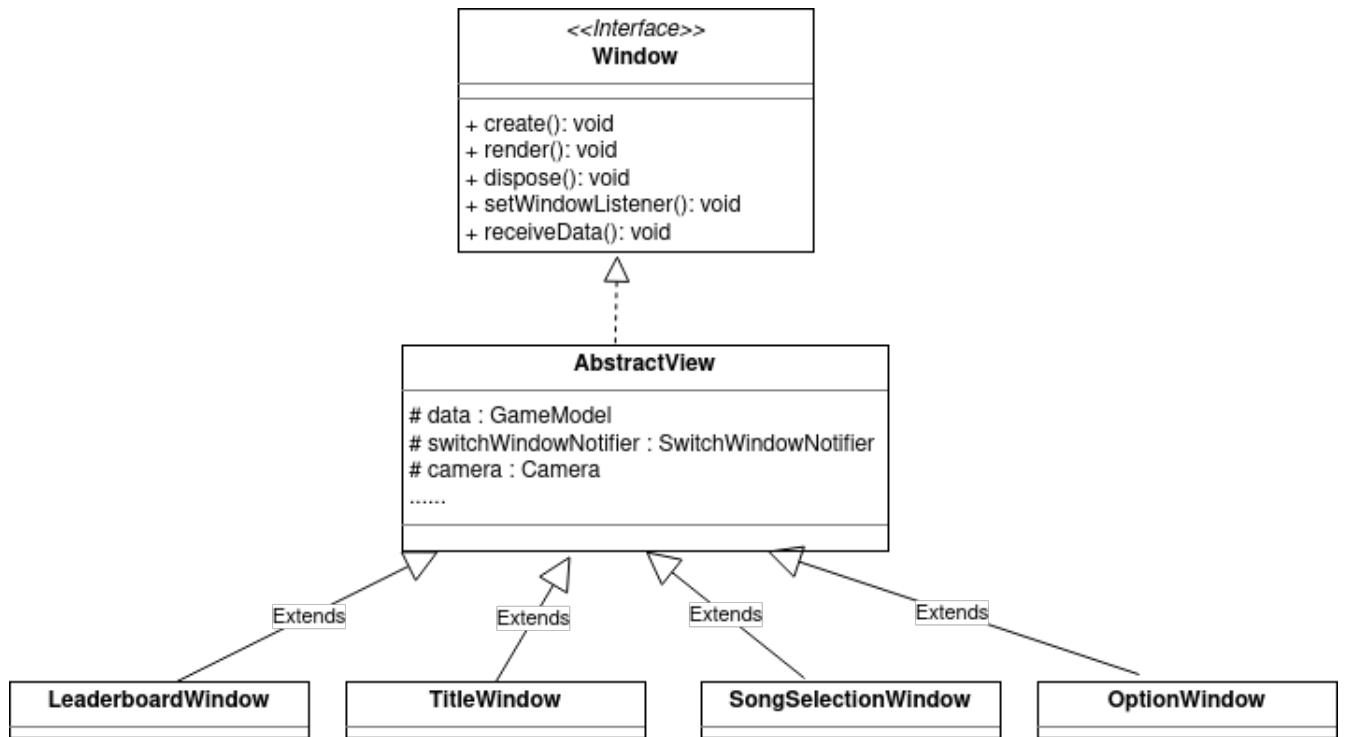


Figura 2.19: Rappresentazione UML dell'applicazione del Template method per la creazione di menù views

