Report progetto “1010”

Ascani Tommaso
Borgiani Andrea

15 dicembre 2023

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il software che si vuole realizzare mira alla costruzione di un gioco arcade ispirato al medesimo "1010". Il gioco consiste nel posizionare nel modo più adeguatamente possibile dei pezzi, generati casualmente, su una griglia che al completare di una riga o colonna provvederà ad eliminarla ricompensando il player con una moneta. L'applicazione è costruita per non avere una fine, poiché, se il giocatore è abbastanza bravo, potrebbe andare avanti all'infinito.

Requisiti funzionali

Quando si formano una o più righe/colonne il gioco deve riconoscere che sono piene e deve eliminarle rendendo le caselle libere di essere riempite con ulteriori pezzi.

Durante una partita si deve poter mettere in pausa e decidere se riprendere la partita, ricominciarla, o tornare al menu principale, in quest'ultimo caso potendo scegliere se salvare la partita corrente oppure no.

Il gioco deve costantemente tenere conto dei pezzi correntemente posizionabili e dei posti liberi nella griglia per poter dire al giocatore se può continuare a giocare oppure se ha perso e di conseguenza visualizzare la schermata di Game Over.

L'applicazione deve poter salvare e caricare dati su file json, fondamentale per poter riprendere una partita salvata, modificare le impostazioni e il tema selezionato.

Possibilità di modifica delle impostazioni come volume e lingua.
Il gioco deve salvare il miglior punteggio realizzato su una specifica griglia e aggiornarlo in caso di miglioramento.

Requisiti non funzionali

Il gioco deve avere una interfaccia semplice, pulita e immediata per poter non indurre il giocatore a scelte errate.

1.2 Analisi e modello del dominio

Come si può vedere nella Figura 1.1 e Figura 1.2, possiamo notare che per implementare le view abbiamo adottato una interfaccia View e una sua implementazione astratta *ViewImpl* che potesse fare da base per tutte le view specifiche HomeView, GameView, SettingView e ShopView. La classe astratta *ViewImpl* quindi, oltre che contenere lo stage e le dimensioni di quest'ultima, dichiara un metodo astratto start(). Questo viene implementato in ogni view e viene richiamato dalla classe di utilità ViewSwitcher poco prima dello show(). In questo modo ogni view inizializzerà i propri campi e imposterà i propri componenti di JavaFX in modo appropriato.

