Relazione Programmazione ad Oggetti

Elia Zavatta, Giovanni Maffi, Pietro Lelli,
Riccardo Leonelli, Andrea Brigliadori

25 Giugno 2021

Indice

Capitolo 1

Analisi

1.1 Requisiti

L'applicazione To kill a Mockingbird emula il gioco 2D platform "Crossy Road". Lo scopo del gioco è quello di arrivare il più lontano possibile con il proprio personaggio senza morire. Ci sono ostacoli che causano morte immediata, come per esempio la collisione con auto e treni. Per superarli, il giocatore deve avanzare quando il percorso è libero. Si guadagnano punti avanzando verso l'alto nella mappa.

Requisiti funzionali

- Il videogioco si compone di un menù principale, in cui sarà possibile iniziare una nuova partita. All'avvio di essa verrà generata la mappa di gioco ed personaggio principale si troverà in una zona iniziale composta solamente da erba ed alberi, una volta superata ci troveremo di fronte a strade e ferrovie su cui passeranno automobili, camion e treni.

- Ogni volta che verrà iniziata una nuova partita sarà generata una mappa la cui disposizione di strade, ferrovie e prato è casuale. La mappa scorrerà autonomamente e verrà generata andando avanti nella partita. Su ogni strada transiteranno macchine o camion, mentre su ogni ferrovia passerà un treno. Nelle zone di prato saranno disposti casualmente alberi che saranno d'ostacolo al personaggio.

- Se il personaggio principale viene in contatto con un veicolo la partita viene conclusa.

- Ad ogni nuovo passo in avanti del personaggio guadagna un punto. Lungo il percorso sono posizionate casualmente monete raccoglibili passandoci sopra.
- Sarà presente un menù iniziale, uno di pausa e uno finale.

Requisiti non funzionali

- Il gioco dovrà risultare graficamente fluido, evitando rallentamenti o interruzioni nella presentazione delle immagini.

1.2 Analisi e modello del dominio

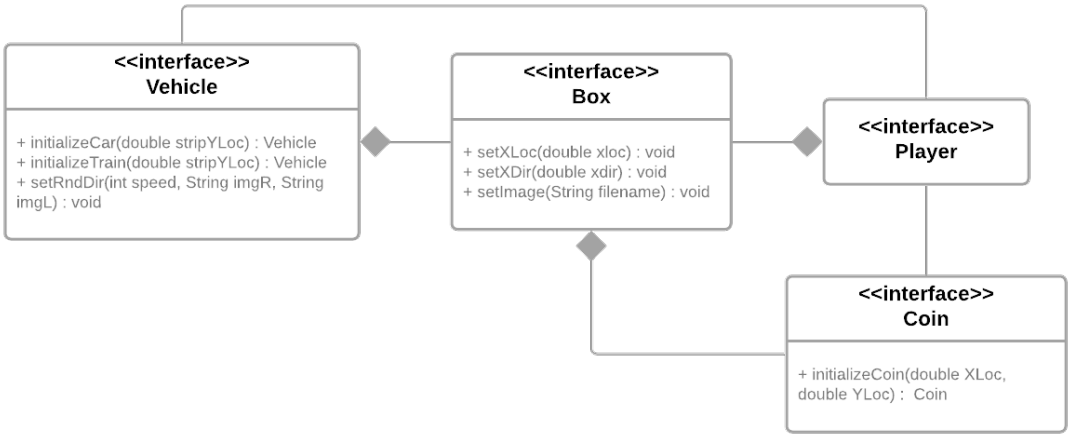


Figura 1.1: Schema UML che rappresenta il dominio applicativo

Il modello del dominio applicativo è strutturato come rappresentato in figura 1.1. La mappa di gioco si compone da diverse strisce composte a loro volta da caselle, esse si muovono veticalmente, possono contenere ostacoli non attraversabili e monete. Sono presenti strisce di strada e ferrovia su cui transitano continuamente veicoli a velocità casuale, a contatto con il personaggio principale interromperanno la partita. Essi sono i nemici del personaggio e possono transitare solamente nella loro apposita striscia, mentre il player si può muovere liberamente nelle quattro direzioni.
Il nostro personaggio non può attraversare i bordi laterali e quello superiore, mentre se si dovesse trovare in corrispondenza del bordo inferiore questo porterebbe al Game Over.
Tramite il menù principale possiamo avviare una partita, ad ogni nuovo passo in avanti del personaggio accumuleremo un punto. In ogni momento possiamo richiamare il menù di pausa per interrompere il gioco e visionare i comandi. La partita è potenzilmente infinita, difatti finchè il player non viene colpito o esce dalla mappa il gioco continua.