Window Switching

L’architettura generale del WindowManager è già stata spiegata in generale nella parte di design dell’architettura nel paragrafo precedente. In questa sezione si vuole parlare in modo più dettagliato del suo funzionamento. All’interno della classe entry point **DukeMania** viene istanziato l’oggetto WindowManager al quale vengono associate le varie view del gioco, tutte implementanti l’interfaccia **Window**, al relativo enum state (tutti implementati l’interfaccia `WindowState`) che associa lo stato della view. Ad esempio per la view *TitleWindow* è associato lo stato *DukeManiaWindowState.TITLE*. WindowManager, inoltre implementa l’interfaccia **SwitchWindowNotifier** che contiene come unico contratto il metodo **switchWindow(WindowState, GameModel)**, che prende in input come argomenti un oggetto implementante `WindowState`, il quale sostanzialmente dice di implementare un metodo che ritorna un intero associato allo stato, e l’oggetto `GameModel` che rappresenta il modello che ci si passa tra le varie view, che a loro volta passeranno al controller. Il metodo `switchWindow` non fa altro che notificare il WindowManager, mediante un meccanismo di callback, che un controller/view vuole cedere il controllo alla view specificata.

Capitolo 3

Sviluppo

3.1 Testing

Abbiamo deciso di eseguire test automatizzati prevalentemente per le classi più complesse. Per quanto riguarda le varie schermate e la gestione di queste, una volta prese in considerazione le limitazioni causate da libGdx, abbiamo preferito eseguire accurati test manuali.

Per il testing del file midi sono stati utilizzati anche dei test manuali, specialmente nella fase iniziale, con lo scopo di comprendere appieno e con quanta più precisione possibile la struttura del file midi

- **testStraightLFO - testSquareLFO - testSineLFO - testIntervalLFO:** per ogni funzione LFO è presente un test per garantirne il corretto funzionamento.
- **TestKeyboardSynth:** testato la correttezza del conto di quante note stanno suonando al momento, il metodo per ottenere il campione audio, il metodo per risuonare una nota.
- **DrumSynthTest:** testato la correttezza del conto di quante percussioni stanno suonando al momento, il metodo per ottenere il campione audio, il metodo per risuonare una percussione.
- **TestBuilder:** testate le eccezioni e il corretto funzionamento di metodo per ritornare l'istanza di KeyboardSynth.
- **TrackFilter:** testate singole tracce con numeri diversi di note e liste di tracce richiedenti filtri differenti.
- **GameUtilities:** testata l'attribuzione delle difficoltà alle tracce.
- **ColumnLogic - noteQueuing:** testate diverse configurazioni di numero di note, sia sovrapposte che non, e colonne.
- **ColumnLogic - verifyNote:** testato il calcolo del punteggio in relazione a precisione della nota e combo di note.
- **testMidiParser:** testata la correttezza della song restituita e degli elementi che la compongono.

3.2 Metodologia di lavoro

3.2.1 Gestione del lavoro

Lo sviluppo del progetto è stato preceduto da una fase preliminare di analisi, durante la quale tutti insieme abbiamo lavorato per definire le basi dell'applicazione e, successivamente, per specificare i vari dettagli implementativi tramite UML. La suddivisione delle parti del progetto è avvenuta in modo equo assegnando a ciascuno una parte significativa e importante del progetto.

Gestione del DVCS

Per il DVCS è stato utilizzato git, in quanto è stato il Software spiegato a lezione. Abbiamo creato una repository su github e abbiamo deciso di creare un branch per ogni membro del team. In seguito, quando ritenuto che ognuno di noi avesse implementato il "core" della propria parte, abbiamo deciso di eseguire un merge in un nuovo branch "alpha", sul quale ogni membro ha continuato a salvare le proprie modifiche e sul quale sono stati effettuati i primi test di giocabilità.

Suddivisione del lavoro

3.2.2 Gianluca Migliarini

In questo progetto, mi sono dedicato principalmente della riproduzione audio. Gli oggetti principali del mio package sono i Synth, i quali sono in grado, alla chiamata di un metodo, di restituire un campione audio ottenuto dalla somma di tutte le note o percussioni che stanno suonando simultaneamente. La classe Engine, si occupa di ottenere i campioni dai vari sintetizzatori, comporre il buffer audio, regolarne il volume e riprodurlo in uscita. Mi sono inoltre dedicato allo sviluppo di delle classi LFOFactory ed Enveloper, per, rispettivamente aggiungere varietà ai timbri sonori e per regolare il volume delle singole note in entrata e in uscita al fine di eliminare i "click" audio, sgradevoli all'udito. Infine, per semplificare la gestione delle forme audio da caricare nelle note, ho sviluppato l'enum read-only WaveTable.