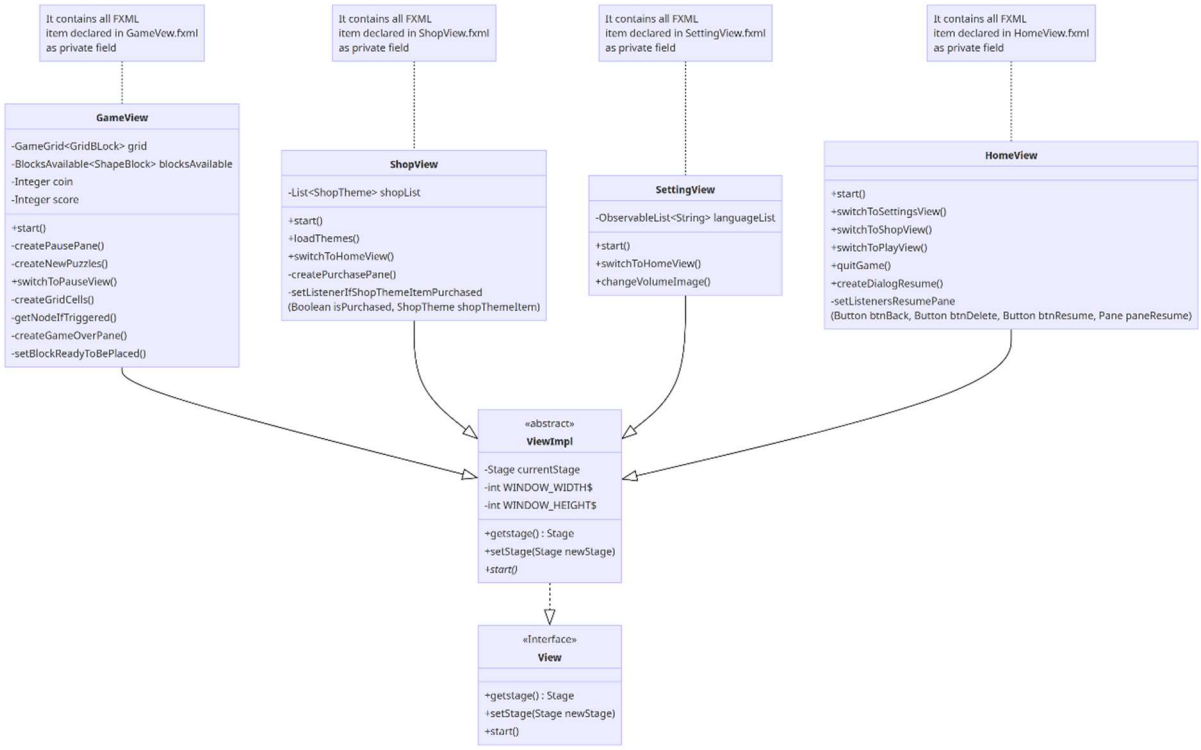


Figura 1.1: Schema UML delle classi che estendono View, con rappresentate le entità principali, i campi e le relazioni tra loro.

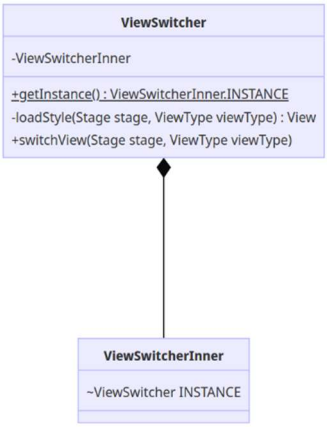


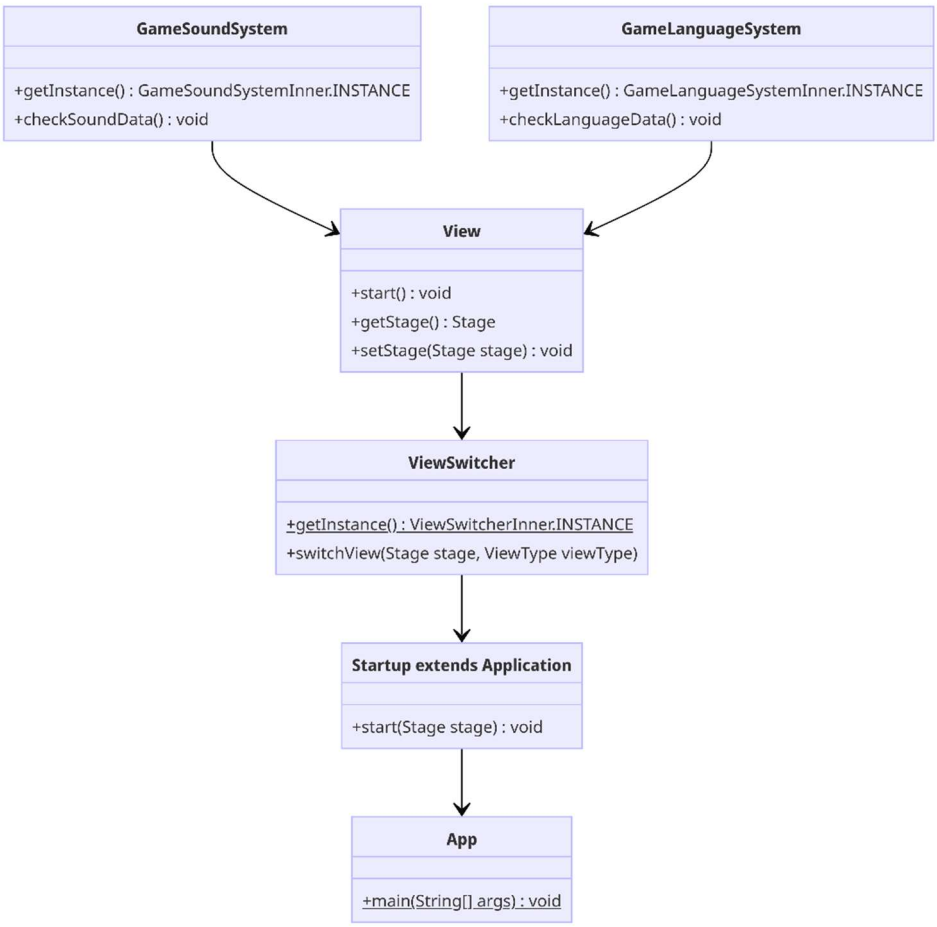
Figura 1.2: Schema UML della classe *ViewSwitcher*, con rappresentati i suoi campi e le sue funzioni.