Capitolo 2

Design

2.1 Architettura

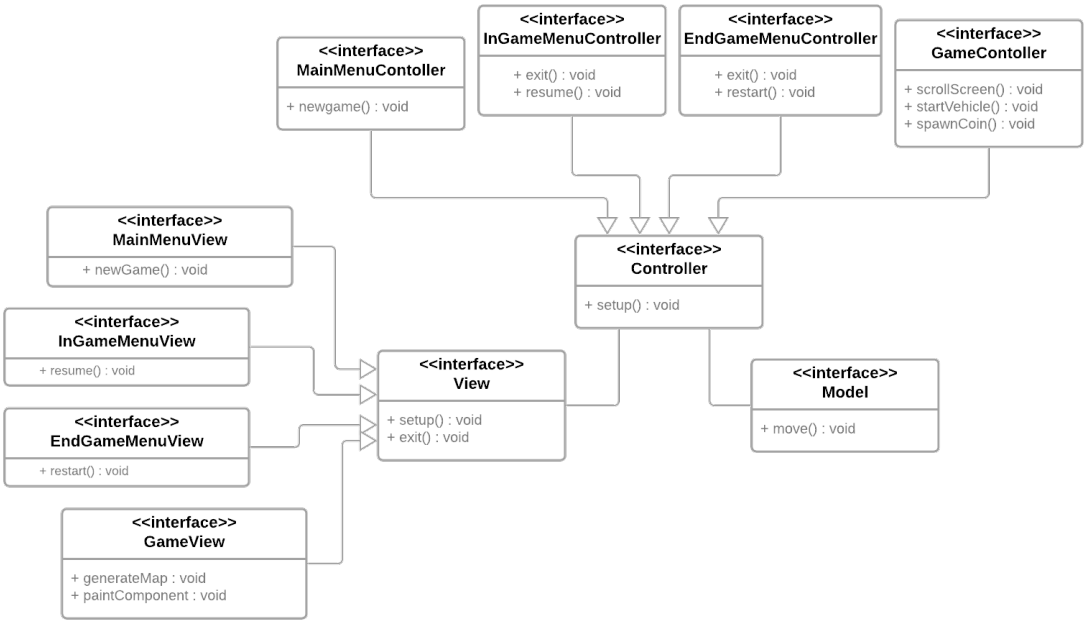


Figura 2.1: UML rappresentante l'architettura dell'applicazione

To kill a Mockingbird segue il pattern architetturale MVC (figura 2.1). In particolare è composto da quattro `View`: di gioco, del menu principale, del menu in-game e del menù finale. Ogni `View` rappresenta graficamente uno stato possibile dell'applicazione, ad ognuna di esse è associato un `Controller` che coordinerà adeguatamente l'interazione con il `Model`.
All'avvio dell'applicazione il controller attivo sarà il `MainMenuController`, il quale si occuperà di far visualizzare la `MainMenuView` a schermo e permetterà di iniziare una nuova partita.
Una volta iniziato il gioco la vista principale sarà gestita dalla `GameView` , mentre dello scorrimento della mappa e del posizionamento dei vari elementi sopra di essa se ne occuperà il `GameController` .
Durante la partita, sarà sempre accessibile il menu di pausa per interrompere momentaneamente il gioco. In tal caso il controller attivo diventerà `InGameMenuController`, che mostrerà la relativa `InGameMenuView`.
La logica dietro a tutti gli oggetti che troveremo nell'applicazione viene gestita dal `Model` e dalle sue sottocomponenti.

2.2 Design dettagliato

In questa sezione andremo a descrivere elementi di design nei loro dettagli. Ogni membro del gruppo presenta le sue principali realizzazioni che hanno composto l'applicazione.

Elia Zavatta

- **Model Strip**

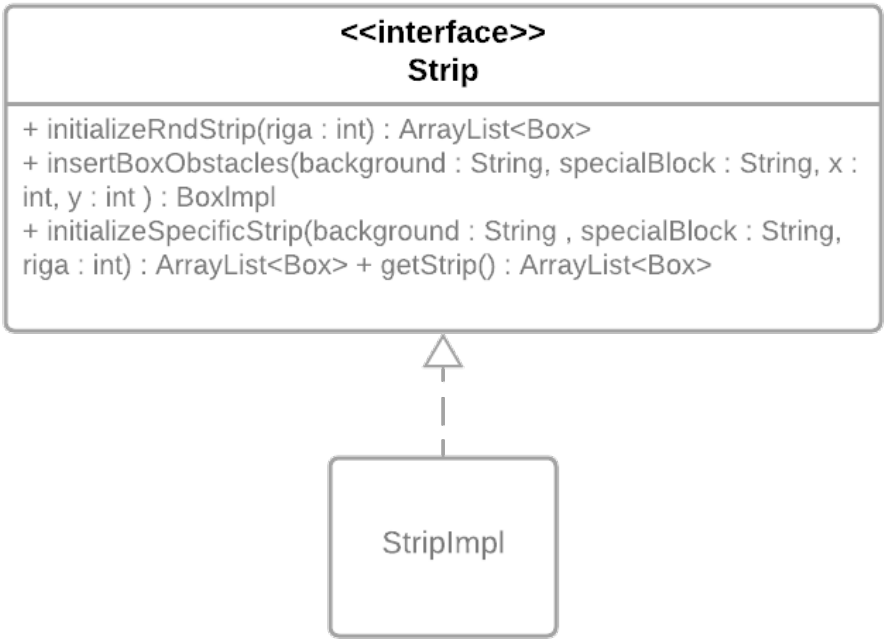


Figura 2.2: Schema UML rappresentante le Strips

La mappa di gioco si compone da striscie orizzontali chiamate `Strip`, esse sono ArrayList di `Box` che verranno stampate in continuazione per permettere al gioco una durata potenzialmente infinita.
L'interfaccia `Strip` conterrà i metodi per creare randomicamente o meno queste striscie, che verranno poi implementati in `StripImpl`.
Nel gioco sono presenti diversi tipi di Strip: `Grass` conterrà ostacoli (alberi), non nocivi al personaggio, generati casualmente dal metodo insertBoxObstacles(); `Road` avrà mezzi stradali ad attraversarla e `Rail` presenterà il passaggio di treni.
Ogni Strip sarà composta da Box di una specifica grafica in base al suo ambiente, così facendo si possono facilmente implementare nuove zone di mappa.
Nel GameController attraverso i metodi setInitialPosition() e generateMap() andremo a generare la mappa utilizzando queste strip.

