Prof vs Students

Anna Bartolucci
Alesja Delja
Eleonora Falconi

A.A. 2023-2024

Indice

Capitolo 1

Analisi

Il progetto consiste nella creazione di un'applicazione informatica che riproduce il gioco Plants vs Zombie, ma a tema universitario in modalità survival. Nel gioco ci sarà un campus universitario dove i professori sono le difese e gli studenti sono gli zombie assetati di conoscenza.
I professori, che personificano le piante, vengono scelti e posizionati dall'utente difendendo la conoscenza, mentre gli studenti, che personificano gli zombie, vengono gestiti dal software stesso e devono cercare di superare gli ostacoli nella loro strada per raggiungere la conoscenza e il successo accademico.
Il gioco termina quando uno studente arriva alla fine della griglia, che simboleggia la conoscenza, quando non ci sono più professori in gara e l'energia è a 0, perciò non è più possibile reperirli, oppure quando scade il tempo. L'utene vince se alla fine del tempo ci sono ancora professori in vita.

1.1 Requisiti

Requisiti funzionali

- All'avvio del gioco verrà visualizzata una schermata principale che include un menù in cui si può scegliere se far partire una partita, vedere come si gioca insieme ai tipi di professore che può usare ed uscire dal gioco.
- Durante la partita il videogioco dovrà permettere al giocatore di tenere sotto controllo le proprie risorse.
- Ogni nuova ondata verrà generata dopo un certo periodo di tempo prestabilito.

- Il movimento dei nemici sarà da destra verso sinistra, mentre le difese, una volta posizionate, saranno immobili e non potranno più essere spostate sino alla loro distruzione.
- Il gioco dovrà tener conto della possibilità di essere fermato prima della fine e tornare al main menù a discrezione del giocatore.
- Una partita termina quando il giocatore decide di arrendersi, uscendo quindi dal gioco, o quando viene sconfitto, ossia quando almeno uno studente raggiunge il margine sinistro della mappa.
- Ogni volta che l'utente preme per uscire gli si aprirà una schermata di conferma.

1.2 Analisi e modello del dominio

Il videogioco sviluppato sarà un survival game, ossia il giocatore dovrà giocare sino al termine del tempo. Nel menù principale l'utente potrà far partire la partita principale, controllare come si gioca ed uscita dal gioco.
Una volta fatto partire il gioco il giocatore si troverà una mappa di gioco che sarà composta da una griglia rettangolare suddivisa in celle. Una riga orizzontale di celle costituisce una corsia. Una partita viene disputata tra due fazioni contrapposte, quella del corpo docente, gestita dal giocatore, e quella del corpo studentesco, gestita dal software. Lo scopo è resistere ad ondate successive degli studenti che tenteranno di attraversare la mappa, per impedirlo il giocatore dovrà utilizzare le proprie risorse per posizionare i professori necessari ad impedire agli studenti di avanzare.
La risorsa usata nel gioco è l'energia che viene inizializzata a numero precedentemente fissato in modo da far apparire inizialmente qualche professore di tipologia `normale`normale o `tutor`. Per posizionare un professore l'utente dovrà seglierne prima il tipo e poi selezionare il punto della mappa in cui posizionarlo, togliendo dall'energia l'ammontare usato per fare apparire quel professore. Non potranno essere posizionati professori per cui non si ha l'energia necessaria e non potranno essere posizionati in caselle non disponibili. Ogni volta che viene ucciso uno studente l'energia aumenta e il punteggio della partita aumenta, mentre ogni volta che viene ucciso un professore l'energia diminiusce.
La partita si articola in ondate sequenziali di studenti ai quali resistere, ossia viene generato un certo numero di studenti da sconfiggere e una volta superata l'ondata ne viene generata un'altra e così via sino al termine della partita. Durante tutto il tempo il giocatore può predisporre i professori nella

maniera che ritiene più efficace.
Il campo di gioco è una griglia dotata di corsie; perciò, uno studente che avanza da destra verso sinistra avanzerà solamente lungo una linea retta e affronterà solo i professori presenti lungo quella linea, riducendone progressivamente la salute.
Una volta conclusa una partita l'utente può decidere se uscire dal gioco o iniziare una nuova partita.