Capitolo 2

Design

2.1 Architettura

Per questo progetto è stato utilizzato il classico pattern architetturale Model-View-Controller (MVC), che permette una buona organizzazione della struttura. Durante lo sviluppo abbiamo optato per unire la gestione delle view e il loro controllo all'interno della stessa classe, poiché ogni view all'interno del gioco ha una logica a sé stante e non esiste un vero e proprio loop di gioco, così ci sembrava superfluo creare un controller per ogni view che gestisse solo quella specifica view. Al caricamento della view vengono predisposti gli elementi grafici necessari al corretto funzionamento, dopo di che ad ogni azione dell'utente la view è in grado di svolgere in ruolo di controller di sé stessa. Tramite l'utilizzo di altre classi gestisce gli eventuali model di cui necessita.



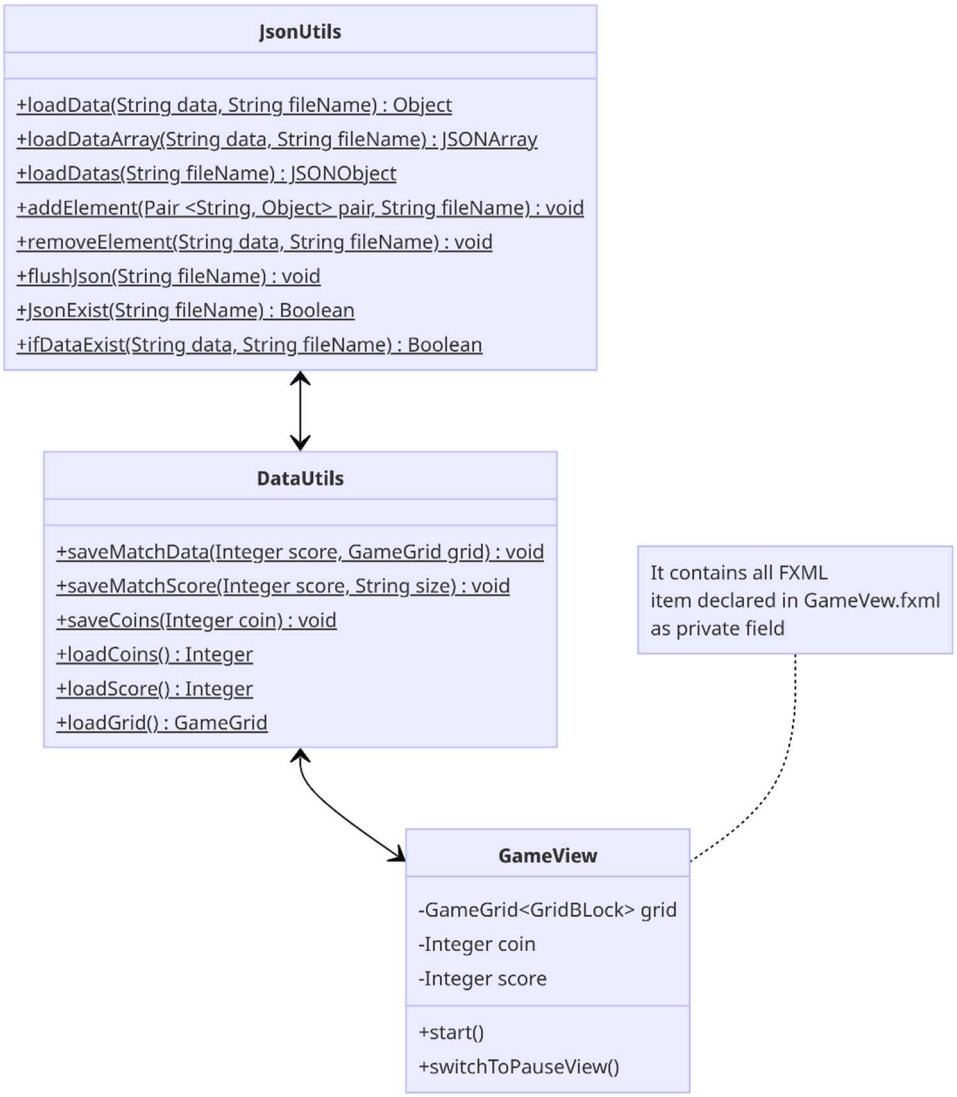
2.2 Design dettagliato

Ascani Tommaso

- ### Gestione dei salvataggi

  Problema: Salvare i dati di una partita in corso.