Classi sviluppate singolarmente:

- DrumSynth
- Engine
- Enveloper
- Filters
- KeyboardSynth
- LFOFactory
- Settings
- SynthBuilder

Inoltre, a mio parere, anche gli enum WaveTable DrumSamples devono essere listati, in quanto contenti metodi e campi oltre alla semplice enumerazione.

3.2.3 Serafino Pandolfini

Per questo progetto mi sono focalizzato principalmente sugli aspetti di logica del gioco, comprendenti la gestione di tracce e note e della precisione, sia nell'aspetto audio-video che nel calcolo dei punteggi di gioco. Per quanto concerne l'aspetto delle tracce la classe fondamentale da me implementata è TrackFilterImpl che, fornita in input una Song, ottenuta dal midiParser, riduce il numero di note presenti in ogni sua traccia in modo da renderle giocabili. Ho provato diverse strategie per ridurre il numero delle note in modo da ottenere tracce con una distribuzione di note uniforme nel tempo e con quantitativi di note quanto più possibile differente per fornire più scelta al giocatore. Questa classe è stata particolarmente facile da integrare in quanto si occupa di elaborare dati già esistenti e a restituirli come era stato previsto in fase di progettazione. Per le note e il calcolo del punteggio ho realizzato un'unica classe principale ColumnLogicImpl che divide le note nelle varie colonne grafiche eliminando le eventuali collisioni e, utilizzando la classe secondaria NoteRange, permette di fornire un punteggio per ogni nota premuta durante il gioco sulla base della configurazione del calcolo scelta tra le classi di calcolo del punteggio. Anche in questo caso l'obiettivo è stato quello di ottenere una distribuzione uniforme tra le varie colonne delle note e di fornire un'attribuzione del punteggio che considerasse eventuali imprecisioni sulla pressione delle note durante il gioco. Questa classe è stata particolarmente difficile da finalizzare per una divergenza di classi/interfacce tra la parte logica e grafica, ed è stato quindi necessario creare una classe intermediaria LogicNoteImpl e dei metodi aggiuntivi in grado di fornire i dati mancanti per permettere un'integrazione tra le due parti. In merito agli aspetti di precisione audio-video ho sviluppato la classe PlayerAudio che suona le note in base al tempo trascorso dall'inizio della traccia, ottimizzando la latenza tra ciò che appare a schermo e l'audio di gioco creando una relazione tra l'aspetto grafico e quello audio. L'integrazione di questa classe non ha generato particolari problemi se non la necessità di avere una conoscenza superficiale del funzionamento delle altre classi presenti nell'audioengine. L'ultima parte sviluppata singolarmente è quella relativa alla classe GameUtilitiesImpl, utilizzata per attribuire a ogni traccia restituita dal trackFilter una difficoltà basata sul numero di note.

Classi sviluppate singolarmente:

- ColumnLogicImpl
- NoteRange
- LogicNoteImpl
- ScoreContext
- FullCalculator
- TrackFilterImpl
- GameUtilitiesImpl
- PlayerAudio
- TestLogic