- **Model Enemy**

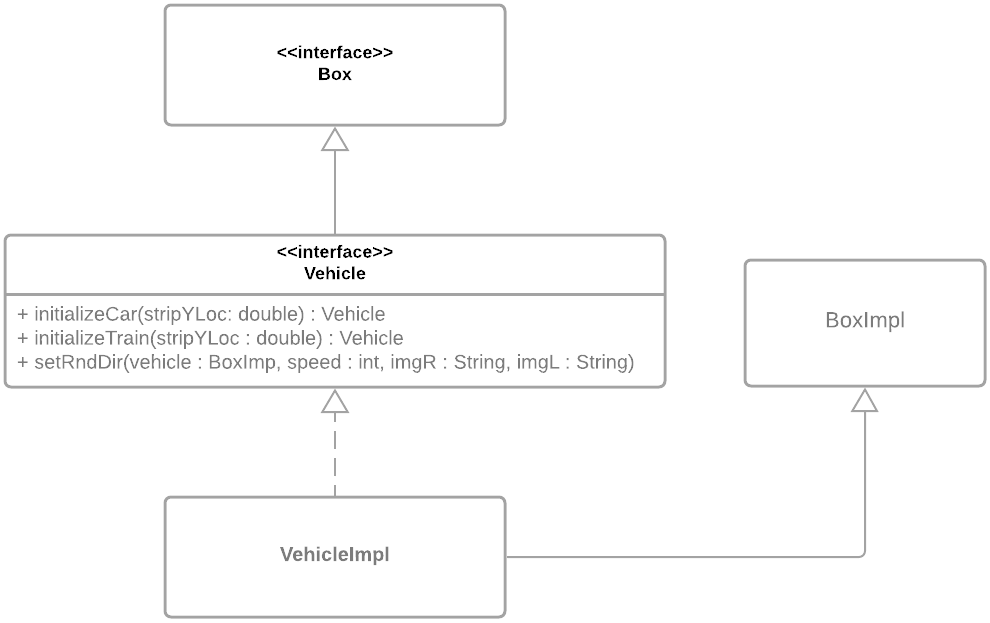


Figura 2.3: Schema UML che rappresenta i Vehicle

Nel Model Enemy sono contenuti tutti i nemici del personaggio principale che, in questo gioco, sono solo veicoli rappresentati dagli oggetti Vehicle.
L'interfaccia `Vehicle` estende da `Box` e la sua implementazione è contenuta nella classe `VehicleImpl` che estende a sua volta `BoxImpl`. Ne consegue che ogni veicolo possiede tutti i metodi di Box e ne aggiunge ulteriori caratteristici come: velocità e tipo di veicolo.
Il metodo initializeCar() creerà randomicamente una macchina o un camion, mentre initializeTrain() creerà un treno; entrambi i metodi restituiranno il mezzo creato e inizializzato. Il GameController attraverso il metodo spawnVehicle() si occupa di posizionarli su strade o ferrovie, con startVehicle() li fa muovere secondo la loro velocità e li riposiziona una volta terminata la loro corsa.

Riccardo Leonelli

- **Player**
  Il player rappresenta l'oggetto principale del gioco ed è modellato nel package `model.player`. L'interfaccia `Player` estende dall'interfaccia `Box` e la sua implementazione è contenuta nella classe `PlayerImpl`.
  Il player possiede tutti i metodi di `BoxImpl` relativi alle coordinate e al movimento simultaneo con mappa. In aggiunta contiene metodi per l'utilizzo delle monete raccolte di livello quantitativo.
  Il player è l'unica entità dell'elaborato in grado di muoversi nella mappa sotto il diretto controllo dell'utente. In particolare la classe del model `PlayerMovementImpl` , che estende da `PlayerImpl` , si occupa della gestione dei movimenti. In questo modo ho reso indipendente il personaggio dai suoi movimenti che eventualmente possono essere modificati senza alterare l'implementazione del personaggio.

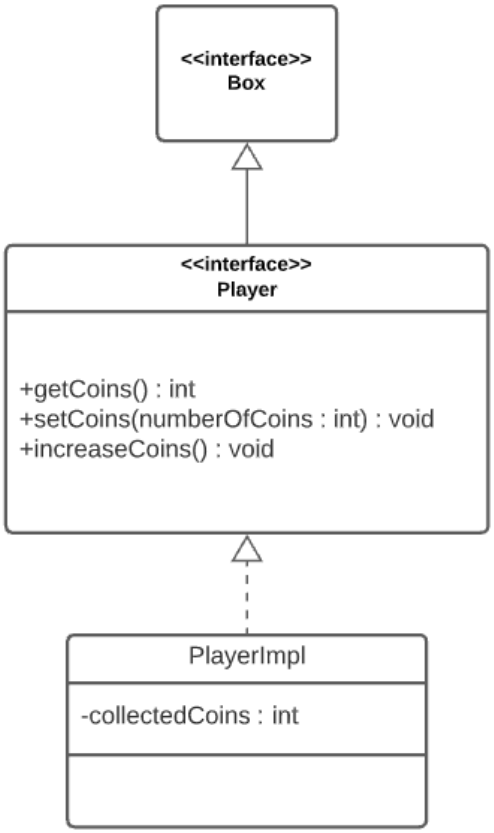


Figura 2.4: Schema UML che rappresenta il player

- **KeyInput**
  Per gestire gli input da tastiera ho realizzato il package `input.player` contenente l'interfaccia `KeyInput` e la rispettiva implementazione nella classe `KeyInputImpl`. Quest'ultima si occupa dei movimenti del personaggio e l'apertura del menù di pausa , inoltre permette il conteggio dello score del player aggiornato in ogni frame.
  Per gestire i movimenti del player utilizza il metodo `moveDirection` presente nell'interfaccia `PlayerMovement` , in grado di cambiare le sue coordinate a seconda del tasto cliccato dall'utente.