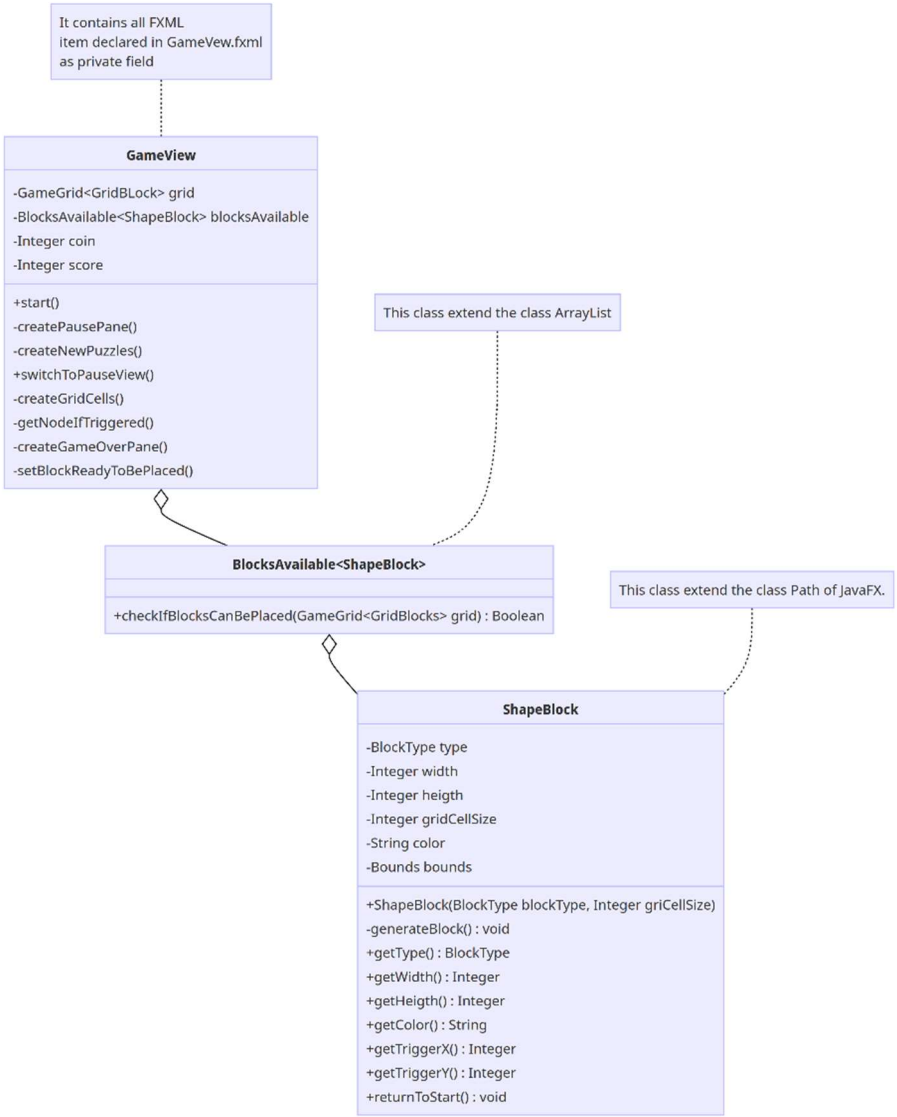
  Soluzione: Creazione della classe DataUtils che tramite i metodi di JsonUtils mi permette di caricare e salvare facilmente la griglia e il punteggio, allo stesso modo posso salvare il miglior punteggio nel suo apposito file se necessario, e poterli mostrare successivamente se e quando disponibili.



- **Controllo griglia per game over**

  Problema: Controllare ad ogni blocco piazzato se nella griglia è possibile piazzare almeno un altro blocco di quelli disponibili, perché, in caso contrario, il gioco deve segnalare il game over.