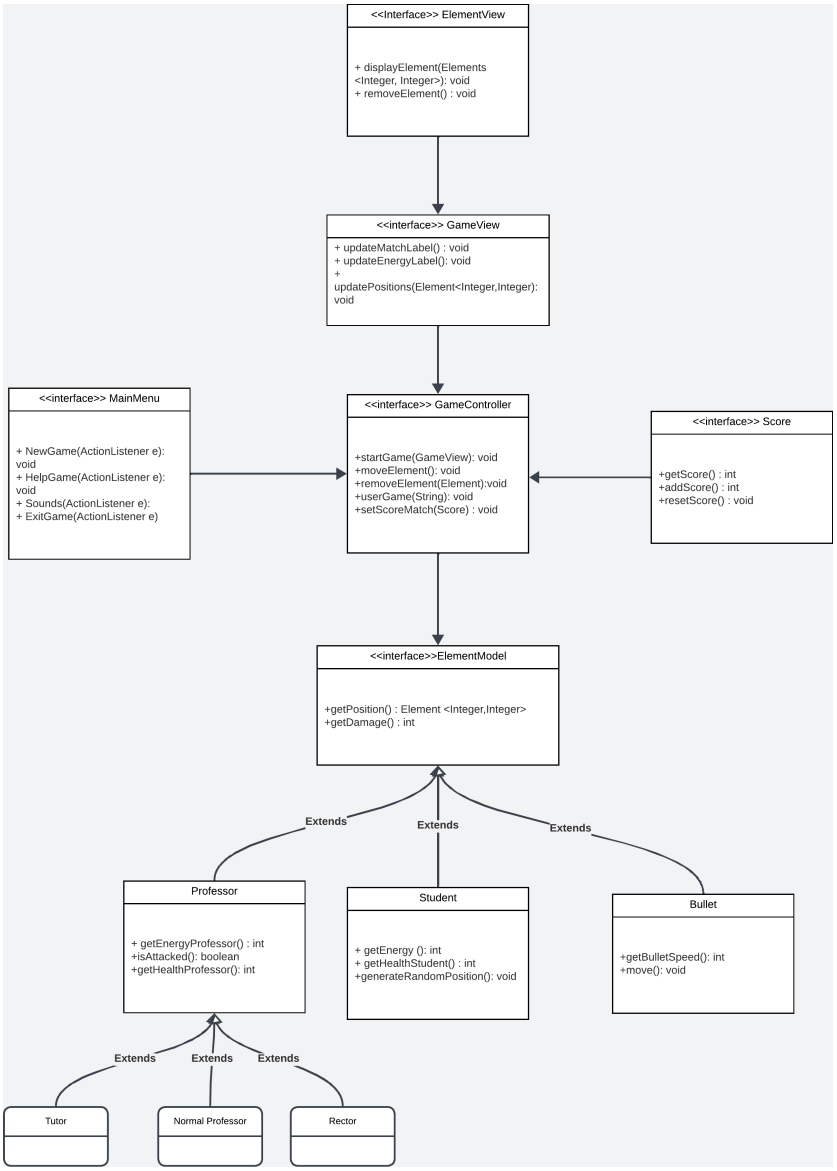

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le principali entità con le relative relazioni.

Capitolo 2

Design

2.1 Architettura

L'architettura di *Prof vs Student* segue il pattern architetturale *MVC*. Il pattern *MVC* consiste nel dividere la logica del codice in tre componenti: Model, View e Controller. Spiegazione breve di quello che è stato fatto per ogni componente:

- **View:** Usata per la presentazione grafica dei vari menù come il main menù e il menù di gioco utilizzando i file *fxml* per questi. Inoltre usata anche per i vari elementi del gioco come lo studente, i vari tipi di professori, il campo di gioco, i vari proiettili e lo stato del gioco.
- **Model:** Utilizzato per la gestione di tutti gli elementi di gioco come lo stato del gioco, soprattutto per sapere se l'utente ha vinto o ha perso, aggiornano anche l'energia e il punteggio del gioco. In aggiunta gestisce anche il diminuire del tempo e i diversi personaggi presenti nel campo come i professori, gli studenti e i proiettili in modo da gestire il loro comportamento e le interazioni tra di loro.
- **Controller:**Si occupa di gestire i vari input forniti dall'utente quando aggiunge un professore nel campo di gioco e tramite il model fa in modo che il professore si trovi nella posizione desiderata gestendo così l'utilizzo dell'energia. Oltre a ciò, il controller gestisce anche i vari input che l'utente esegue nei diversi menù quando si preme su un buttone. In questo modo si è in grado di capire se l'utente vuole creare un nuovo gioco, vuole sapere come giocare o uscire invocando così la view competente per ogni click sui diversi *button*.

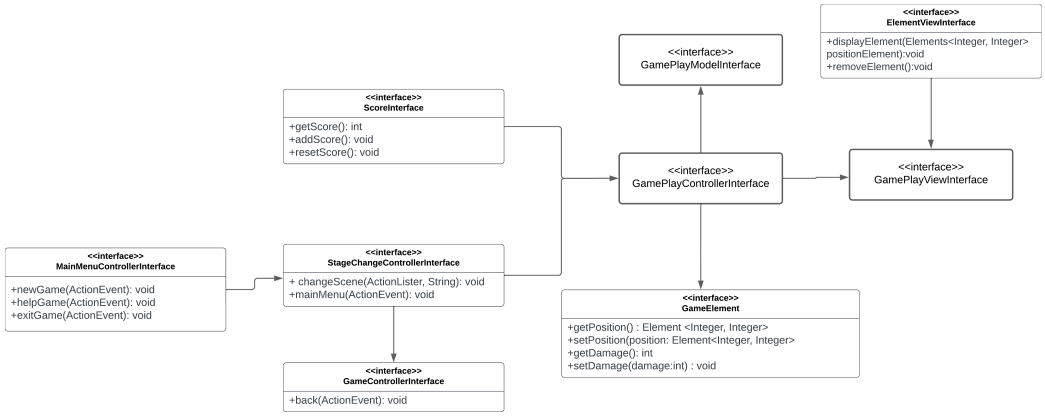

Figura 2.1: Rappresentazione UML generica dell'architettura del gioco

2.2 Design dettagliato

2.2.1 Alesja Delja

Gestione del punteggio del gioco.