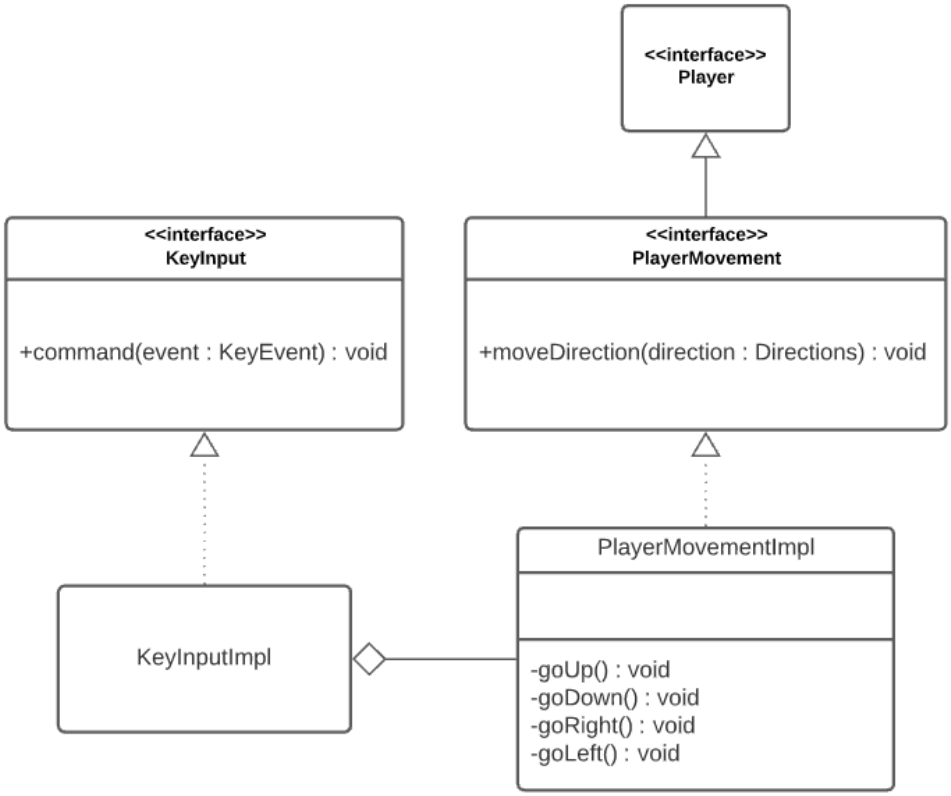


Figura 2.5: Schema UML che rappresenta la gestione dei movimenti del player

Andrea Brigliadori

- **CollisionController**

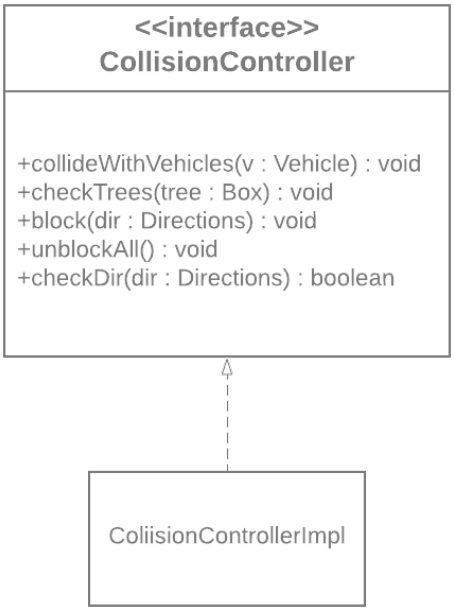


Figura 2.6: Schema UML descrivente il CollisionController

Il `CollisionController` è il gestore delle collisioni. Si occupa di gestire le collisioni tra tutti gli oggetti di gioco quali monete, auto, treni e alberi. Viene inizializzato dal `GameController` e necessita di accedere al `Player` per poterne determinare la posizione in ogni momento. Viene chiamato con regolarità ad ogni ciclo di gioco per gestire le collisioni con i vari oggetti presenti sulla mappa.
Il metodo `collideWithTrees` controlla le quattro caselle adiacenti al giocatore, e se è presente un albero impedisce al `Player` di muoversi in quella direzione. Quando la classe `Input` dovrà muovere il giocatore, chiederà al `CollisionController` se la direzione selezionata è percorribile. Il metodo `checkBorders` si comporta in modo analogo, bloccando il movimento oltre il bordo superiore e laterale. Alla collisione con un oggetto di tipo `Vehicle`, controllato da `collideWithVehicles`, il `CollisionController` comunicherà al `GameController` di terminare la partita. Un'altro evento che causa la fine della partita è lo scavalcamento del bordo inferiore dello schermo.

Giovanni Maffi

- **Box**

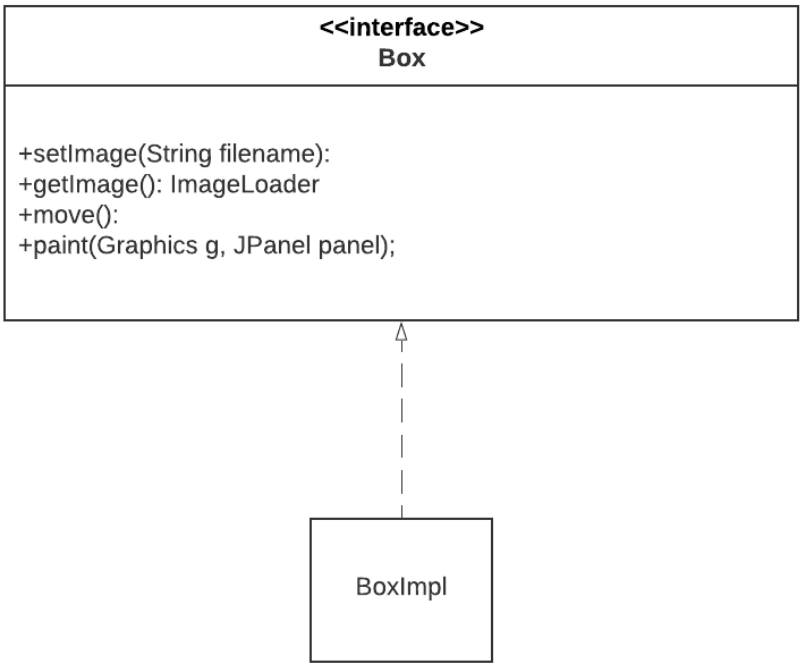


Figura 2.7: Schema UML descrivente Box

L'interfaccia `Box` è la principale componente per la creazione della mappa, infatti, come detto in precedenza, le strip che compongono la mappa sono formate da box.
All'interno di questa interfaccia sono presenti i metodi per impostare la posizione, la direzione di movimento e l'immagine (erba, strade, rotaia, erba con albero) per ogni box, mentre BoxImpl, la classe che realizza l'interfaccia box, contiene le rispettive variabili dei metodi prima citati. Inoltre l'interfaccia Box viene utilizzata anche per la creazione di tutti gli "oggetti dinamici": player, vehicles e coins.