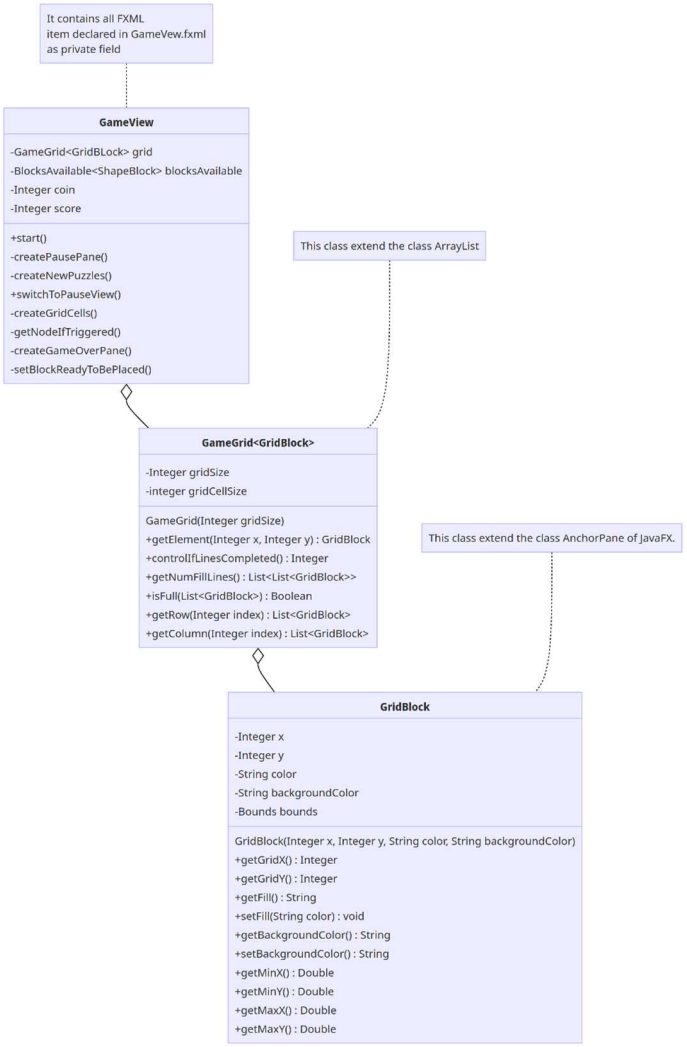
  Soluzione: Tramite la creazione della classe *BlocksAvailable* posso tenere traccia di tutti i blocchi che l'utente può piazzare, al suo interno ho implementato il metodo *checkIfBlocksCanBePlaced* che ad ogni cella vuota che incontra nello scorrimento della griglia, controlla se partendo da quella cella il blocco può essere piazzato. In caso negativo mostra la schermata di game over.



- **Controllo griglia per assegnazione punti e monete**

  Problema: Controllare ad ogni blocco piazzato se ci sono linee o colonne completate da poter rimuovere, con conseguente assegnazione di monete e punti.

  Soluzione: Ho creato la classe *GameGrid* che mi permette di raccogliere in un unico oggetto tutte le celle della griglia. Al suo interno ho implementato *controllIfLinesCompleted* che ad ogni piazzamento di un blocco controlla se ci sono righe o colonne piene, in caso positivo le cancella e restituisce il numero di linee cancellate così da permettere l'incremento delle monete e l'assegnazione incrementale dei punti.