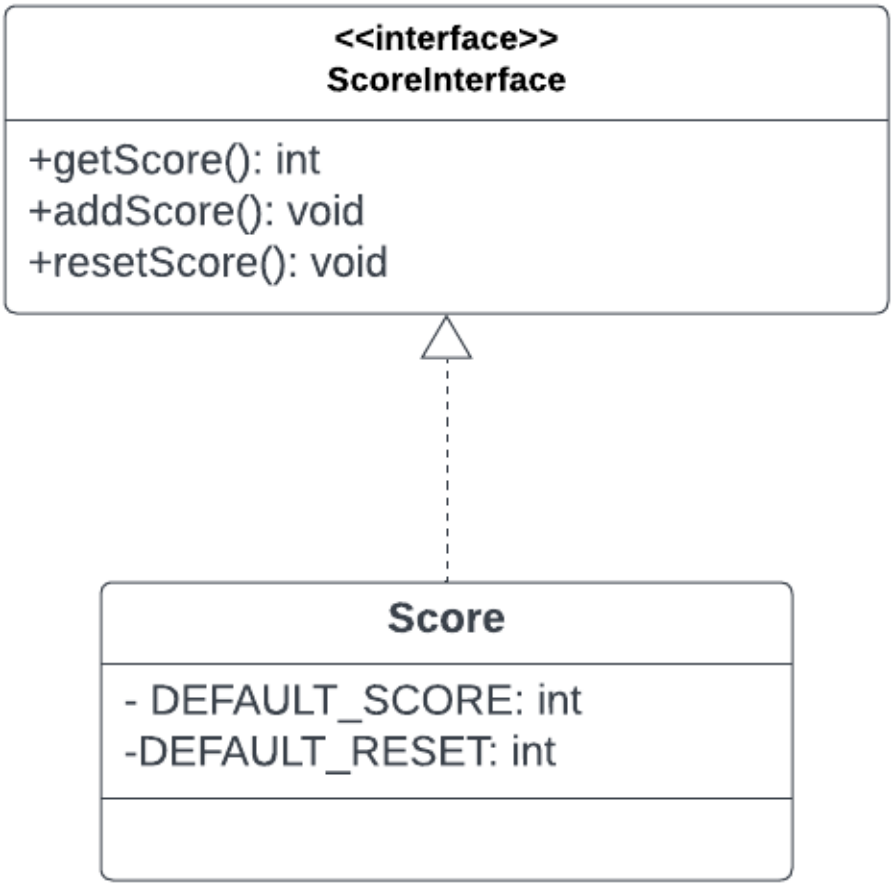

Figura 2.2: Rappresentazione UML della gestione del punteggio.

Problema Gestire in modo efficiente e strutturato il punteggio del gioco.

Soluzione Per la gestione del punteggio è stato deciso di realizzare un'interfaccia `ScoreInterface` che viene poi usato da `Score`. Queste due classi

permettono di restituire il punteggio corrente della partita aggiungere al punteggio attuale un *default score* usato quando uno studente muore. Per giunta viene gestito anche il reset del punteggio a un valore dato quando finisce la partita.

Navigazione tra le scene del gioco.

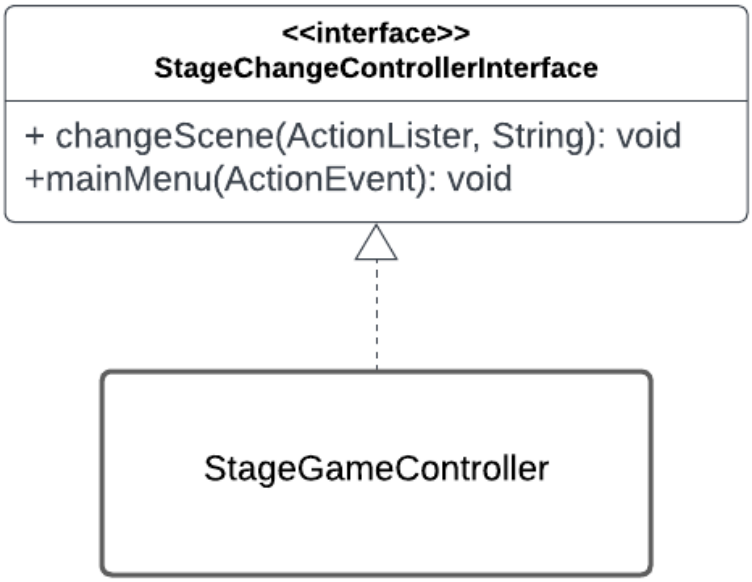

Figura 2.3: Rappresentazione UML sulla gestione della navigazione.

Problema Gestire la navigazione tra le diverse scene senza dover ripetere la logica della navigazione.

Soluzione Per gestire il cambio di scena ho creato una classe chiamata `StageGameController`, che implemeta la sua interfaccia, in modo da gestire la navigazione tra una scene e l'altra quando l'utente ne aveva bisogno.

Inoltre è stato implementato anche un metodo che gestisce il ritorno al menù principale in modo da non riscrivere lo stesso codice per tutti i *button* che ritornavano ad esso.

Gestione delle azioni utente nel menù principale.