- **Coin**

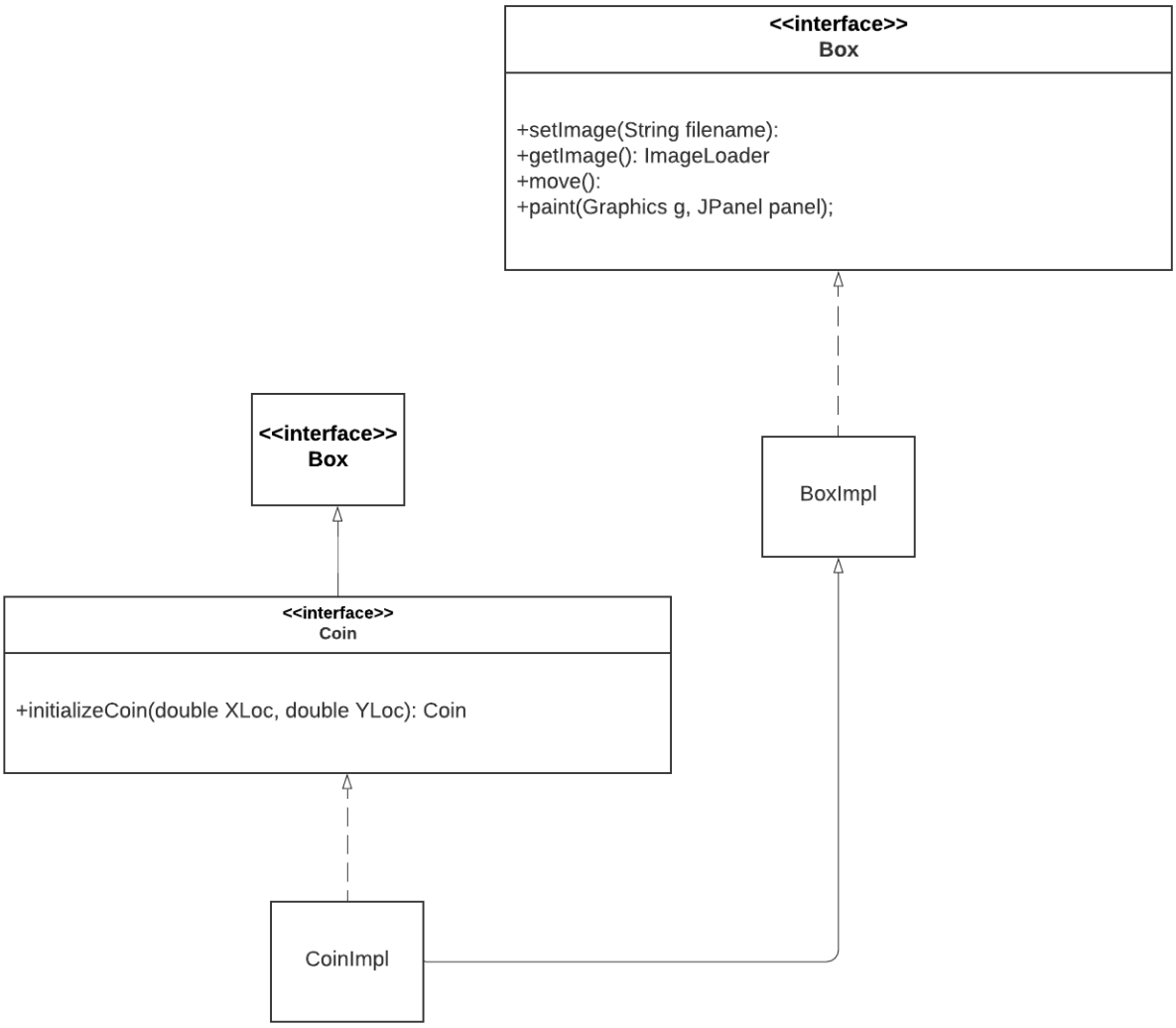


Figura 2.8: Schema UML descrivente Coin

L'interfaccia `Coin` è l'interfaccia che permette (tramite CoinImpl) la creazione e la disposizione delle monete sulla mappa di gioco. Implementa l'interfaccia Box. Ha soltanto un metodo (initializeCoin) che permette di impostare le coordinate e l'immagine per una singola moneta. La disposizione e la frequenza di inserimento delle monete avviene in modo causale.

Pietro Lelli

- **GameMenu**

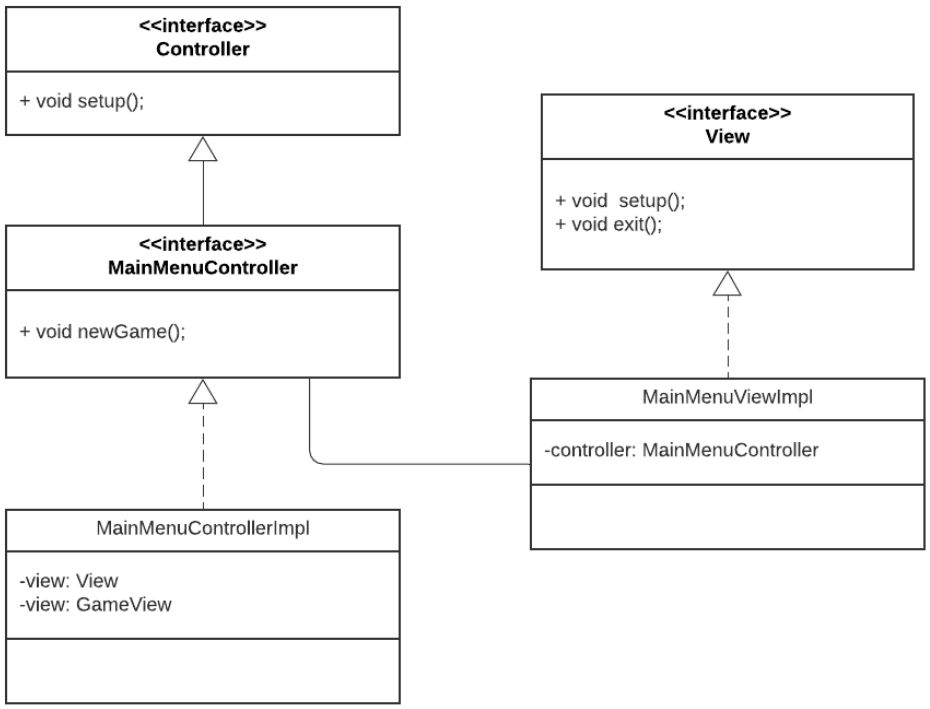


Figura 2.9: Schema UML che rappresenta GameMenu

Il menù iniziale di gioco è strutturato con il pattern architetturale MVC. Questo permette di mantenere separate le tre parti: `Model`, `View` e `Controller`. La `View` si occupa della parte grafica, corrisponde quindi alla parte dell'interfaccia utente.
Il `Controller` si occupa di gestire e far interagire la grafica con la logica, richiamando gli opportuni metodi.
Il menù di gioco dà la possibilità di iniziare la partita, di uscire dal gioco (Exit) e di visualizzare i controlli di gioco, quindi come muovere il personaggio e come mettere in pausa il gioco.
Premendo il pulsante START il JPanel relativo al menù scomparirà e verrà creata la mappa di gioco.