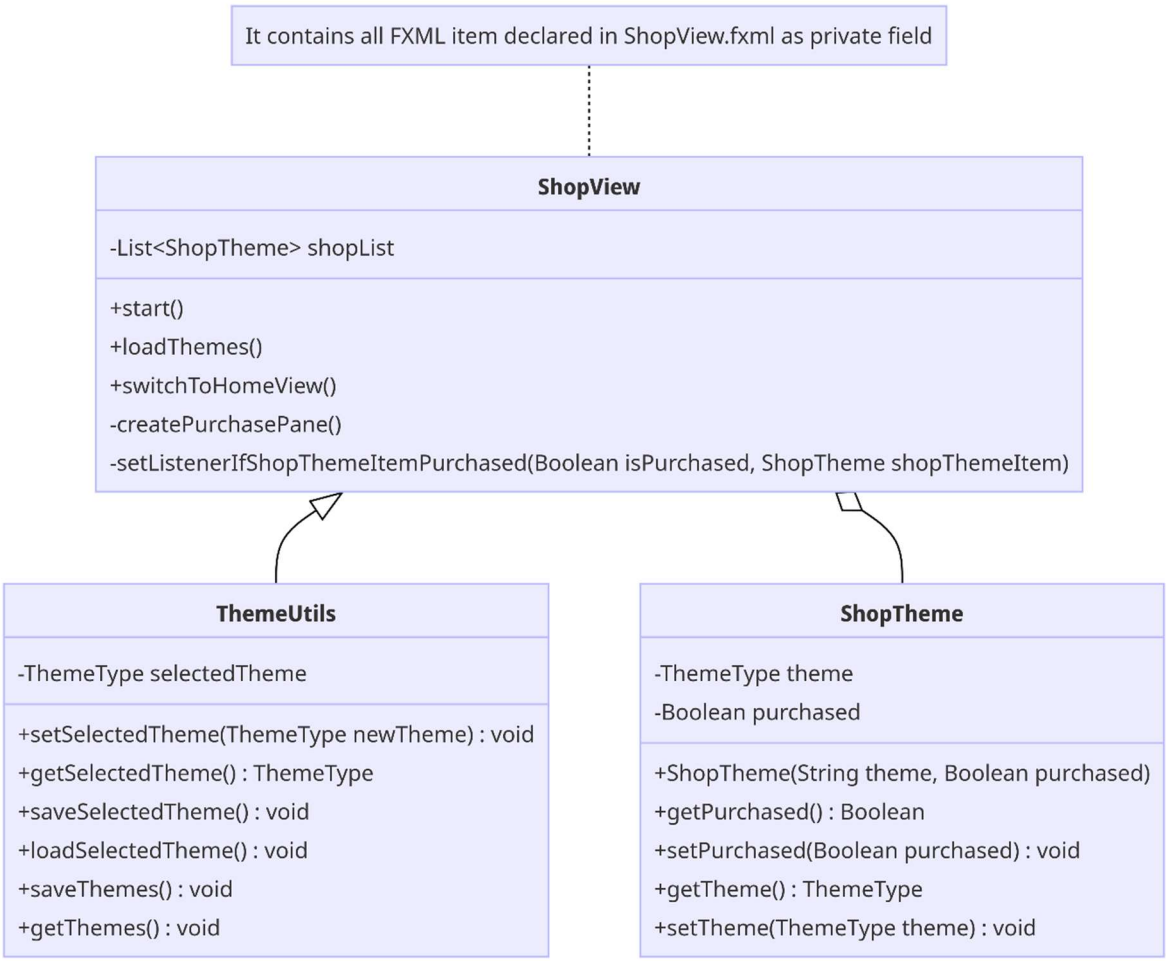


Borgiani Andrea

- ## Gestione del negozio

  Problema: Creazione di un negozio che permetta all'utente di acquistare vari temi tramite le monete, e di poter selezionare (quando già acquistati) un tema tra quelli disponibili nel negozio.

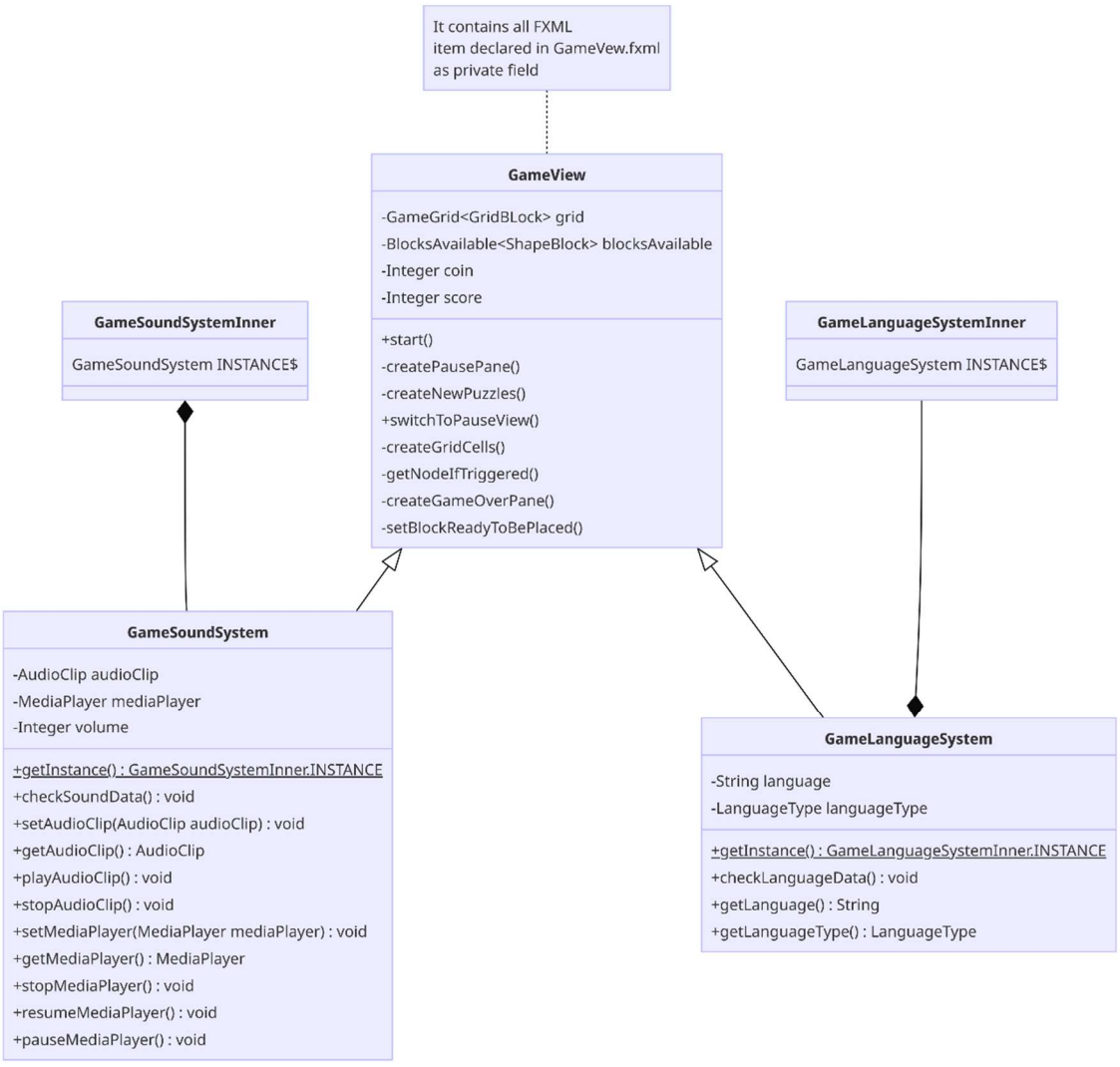
  Soluzione: Creazione della classe *ShopTheme* che al suo interno contiene tutti i campi e metodi necessari alla gestione di un tema, i quali vengono poi mostrati all'utente nella *ShopView* dove quest'ultimo può effettuare l'acquisto oppure, in caso sia già in suo possesso selezionarlo. Inoltre, tramite i metodi di *ThemeUtils* è possibile tenere traccia di tutti i dati relativi ai temi, che vengono salvati in un file json apposito.