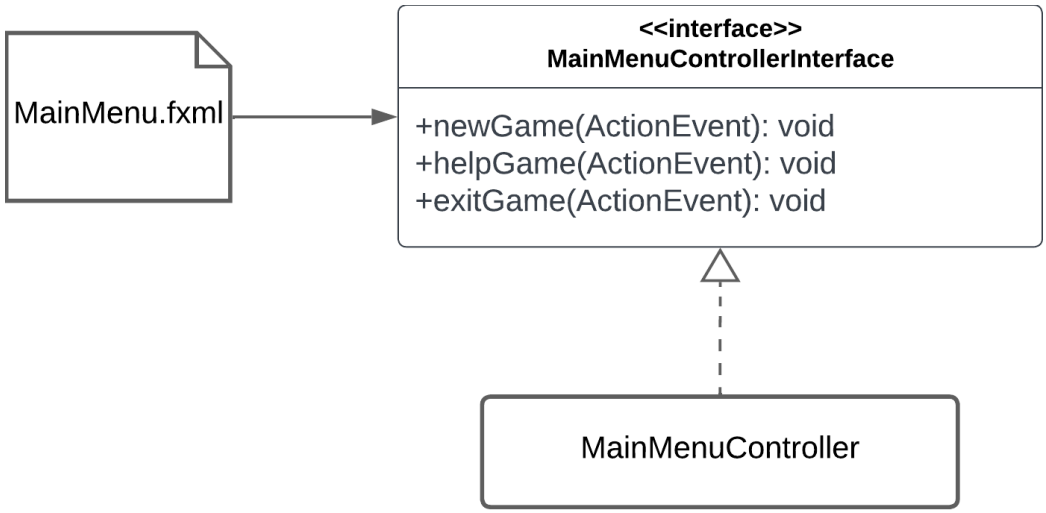

Figura 2.4: Rappresentazione UML della gestione delle azioni

Problema Neccesità di gestire le azioni dell'utente nel menù principale come l'inizio di un nuovo gioco, visualizzare le istruzioni e l'uscita dal gioco.

Soluzione Utilizzato un file di controller che viene usato dal file *fxml* che è la view del Main Menu. Per ogni button presente nel file *fxml* viene creato un metodo che gestisce cosa accade a seconda di quello premuto e cambia lo stage da quello corrente al nuovo utilizzando lo `StageChanegeController` prima spiegato. Ogni volta che l'utente vuole uscire gli viene chiesto se è sicuro tramite un alert.

Gestione delle azioni utente nel menù di aiuto.

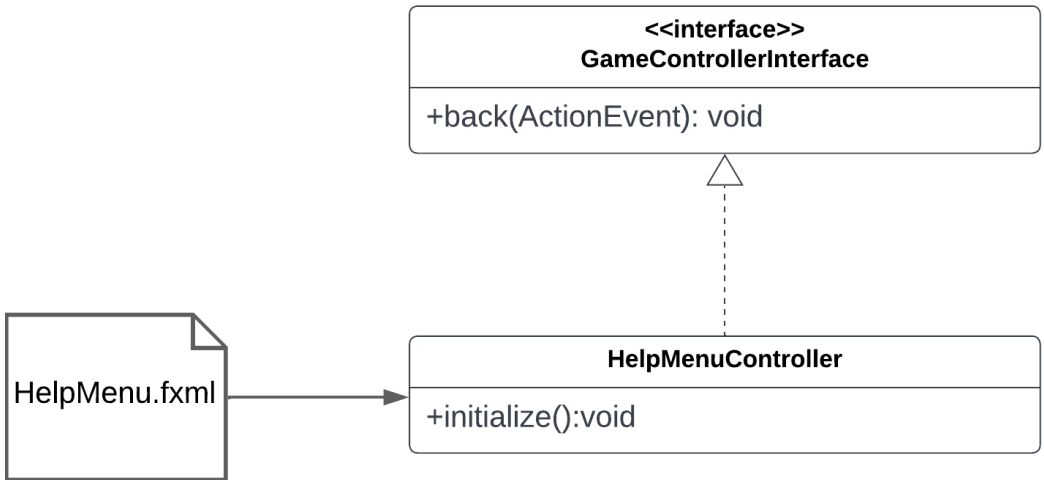

Figura 2.5: Rappresentazione UML della gestione delle azioni nel menu di aiuto

Problema Neccesità di gestire le azioni dell'utente nel menu di aiuto, facendogli vedere come si gioca e poi permettendogli di tornare al menù principale.

Soluzione Utilizzato un file di controller che viene usato dal file *fxml* che è la view del Help. Viene quindi visualizzato nel file *fxml* una label che spiega il funzionamento del gioco insieme a una legenda sugli studenti e sui diversi tipi di professori presenti nel gioco. Inoltre nel controller dopo aver istanziato il testo della label che avviene all'apertura della scene, viene usato anche `StageChangeController` per tornare al menu principale.

Implementazione dell'interfaccia del menù principale.