- **InGameMenu**

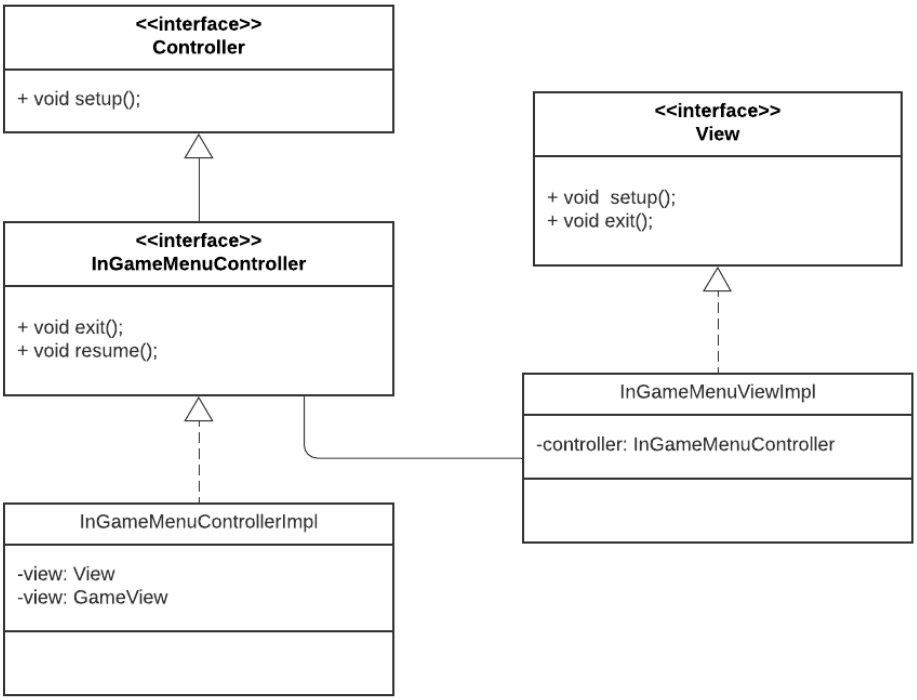


Figura 2.10: Schema UML che rappresenta InGameMenu

Il menù InGame è strutturato anche questo con il pattern architetturale MVC. Il menù di gioco dà la possibilità di riprendere la partita corrente chiudendo il menu (Resume) e cliccando da tastiera "esc", di uscire dal gioco (Exit) e di visualizzare i controlli del gioco.
Questo menù verrà aperto quando durante il gioco verrà premuto il tasto "esc", in questo modo verrà visualizzato il JPanel relativo al menù. Cliccando poi sul bottone "Resume" questo menù scomparirà e verrà visualizzata nuovamente la mappa di gioco

EndGame

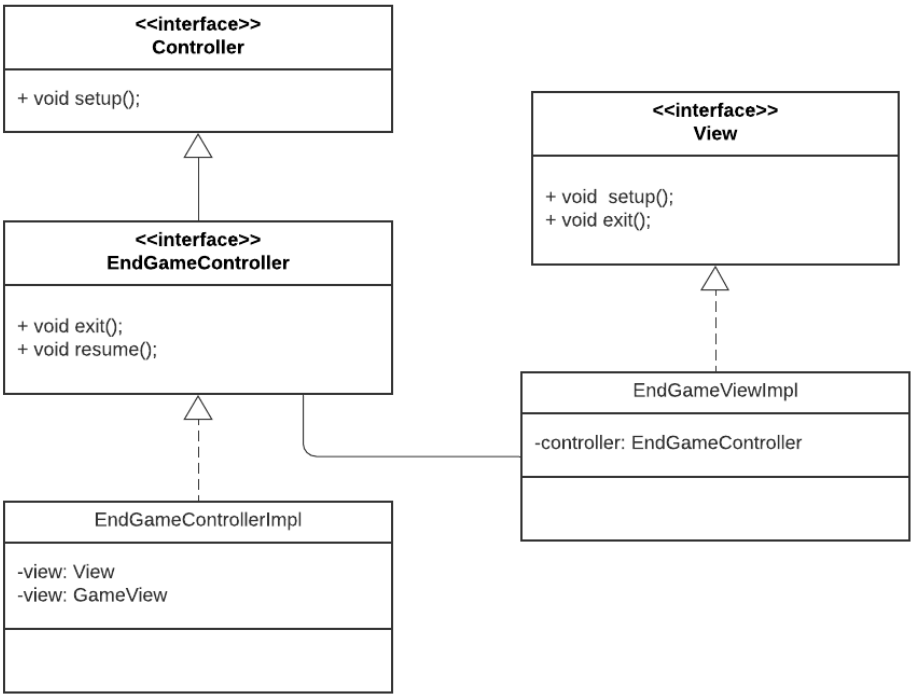


Figura 2.11: Schema UML che rappresenta EndGame

Il menù EndGame è strutturato anche questo con il pattern architetturale MVC. Il menù di gioco dà la possibilità di cominciare una nuova partita (Restart), di uscire dal gioco (Exit).
Questo menù verrà aperto al termine del gioco, quindi quando il personaggio mosso dall'utente viene investito da un ostacolo dinamico (auto o treno) o quando il personaggio non si muove in avanti per troppo tempo rimanendo così indietro rispetto alla mappa di gioco che si muove in avanti.
Quando questo accade verrà richiamato il metodo per visualizzare questo menù.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

In questa parte verranno illustrati i test che sono stati effettuati per verificare il corretto funzionamento delle componenti principali di To Kill a Mockingbird.

- **TestEnemy :**
  Il test consente di verificare il corretto funzionamento dei veicoli, in particolare controlla se la posizione di spawn, velocità e tipo del veicolo sono corretti.

- **TestBox :**
  Il test consente di verificare il corretto funzionamento delle box, controllando se le posizioni sono assegnate nel modo giusto e verificando il corretto funzionamento del metodo move (metodo per scorrere la mappa).

- **TestStrip :**
  Il test consente di verificare la corretta creazione delle `Strip`, facendo controlli mirati sulle varie modalità di creazione di esse.

- **TestCoin :**
  Il test consente di verificare il corretto funzionamento delle monete, verificando se le posizioni sono assegnate nel modo giusto e se viene effettuato lo scorrimento.