- **Gestione audio e lingue**

  Problema: Implementare una corretta gestione dell'audio durante il gioco e delle lingue nelle diverse view. All'interno delle impostazioni, l'utente deve poter controllare il livello del volume e selezionare la lingua richiesta.

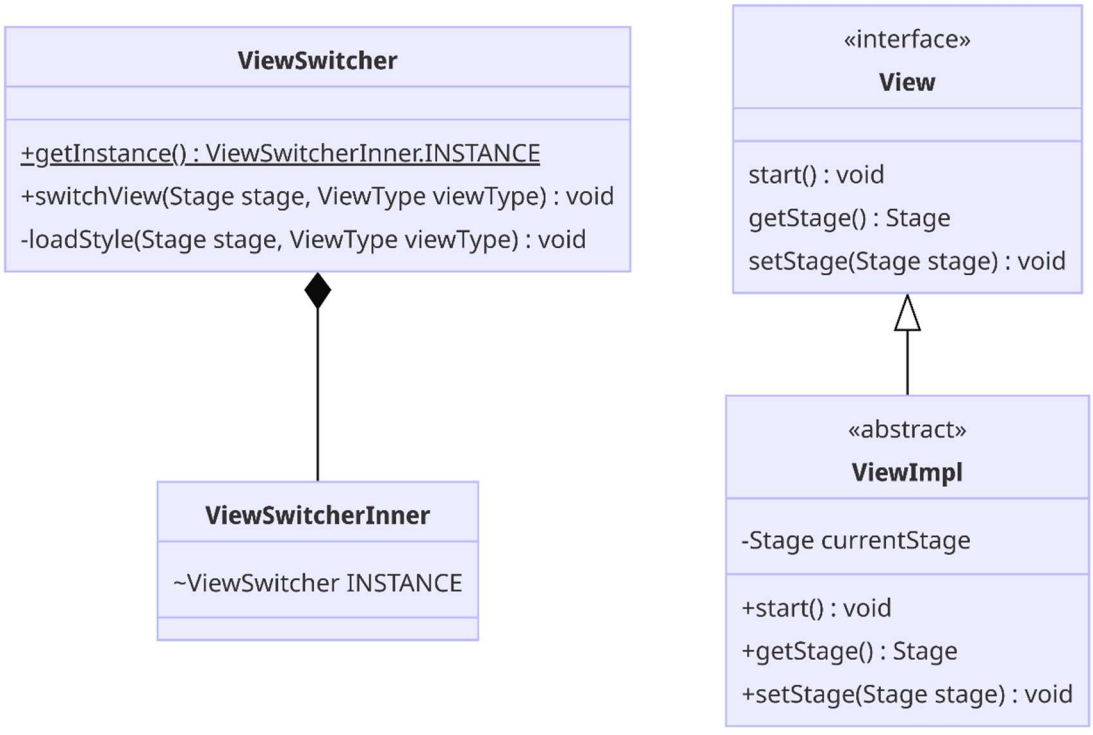
  Soluzione: Creare la classe *GameSoundSystem* per la gestione dei suoni del gioco, nella quale sono presenti i metodi che permettono la gestione e l'utilizzo dei suoni di gioco. Mentre, con la creazione della classe *GameLanguageSystem* viene gestita – tramite selezione dell'utente – la lingua da utilizzare nelle varie view. Entrambe le classi implementano il pattern Singleton, che permette loro di istanziare esclusivamente un oggetto, per evitare multiple o ripetute creazioni indesiderate.