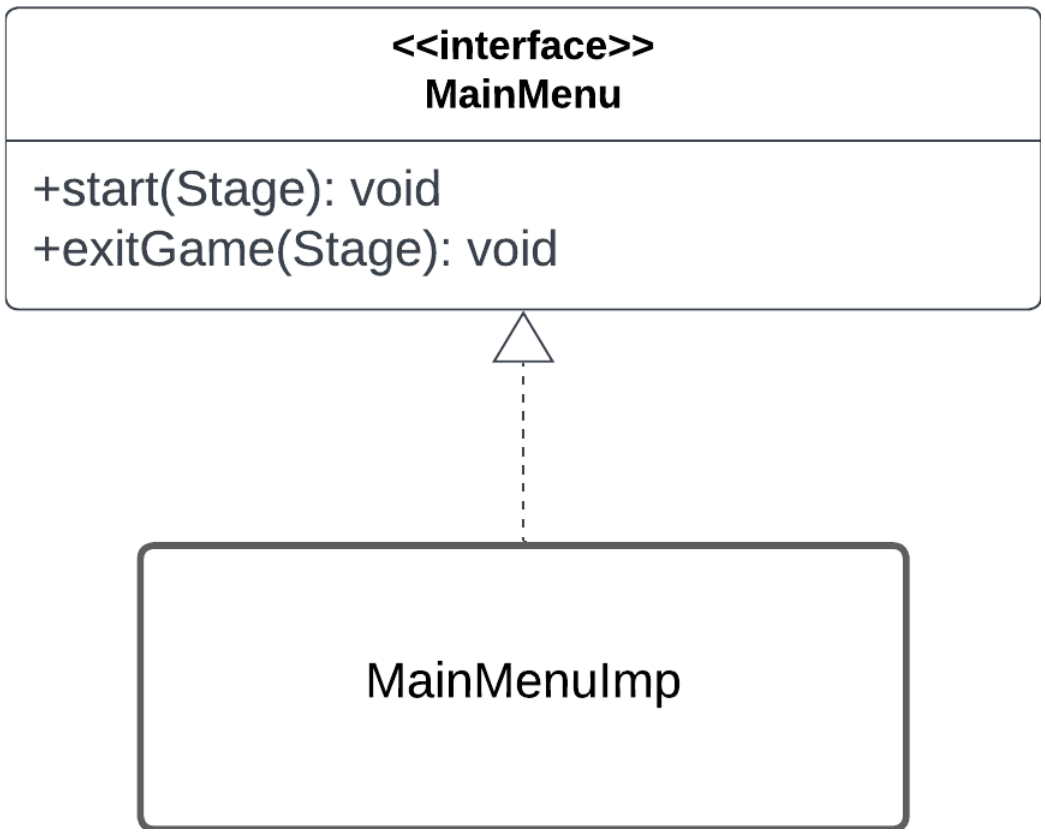

Figura 2.6: Rappresentazione UML dell'implementazione del menù principale.

Problema Neccesità di implementare l'interfaccia del menù pricipale che apppare all'avvio del gioco.

Soluzione Creata una classe che implementa l'inizio dell'applicazione del gioco in cui si richiama la view del menu principale tramite la chiamata del file *fxml* che lo contiene. Inoltre è stato aggiunto anche un metodo che se l'utente chiude la finestra di gioco senza fare click sul button, gli apparirà l'alert che veniva fuori anche dal click.

Rappresentazione degli elementi in gioco e degli studenti.

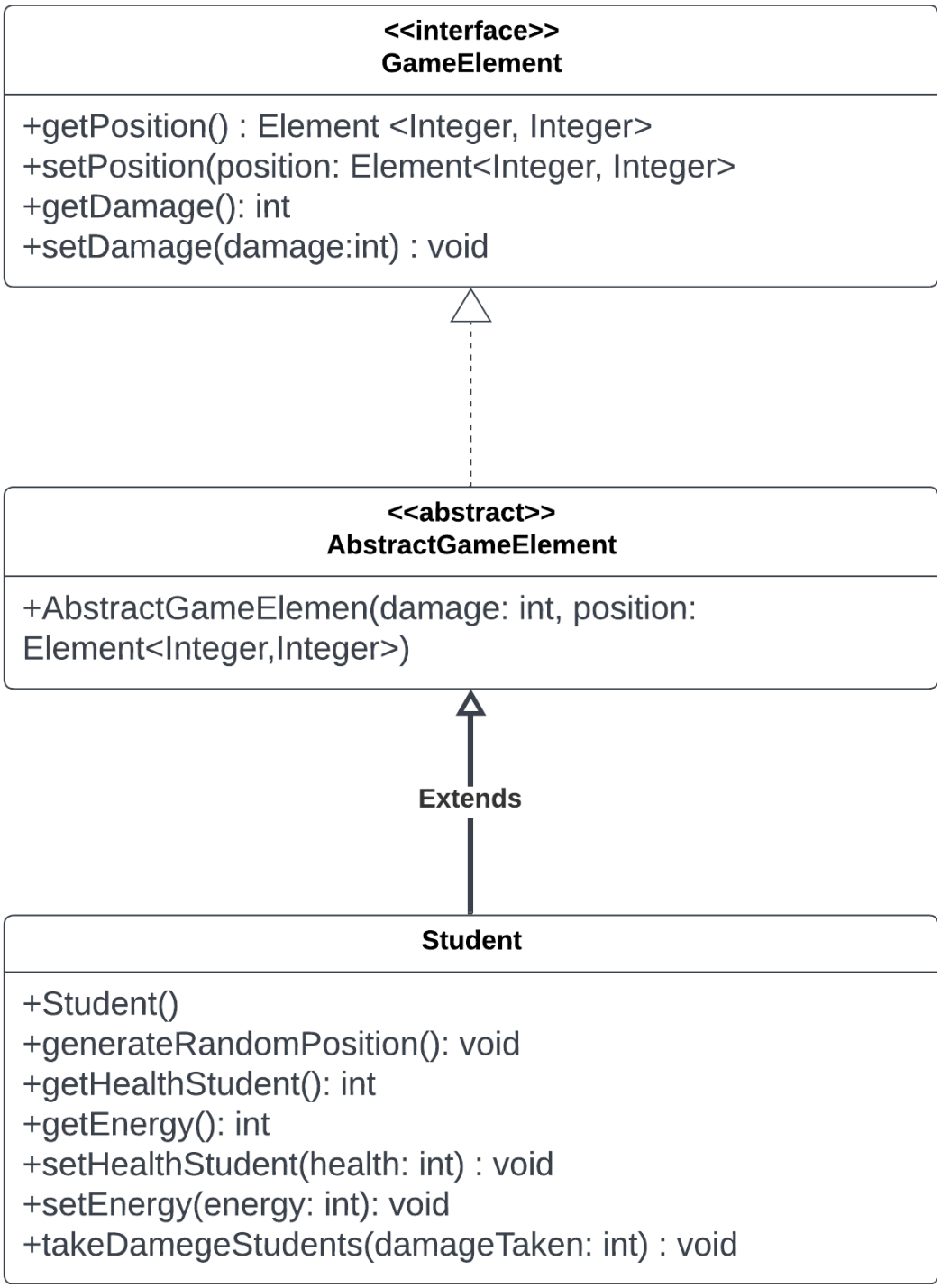

Figura 2.7: Rappresentazione UML degli elementi in gioco e degli studenti.