3.2.4 Sofia Tosi

Per questo progetto mi sono focalizzato principalmente sugli aspetti di logica del gioco, comprendenti la gestione di tracce e note e della precisione, sia nell'aspetto audio-video che nel calcolo dei punteggi di gioco. Per quanto concerne l'aspetto delle tracce la classe fondamentale da me implementata è TrackFilterImpl che, fornita in input una Song, ottenuta dal midiParser, riduce il numero di note presenti in ogni sua traccia in modo da renderle giocabili. Ho provato diverse strategie per ridurre il numero delle note in modo da ottenere tracce con una distribuzione di note uniforme nel tempo e con quantitativi di note quanto più possibile differente per fornire più scelta al giocatore. Questa classe è stata particolarmente facile da integrare in quanto si occupa di elaborare dati già esistenti e a restituirli come era stato previsto in fase di progettazione. Per le note e il calcolo del punteggio ho realizzato un'unica classe principale ColumnLogicImpl che divide le note nelle varie colonne, organizzate come un enum, grafiche eliminando le eventuali collisioni e, utilizzando la classe secondaria NoteRange, permette di fornire un dato per confrontare le note premute dal giocatore e assegnarle un punteggio. Questa sezione è stata particolarmente difficile da costruire per una divergenza di classi/interfacce tra la parte logica e grafica, ed è stato quindi necessario creare una classe intermediaria LogicNoteImpl e dei metodi aggiuntivi in grado di fornire i dati mancanti per permettere un'integrazione tra le due parti. In merito agli aspetti di precisione audio-video ho sviluppato la classe PlayerAudio che suona le note in base al tempo trascorso dall'inizio della traccia, ottimizzando la latenza tra ciò che appare a schermo e l'audio di gioco creando una relazione tra l'aspetto grafico e quello audio. L'integrazione di questa classe non ha generato particolari problemi se non la necessità di avere una conoscenza superficiale del funzionamento delle altre classi presenti nell'audioengine. L'ultima parte sviluppata singolarmente è quella relativa alla classe GameUtilitiesImpl, utilizzata per attribuire a ogni traccia restituita dal trackFilter una difficoltà, ottenuta da una enum, basata sul numero di note.

Classi sviluppate singolarmente:

- PlayScreen
- AssetsManager
- EventsFromKeyboardImpl
- GraphicNoteImpl
- SizeImpl
- KeyImpl
- ComputingShiftImpl

3.2.5 Laura Leonardi

In questo progetto mi sono dedicata al midi parsing, ovvero la restituzione, a partire da un file midi, di una classe Song contenente informazioni generali sulla canzone e composta da tracce. Queste ultime sono identificate da un numero di canale e dallo strumento utilizzato, e sono a loro volta composte da note, dotate di attributi quali durata, tempo di inizio e identificatore. Vi sono inoltre un tipo particolare di tracce e corrispettive note, ovvero tracce e note di tipo percussione. In questo caso la traccia non sarà associata ad uno strumento, bensì saranno le note ad essere identificate, invece che da una frequenza, da un tipo particolare di percussione. Per quanto riguarda il collegamento tra gli elementi dell'enum Percussion a DrumSamples e l'associazione di Instrument al rispettivo SynthBuilderImpl ho agito in collaborazione con gli autori delle classi non di mia pertinenza, specialmente per comprendere meglio il modo con cui interfacciarmi ad esse. Dal mio punto di vista durante lo sviluppo non sono sorti grossi problemi, se non qualche aggiunta di metodo del tutto ordinaria.

Classi sviluppate singolarmente:

- AbstractNote
- FactoryConfigurator
- FactoryImpl
- Instrument
- InvalidNoteException
- MidiParser
- ParsedTrack
- Note
- PercussionFactoryImpl
- PercussionNote
- PercussionTrack
- Song
- TestMidiParser
- KeyboardTrack