- **Caricamento view**

  Problema: Implementare un'efficiente gestione delle view che permetta di caricarle correttamente. Inoltre, è necessario lo sviluppo di un'interfaccia comune per l'implementazione delle view.

  Soluzione: Sviluppo della classe *ViewSwitcher*, la quale consente di caricare sullo stage la view richiesta, in modo univoco e dinamico, anche grazie al pattern Singleton. Per fare ciò è stata necessario lo sviluppo dell'interfaccia *View* – che tramite la sua implementazione in *ViewImpl* – permette la creazione delle view partendo da una base comune.



Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per quanto riguarda la parte dei test, abbiamo utilizzato la libreria di *Junit*, che ci ha permesso di monitorare costantemente quelle parti di codice che reputavamo più soggette a possibili errori, ed evitare regressioni durante lo sviluppo. Il codice soggetto a test è il seguente:

- **MovementTest:** controlla che i nodi passati a *Movement* abbiano effettivamente un listener agganciato.
- **StartUpTest:** controlla che l'icona di gioco sia disponibile.
- **GameGridTest:** controlla che dopo la creazione della griglia vengano ritornati gli elementi corretti.
- **JsonUtilsTest:** controlla i metodi utilizzati per la gestione dei file json, effettuando scritture letture sui file di test.
- **RandomUtilsTest:** controlla che il metodo *getRandomPuzzle* restituisca l'oggetto corretto.
- **ViewSwitcherTest:** controlla che esista un'istanza di *ViewSwitcher*.

3.2 Note di sviluppo

Ascani Tommaso

- **Lamba:** Link
- **Listener:** Link
- **Generics:** Link
- **JavaFX:** Link
- **Json:** Link
- **Reflection:** Link
- **Logging:** Link

Borgiani Andrea

- **Lamba: Link**
- **Listener: Link**
- **Generics: Link**
- **JavaFX: Link**
- **Json: Link**
- **Reflection: Link**
- **Logging: Link**

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Ascani Tommaso

In conclusione, mi ritengo soddisfatto dell'impegno che ho messo durante lo sviluppo del progetto, ma un po' meno del risultato ottenuto. Considero il nostro lavoro di buona qualità, ma ritengo che con un po' più di esperienza ed attenzione, avremmo potuto svolgere meglio alcune parti del progetto, che in alcuni casi non hanno rispetto le aspettative che mi ero prefissato. Per quanto riguarda la collaborazione con il mio collega, penso sia stata per entrambi ottima e di grande importanza, ed ha permesso ad entrambi di migliorare le proprie conoscenze e di ottimizzare il processo di sviluppo. Tutto sommato mi è stata utile come prima vera esperienza di progetto in un gruppo, e mi ha consentito di comprendere quelle che sono i principali vantaggi e svantaggi del lavorare in gruppo.

Borgiani Andrea

Questo progetto è stato per me la prima esperienza di lavoro in gruppo, ma, nonostante questo e le diverse difficoltà riscontrate durante il progetto, mi ha insegnato molto per quanto riguarda la comunicazione e l'organizzazione delle cose da fare. Sicuramente se, nei mesi precedenti, avessi seguito il corso, lo svolgimento e l'implementazione della mia parte di progetto sarebbe avvenuta più velocemente e, senza dubbio, dal punto di vista della qualità del codice, ne sarebbe uscito un prodotto più maturo e completo. Nonostante questo, nel complesso mi ritengo soddisfatto del lavoro svolto.