Problema Rappresentare gli elementi del gioco.

Soluzione Creata un'interfaccia e poi una classe astratta comune per gli altri elementi del gioco creando dei metodi comuni tra le diversi classi che lo estendono. In essa si gestisce la loro posizione e il danno causato dai diversi elementi e i diversi valori possono essere anche poi settati in caso di neccessità dagli altri elementi che la estendono.
Usato poi una classe che estende `AbstractGameElement` per creare l'elemento studente che gestisce la posizione randomica nel campo da gioco di esso e ne imposta un danno di default(25). Inoltre viene gestita anche la salute che ha ed aggiornandola quando viene colpito ed energia che si vince in gioco alla sua morte.

Gestione delle view degli elementi e dei studenti.

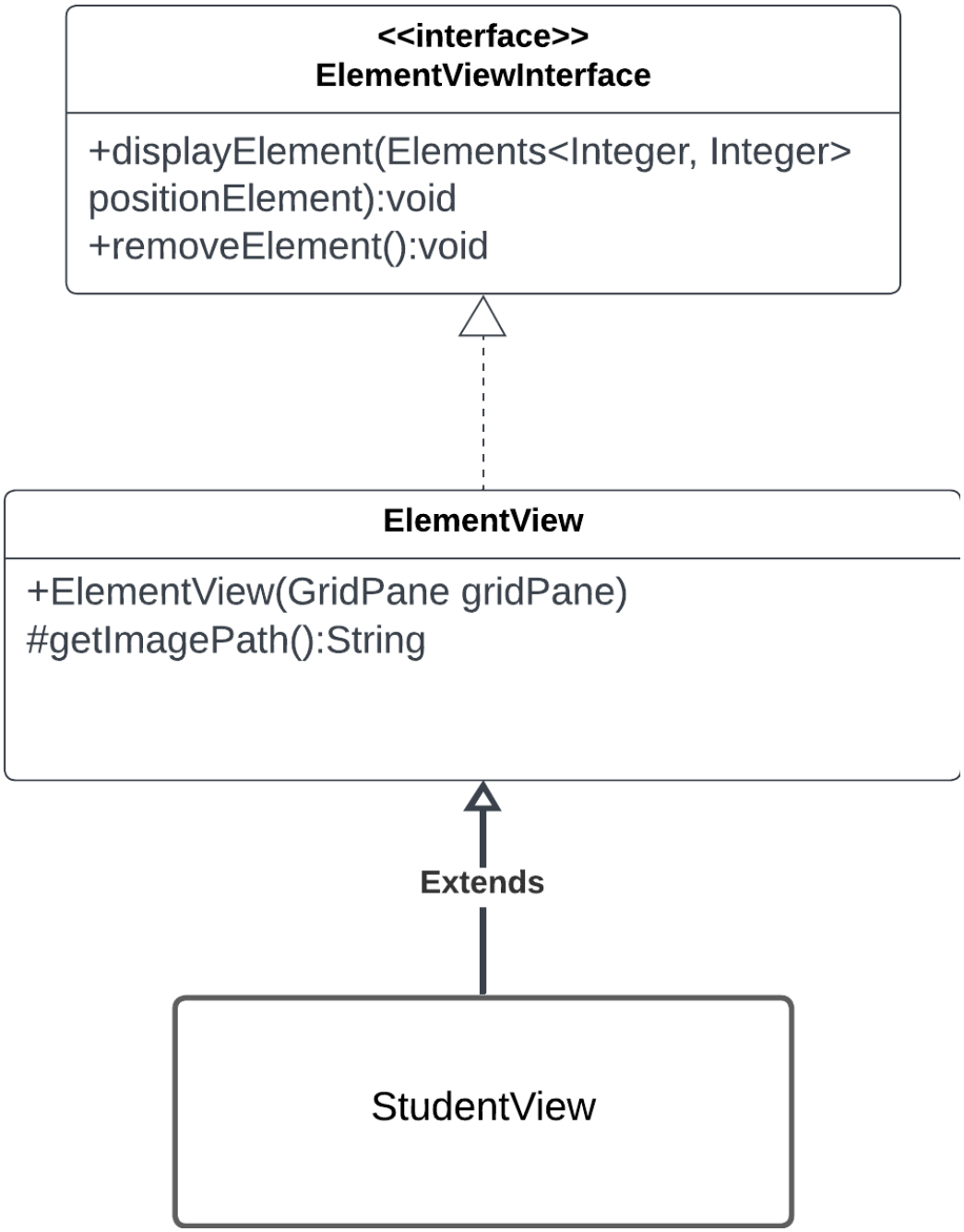

Figura 2.8: Gestione delle view degli elementi e dei studenti.

Problema Rappresentare gli elementi del gioco.