3.2.6 Alessandro Brasini

All'interno del progetto mi sono occupato di sviluppare le view e relativi controller per le varie schermate del menù; realizzazione di serializzatori e deserializzatori custom per le configurazioni delle canzoni salvate, leaderboard e preset dei sintetizzatori; gestione dell'accesso ai file di configurazione; implementazione della core architecture. Il processo di integrazione della mia parte con le altre è stato relativamente semplice in quanto non necessitava di molte interazioni tra le varie classi sviluppate dai miei colleghi. Le parti in cui si necessitava di una integrazione, come quella per integrare la schermata di PlayScreen, è stata risolta facendo implementare a quest'ultima l'interfaccia Window, mentre per il passaggio dei dati tra le varie Window è stata implementata la classe GameModel, la quale racchiude tutti i dati attuali della partita (nome del giocatore, traccia e canzone selezionata, punteggio, hash della canzone e il numero di colonne sul quale vuole giocare). Package sviluppati:

- **it.dukemania.controller.filialog**
- **it.dukemania.controller.leaderboard**
- **it.dukemania.controller.option**
- **it.dukemania.controller.songselection**
- **it.dukemania.model**
- **it.dukemania.util.storage**
- **it.dukemania.view.menu**
- **it.dukemania.windowmanager**

Classi sviluppate:

- **DukeMania.java**

3.3 Note di sviluppo

3.3.1 Gianluca Migliarini

Feature avanzate del linguaggio:

- **lambda:** usate per caricare gli iteratori delle note per ogni traccia e i buffer dei campioni audio nei sintetizzatori di tastiere.
- **stream:** usati per controllare lo stato di tutti i sintetizzatori e sommare i vari campioni di questi ultimi.
- **optional:** usati per i campi opzionali nel builder dei sintetizzatori di tastiera.
- **function:** usate per gestire gli LFO

Librerie utilizzate:

- **libgdx:** utilizzata per Gdx.audio.newAudioDevice, necessario per riprodurre sul dispositivo audio hardware il buffer di campioni.

Classi riutilizzate:

Per comodità ho deciso di riutilizzare la classe generica Pair, introdotta dal prof. Viroli durante il corso.

3.3.2 Serafino Pandolfini

Feature avanzate del linguaggio:

- **lambda:** utilizzate per il filtraggio delle tracce, l'incolumnamento delle note, l'individuazione del range della nota premuta, l'assegnazione delle difficoltà alle tracce, all'interno del note player.
- **stream:** utilizzate per il filtraggio delle tracce, l'incolumnamento delle note, l'individuazione del range della nota premuta, l'assegnazione delle difficoltà alle tracce, all'interno del note player.
- **optional:** utilizzate per il filtraggio delle tracce, l'incolumnamento delle note, l'individuazione del range della nota premuta, l'assegnazione delle difficoltà alle tracce, all'interno del note player, nell'adapter per il passaggio dati alla grafica.

3.3.3 Sofia Tosi

Feature avanzate del linguaggio:

- **lambda:** usate nella dispose dell'AssetsManager per rilasciare ogni singola risorsa utilizzata.
- **stream:** usato nel metodo associationKeyColumn per associare in maniera efficiente ogni colonna al relativo tasto della tastiera e nel PlayScreen per calcolarmi la lista con le note che stanno suonando al momento.
- **interfacce funzionali:** usate per eliminare le note che hanno terminato di suonare.
- **Optional:** usati nella classe GraphicNoteImpl.

librerie utilizzate:

- **libgdx:** libreria grafica Libgdx: studio autonomo di tale libreria per realizzare l'interfaccia di gioco.

Classi riutilizzate:

Utilizzo della classe Pair fornita gentilmente dal professore Viroli

3.3.4 Laura Leonardi

Feature avanzate del linguaggio:

- **lambda:** usate negli stream
- **stream:** utilizzati per l'aggiunta e la gestione delle note, per il riempimento di una mappa che associa note a durata massima, per l'associazione tra strumento e sintetizzatore e per l'associazione di percussioni ai relativi DrumSamples e per ottenere tracce correttamente nella classe di testing.
- **optional:** usati per l'inserimento di note "parziali" in attesa della loro conclusione dettata dal midi.

library utilizzate:

- **javax.sound.midi:** gestione file .mid

Classi riutilizzate:

Classe generica Pair, fornita dal prof. Viroli.