- **TestPlayer :**
  Il test permette di verificare il corretto funzionamento del player verificando le rispettive caratteristiche iniziali , come la posizione e l'acquisizione delle monete.

- **TestCollision :**
  Il test permette di verificare il corretto funzionamento delle collisioni tra `Player` e tutti gli oggetti di gioco per accertarsi che non vi siano bug o errori.

3.2 Metodologia di lavoro

Per la realizzazione di To kill a Mockingbird si è suddiviso equamente il carico di lavoro, assegnando ad ognuno una parte di logica.
È stato utilizzato il DVCS Git, fondamentale per la collaborazione tra i membri del progetto. Ognuno è in possesso di una copia locale del repository e ad ogni importante modifica procede ad aggiornare quello remoto.
Di seguito vengono illustrate le parti sviluppate da ogni membro del team.

Elia Zavatta: Realizzazione delle `Strip` per la creazione della mappa (`Grass`, `Road`, `Rail`) e collocazione degli ostacoli(Alberi) nelle zone di prato. Implementazione dei `Vehicle` di differenti tipi (`Car`, `Truck`, `Train`), generazione randomica di ognuno di essi sull'apposito ambiente ed assegnazione di un movimento ad attraversare la mappa. Creazione dell'intera mappa, scorrimento della mappa, movimento dei veicoli e stampa di tutti i componenti.

Riccardo Leonelli: Realizzazione del personaggio principale e gestione dei suoi movimenti nel mondo di gioco. Gestione dei comandi da tastiera e implementazione delle rispettive classi.

Giovanni Maffi: Creazione della mappa di gioco tramite strip e box. Realizzato inoltre lo scorrimento della mappa e la rigenerazione di questa una volta uscita dai bordi.
Infine è stata fatta la creazione e la generazione delle monete lungo il percorso, che il giocatore potrà raccogliere. La generazione delle monete può verificarsi in qualsiasi punto (eccetto sugli alberi).

Pietro Lelli: Realizzazione dell'interfaccia grafica e della logica del `GameMenu`, il menù iniziale che permette di avviare il gioco, l'`InGameMenu`, il menù che mette in pausa e l'`EndGameMenu` il menù di fine gioco che al GameOver permette di cominciare una nuova partita.
A questo va aggiunta la gestione del punteggio e del GameOver.

Andrea Brigliadori: Implementazione delle collisioni tra il player e i vari oggetti di gioco attraverso il `CollisionController`. Gestione degli eventi legati alla collisione con i bordi della mappa, con gli alberi, con i veicoli e con le monete.

In cooperazione: `GameController` e `GameView` sono le parti centrali del progetto, all'interno sono presenti porzioni di codice utili ad ogni membro. Queste classi sono state realizzate in cooperazione e ogni membro ha contribuito aggiungendo le funzionalità necessarie alla propria parte.

Divisione classi e package svolti singolarmente :

Elia Zavatta:

- model.enemy
- model.map.Strip.java
- model.map.StripImpl.java
- model.map.StripEnvironment.java
- view.ImageLoader.java
- test.TestEnemy.java
- test.TestStrip.java

Riccardo Leonelli:

- model.player
- input.player
- test.TestPlayer.java

Giovanni Maffi:

- model.score
- model.map.Box.java
- model.map.BoxImpl.java
- test.TestBoxes.java
- test.TestCoin.java

Pietro Lelli:

- controllers.MainMenuController.java
- controllers.MainMenuControllerImpl.java
- controllers.InGameMenuController.java
- controllers.InGameMenuControllerImpl.java
- controllers.EndGameMenuController.java
- controllers.EndGameMenuControllerImpl.java
- view.mainMenuViewImpl.java
- view.InGameMenuViewImpl.java
- view.EndGameMenuViewImpl.java

Andrea Brigliadori:

- controller.CollisionController.java
- controller.COllisionControllerImpl.java
- test.TestCollision.java

3.3 Note di sviluppo

Di seguito si elencano per ogni componente del team le funzionalità avanzate del linguaggio utilizzate.

Elia Zavatta:

- `Lambda expression`: per muovere i veicoli nel GameController e stamparli nel metodo paintcomponent del GameView.
- `Stream`: per restituire il numero di alberi in una Strip in StripImpl.

Andrea Brigliadori:

- `Stream`: utilizzati per filtrare i veicoli e le monete all'interno del `GameController`
- `Optional`: impiegati per restituire l'eventuale nemico in collisione con il personaggio.

Pietro Lelli:

- `Lambda expression`: nell'aggiunta degli eventi dei JButton e dei componenti grafici dei menù.

Giovanni Maffi:

- `Lambda expression`: per muovere le monete nel GameController.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

E' possibile ampliare il software sviluppato aggiungendo diversi tipi di nemici e ambienti, uno shop dove utilizzare le monete raccolte per comprare nuovi personaggi e abilità speciali del player. Di seguito si presentano le autovalutazioni dei singoli componenti del team.

Elia Zavatta:

Sono molto soddisfatto del mio contributo per la realizzazione di questo progetto. Durante la programmazione esso mi ha messo alla prova ed ha stimolato il mio interesse verso la progettazione software e la programmazione ad oggetti, spingendomi a mettere in discussione ed ottimizzare il codice che producevo di volta in volta.
Per me questo è stato il primo lavoro di gruppo, non è stato facile coordinarsi al meglio tra di noi anche per via delle lezioni svolte a distanza che non ci permettevano di incontrarci per discutere del progetto. Nello stesso periodo eravamo impegnati in progetti per altri esami che hanno portato via molto tempo ma siamo riusciti a tenerci in contatto con varie videochiamate e messaggi. Per quanto riguarda le difficoltà incontrate, penso si siano presentate soprattutto durante la fase di progettazione iniziale, in particolare nella divisione delle varie parti secondo il modello MVC e nella creazione della mappa.