Soluzione Creato una interfaccia e poi una classe comune per gli altri elementi del gioco creando dei metodi comuni tra le diversi classi che lo estendono. In essa si gestisce la loro visualizzazione e la loro rimozione, inoltre viene anche gestito il path dell'immagine da utilizzare.
Usato poi una classe che estende `ElementView` per creare l'elemento studente che sovrascrive il path mettendo quello giusto per la visualizzazione.

2.2.2 Eleonora Falconi

Nel corso del progetto mi sono concentrata sulla creazione dei personaggi fondamentali del gioco, tra cui *Tutor*, *Professori* e il *Rettore*. Ho sviluppato le loro classi e interfacce, prendendomi cura della logica di gioco specificamente per i professori. Inoltre, ho implementato la classe principale e l'interfaccia grafica dei proiettili, che rappresentano l'elemento principale con cui i professori possono attaccare gli studenti.

Gestione dei professori

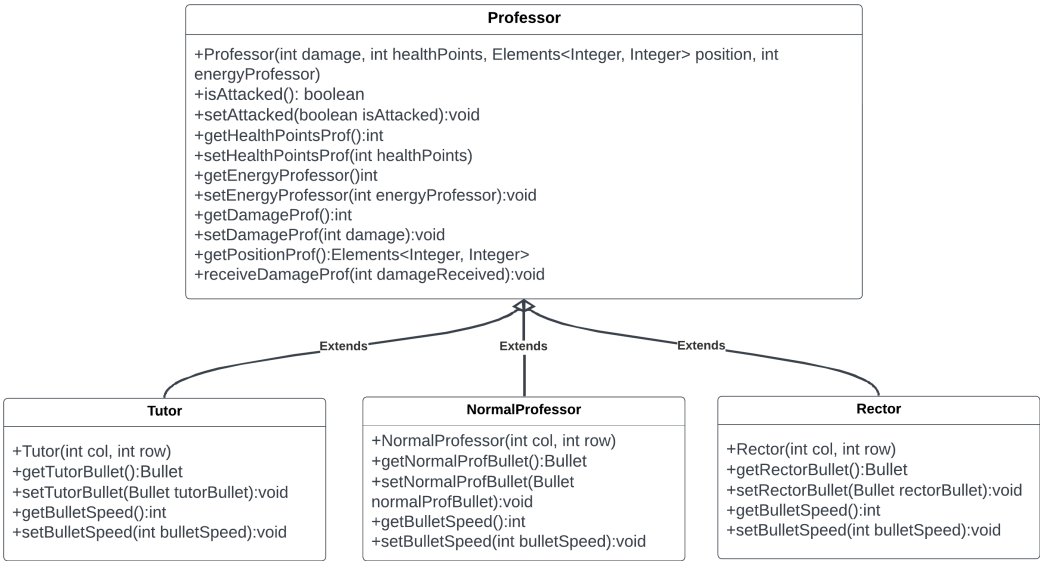

Figura 2.9: Rappresentazione UML del implementazione dei personaggi dei professori

Problema Gestione dei personaggi compresi nel corpo docenti

Soluzione Per risolvere il problema ho implementato una gerarchia di classi con la classe `Professor` come base e le sottoclassi `Tutor`, `NormalProfessor`

e `Rector`, i quali si differenziano per modalità di attacco, energia per poter essere scelti dall'utente e danno inflitto allo studente. La classe `Professor` è anche chiamata superclasse, è la classe principale che le sue sottoclassi estenderanno e possiede il metodo `isAttacked()` che servirà poi per gestire la collisione tra uno studente e il professore. Inoltre sono presenti i metodi `getter`getter e `setter` che impostano o ritornano i valori dei campi inizializzati. Poi passando alle sottoclassi, ad esempio il `tutor` può sparare solo davanti a sè, possiede in partenza 50 punti vita, infligge 25 punti di danno allo studente e per essere utilizzato dall'utente vengono sotratti 10 punti energia. Poi abbiamo `NormalProfessor` che possiede 100 punti vita, può causare un danno di 50 punti a uno studente e per essere acquistato servono 20 punti energia. Mentre il `Rector` può sparare in diagonale in modo random usando la funzione `shootDiagonal()`, possiede 150 punti vita, infligge un danno di 50 punti e vale 30 punti energia.

Gestione dell'interfaccia dei professori nel campo da gioco

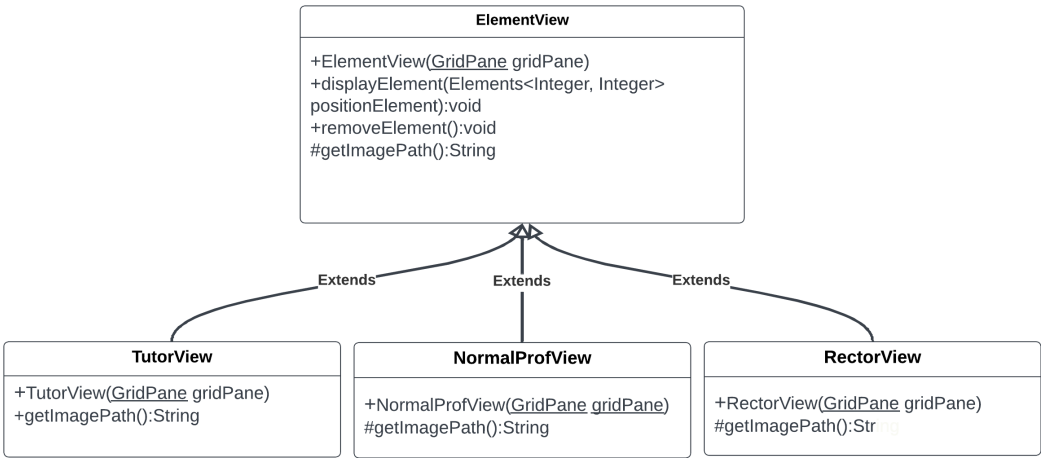

Figura 2.10: Rappresentazione UML dell'implementazione delle interfacce grafiche dei professori

Problema Rappresentare graficamente i personaggi della categoria "professore" nel gioco

Soluzione In questo contesto, abbiamo sviluppato diverse classi che gestiscono la visualizzazione di specifici tipi di elementi, come tutor, professori normali e rettori, all'interno dell'interfaccia grafica.