3.3.5 Alessandro Brasini

Aspetti avanzati del linguaggio:

- **Lambda**
- **Stream:** utilizzati ampiamente nelle implementazioni dei vari controller delle view
- **Interfaccia funzionale Function:** utilizzata come interfaccia strategy per la creazione dei vari tipi di storage.

Librerie utilizzate:

- **Jackson:** utilizzata per la serializzazione e deserializzazione delle leaderboard, preset dei sintetizzatori e configurazioni delle canzoni in formato JSON.
- **Libgdx:** libreria grafica utilizzata per la creazione dell'interfaccia grafica.

Capitolo 4

Commenti Finali

4.1 Gianluca Migliarini

4.1.1 commenti

Sono molto soddisfatto del gruppo e del lavoro che siamo riusciti a portare a termine. Essendo una delle mie prime esperienze di lavoro in team ho trovato delle difficoltà principalmente riguardanti la gestione del tempo e a volte la coordinazione con gli altri membri, per questo ringrazio i miei compagni per la disponibilità e per aver tenuto sempre un clima sereno durante lo sviluppo; ritengo quindi che questo progetto mi abbia permesso di crescere notevolmente dal punto di vista organizzativo, in particolare per l'utilizzo di un software per la gestione del versioning. Inoltre sono molto felice di essere riuscito ad inserire in un progetto concreto il mio interesse per la sintesi audio.

4.1.2 difficoltà riscontrate

In conclusione, ritengo che il corso sia stato più che sufficiente per garantirmi uno sviluppo del progetto senza problematiche troppo grandi a livello implementativo, e che mi abbia incentivato ad utilizzare gli aspetti più avanzati del linguaggio, i quali avrei probabilmente evitato.

4.2 Serafino Pandolfini

4.2.1 commenti

Sono personalmente soddisfatto del lavoro che ho svolto; fin dai miei primi approcci alla programmazione ho sempre preferito, piuttosto che dedicarmi a particolari funzionalità di librerie, a porre il mio interesse sulle strutture dati e le loro organizzazioni e in questo progetto ho avuto modo di mettermi alla prova in un contesto che non fosse il mio solito programma giocattolo scoprendo le mie effettive conoscenze e capacità. Ho avuto modo di apprezzare in particolare gli stream e le loro funzionalità che riuscivano sempre a stupirmi. È stata ricorrente la situazione in cui emergeva un problema ed esisteva già un'operazione in grado di risolverlo in modo quasi istantaneo. Non ho intenzione di continuare a sviluppare questo progetto ma se non mi fossi limitato al monte ore richiesto mi sarebbe piaciuto sviluppare altre modalità di gioco con calcoli dei punteggi e gameplay differenti prendendo ispirazione da altri giochi simili.

4.2.2 difficoltà riscontrate

Il progetto non è stato esente da difficoltà, principalmente la coordinazione tra i vari membri per quanto ottimale ha portato a incomprensioni sul funzionamento di certi aspetti e talvolta si è dovuto ricorrere a soluzioni non ottimali dal mio punto di vista. Dal mio punto di vista però l'emergere di queste difficoltà mi ha permesso di avere una comprensione più completa del progetto che non avrei avuto altrimenti.

4.3 Sofia Tosi

4.3.1 commenti

DukeMania è stato il primo grande progetto a cui ho preso parte. Non nego la mia preoccupazione iniziale, poi, pian piano durante l'analisi e implementando codice, è svanita. Questo progetto mi ha permesso di crescere ampiamente, in vari ambiti. In primo luogo, ho imparato cosa significa scrivere codice di qualità e quanto siano importanti i pattern. Al contempo, ho anche aumentato la mia conoscenza studiando una nuova libreria grafica e imparando meglio le potenzialità di Git e LaTex. Realizzare DukeMania non è stato affatto facile, soprattutto perché si tratta di un gioco musicale e le mie conoscenze in quell'ambito sono esigue. Ma grazie alla collaborazione con i miei compagni, tutti molto disponibili e gentili siamo riusciti nel nostro intento. Indipendentemente dalla valutazione finale, mi ritengo molto soddisfatta del progetto.