Riccardo Leonelli:

L'elaborato che abbiamo sviluppatto è stato complessivamente un ottimo lavoro di cooperazione. Tuttavia all' inizio ho riscontrato dei problemi in quanto non avendo conoscenze preliminari in progetti di questo tipo, ho do-

vuto approfondire molti aspetti. Nel mio caso questo è il primo elaborato di gruppo che io abbia mai svolto che mi ha permesso di apprendere al meglio aspetti della programmazione ad oggetti che non conoscevo. Per quanto riguarda la mia parte sono generamente soddisfatto, la parte che mi ha messo più alla prova è stata la gestone degli input da tastiera.
Questo elaborato mi ha permesso di ampliare le mie conoscenze della materia e soprattutto mi ha fatto apprendere come si deve lavorare in un team.

Giovanni Maffi:

Tutto sommato, sono soddisfatto del mio contributo per questo progetto. E' stato il mio primo lavoro di gruppo ed è stato lungo e faticoso, siccome le conescenze per gestire la parte grafica (inserimento immagini, movimento mappa...) non erano così avanzate; inoltre, anche le mie conoscenze sul MVC erano scarse, quindi ho avuto delle difficoltà iniziali per la gestione del progetto. Ma con il passare del tempo e confrontandomi con i miei compagni, sono riuscito a risolvere questi problemi e portare a termine il mio lavoro.

Pietro Lelli:

Mi reputo abbastanza soddisfatto della realizzazione di questo progetto.
Inizialmente durante la fase di progettazione concettuale ho riscontrato alcune difficoltà per quanto riguarda il lavoro di gruppo, essendo il primo progetto di gruppo al quale ho partecipato. La difficoltà principale è stata quella di implementare il pattern architetturale MVC. Poiché inizialmente non lo conoscevo sufficientemente e ho utilizzato perte del tempo per lo studio di questo.
Ho trovato particolarmente stimolante la realizzazione di questo progetto, in quanto mi ha permesso di affrontare e trovare soluzioni ai problemi incontrati durante lo sviluppo dell'applicazione, inoltre mi ha fatto capire come si lavora in gruppo e quali difficoltà questo comporta.

Andrea Brigliadori:

Sono soddisfatto del lavoro svolto, è stato il mio primo progetto di gruppo di questo livello e ho trovato molto difficile, soprattutto all'inizio coordinarsi nello svolgere ognuno le proprie parti. È stato interessante approcciarsi all'MVC per la prima volta e lo studio di essa è risultato fondamentale per lo svolgimento del progetto. Una difficoltà che non mi aspettavo è stata la revisione e pulizia del codice, che è risultata molto più dispendiosa di quanto pensassi.
Sono rimasto colpito dall'utilità e dalla potenza di git che ci ha permesso di sviluppare da distanza la totalità del codice e non vedo l'ora di utilizzarlo in futuro.

Appendice A

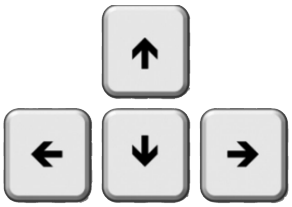
Guida utente

Di seguito si illustra la configurazione dei comandi di gioco.

Main Menu:

Figura A.1 : schermata del Main Menu.

Movimento:



Freccia su: Movimento del personaggio verso l'alto.
Freccia giù : Movimento del personaggio verso il basso.
Freccia sinistra: Movimento del personaggio verso sinistra.
Freccia destra: Movimento del personaggio verso destra.

Apertura Menu In Game:

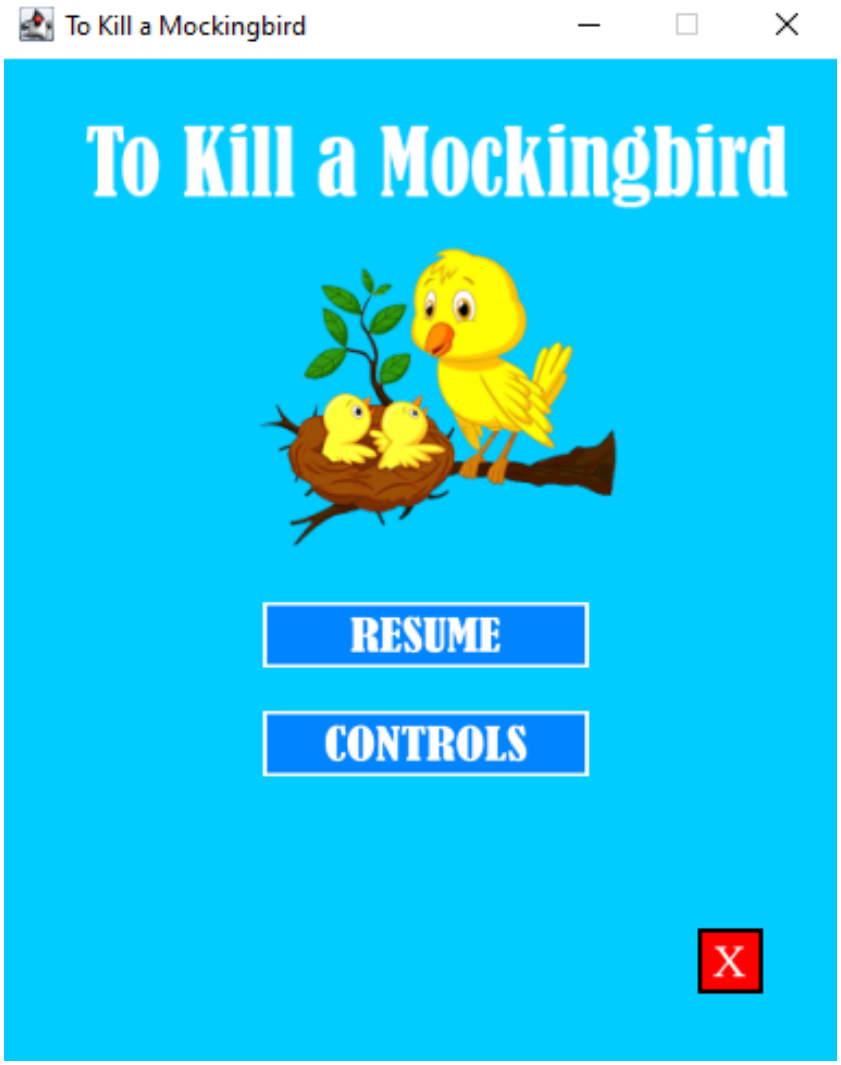


Figura A.2 : schermata del Menu In Game.

Figura A.3 : schermata dell' EndGame.