4.3.2 difficoltà riscontrate

Ritengo il corso di Programmazione ad Oggetti un corso molto valido e interessante. Mi ha permesso di approfondire nuovi argomenti non soltanto da autodidatta, ma anche con le lezioni dei prof che ho trovato chiare e stimolanti. L'unica modifica che apporterei è l'aggiunta di ore di laboratorio per imparare già da subito ad applicare correttamente i pattern e C#. Nella mia modesta opinione avere più ore da dedicare a questi argomenti comporterebbe un notevole aiuto.

4.4 Laura Leonardi

Personalmente ritengo che in questo progetto sia stata effettuata un'ottima collaborazione tra membri del team, ognuno di quali è sempre stato molto disponibile, inoltre sebbene non possedessi molta esperienza col DVCS sono riuscita comunque a farne uso e a comprenderne la comodità. Un altro lato decisamente positivo del progetto è stata la possibilità di approfondire nuovi argomenti, infatti ho trovato molto interessante lavorare e scoprire il funzionamento del protocollo MIDI, del quale prima di questo progetto, non essendo particolarmente appassionata di musica, non conoscevo neppure l'esistenza. Sono in definitiva soddisfatta dell'esito del progetto.

4.4.1 difficoltà riscontrate

Dal mio punto di vista il corso di oop si è rivelato impegnativo ma decisamente utile per quanto riguarda le pratiche di buona programmazione e l'utilizzo di DVCS, nonostante questo ritengo che purtroppo le ore di laboratorio dedicate all'effettiva programmazione siano state un poco ridotte, in particolare per quanto riguarda C#.

4.5 Alessandro Brasini

4.5.1 commenti

Tenendo in conto degli ostacoli incontrati nel corso dello sviluppo del progetto per arrivare a conoscere libgdx, mi sento molto soddisfatto del risultato finale ottenuto. Sono inoltre riconoscente ai miei compagni di team, i quali hanno mantenuto un approccio serio e professionale, nonostante la nostra comune inesperienza riguardante il lavoro in gruppo. Ho imparato l'importanza della fase di ideazione del progetto e l'utilità di suddividere equamente, tra compagni, il carico di lavoro, senza mai sottovalutare l'importanza di un DVCS, il quale si è rivelato fondamentale. In conclusione, la cosa che ho imparato di più da questo progetto, è il programmare in una modalità più responsabile e consapevole, in quanto i miei collaboratori riponevano fiducia nel mio lavoro come io nel loro.

4.5.2 difficoltà riscontrate

Durante il mio percorso formativo avevo già avuto occasione di programmare ad oggetti, ma è stato solamente dopo questo corso che ho capito la vera utilità di concetti come interfacce e classi astratte. Inoltre sono grato di aver utilizzato java, in quanto ho avuto la possibilità di imparare a programmare in un linguaggio del tutto nuovo, e per la prima volta ricco di sfaccettature e funzioni avanzate. Ritengo infine che le uniche problematiche inerenti al corso siano dovute dalla metodologia di lezione a distanza, problemi che purtroppo si sono riversati durante gli appelli d'esame e che mi hanno impedito di beneficiare degli insegnamenti al massimo.

Capitolo 5

Appendice - Guida Utente



Figura 5.1: La schermata iniziale del gioco.



Figura 5.2: La schermata di configurazione della partita.

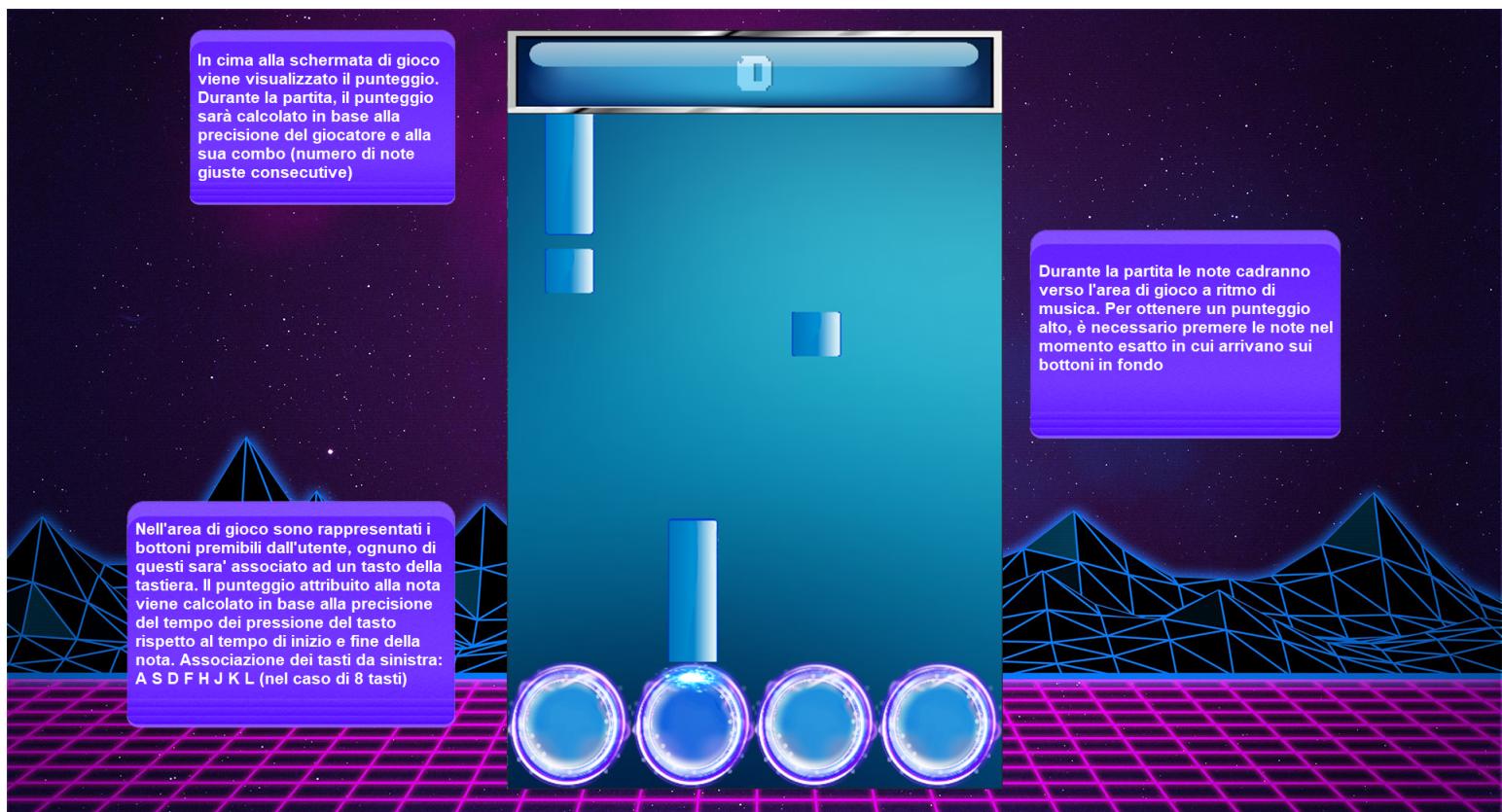


Figura 5.3: La schermata di gioco.

Known Issue: su Linux, a causa di un problema di resize di libGdx, la finestra di gioco potrebbe scomparire se trascinata con il mouse.