La classe base `ElementView` fornisce un'astrazione generica per la visualizzazione di tutti gli elementi di gioco. Questa classe definisce il metodo `displayElement`, che carica l'immagine corrispondente all'elemento da visualizzare e la posiziona all'interno di un `GridPane`. Inoltre, implementa il metodo `removeElement` per rimuovere l'elemento dalla visualizzazione quando non è più necessario.

Le classi derivate da `ElementView`, come `TutorView`, `NormalProfView` e `RectorView`, estendono la funzionalità di base per gestire la visualizzazione specifica di tutor, professori normali e rettori, rispettivamente. Ogni classe derivata sovrascrive il metodo `getImagePath` per specificare il percorso dell'immagine corrispondente all'elemento di gioco.

Gestione del metodo per il posizionamento del professore

Problema Gestire il mouseclick dell'utente per il posizionamento del professore nel campo da gioco

Soluzione Il metodo `handleMouseClick(MouseEvent event)` gestisce l'interazione del giocatore con la griglia di gioco attraverso il click del mouse. Quando il giocatore clicca su una cella della griglia, il metodo ottiene le coordinate della cella cliccata e le stampa a console. Successivamente, verifica se è stato selezionato un tipo di professore per essere posizionato sulla griglia. Se una delle tre tipologie di professore è stato selezionato e la cella cliccata è vuota, il metodo crea una nuova istanza del professore corrispondente e la posiziona nella cella. Inoltre, aggiunge il professore alla lista dei professori corrispondente e il proiettile del professore alla lista dei proiettili per essere poi gestito dal lato *Controller*. Infine, diminuisce l'energia totale del giocatore in base all'energia iniziale del professore inserito. Questa parte del codice è stata integrata nella classe della mia collega, Anna Bartolucci, che si è occupata della gestione della view per quanto riguardava il campo da gioco.

Gestione dei proiettili

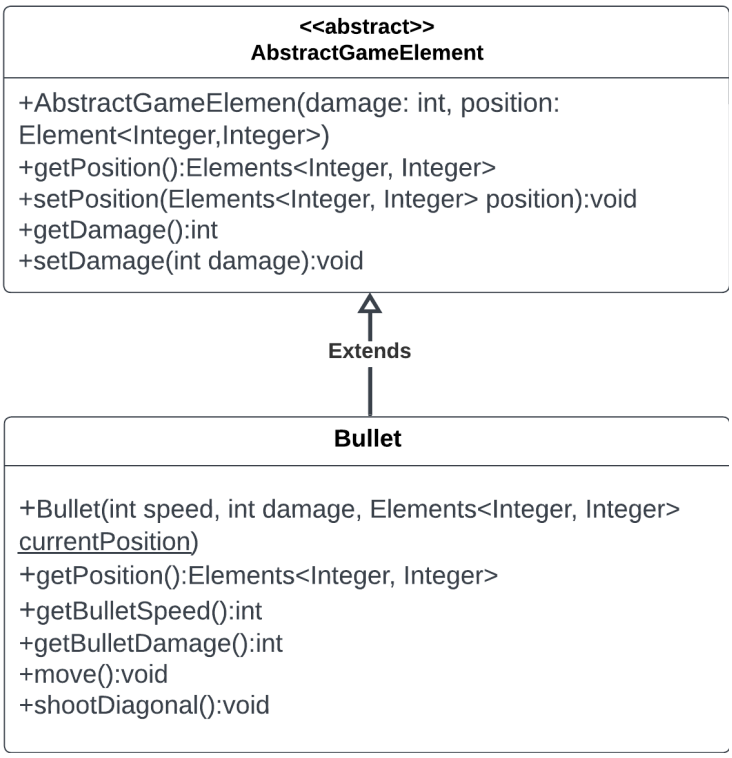

Figura 2.11: Rappresentazione UML dell'implementazione dei proiettili

Problema Rappresentare gli elementi del gioco, in particolare i proiettili

Soluzione La gestione dei proiettili è affidata alla classe `Bullet`, che estende una classe astratta comune chiamata `AbstractGameElement`. Questa classe astratta è progettata per gestire i metodi condivisi tra diverse classi che la estendono, come `getPosition()` o `getDamage()`, che restituiscono rispettivamente la posizione o il danno del proiettile, insieme ai rispettivi metodi setter per impostare tali valori.

Tra i metodi implementati per la gestione dell'attacco da parte dei professori verso gli studenti, si distinguono principalmente due:

- Il metodo `move()`, che consente di sparare frontalmente contro il nemico.
- Il metodo `shootDiagonal()`, che invece spara in diagonale in modo casuale, orientandosi sulla diagonale destra o sinistra.

Questi metodi offrono diverse modalità di attacco da parte dei professori durante il gioco, garantendo una varietà di strategie e comportamenti nell'interazione con gli studenti.

Implementazione dell'interfaccia grafica dei proiettili

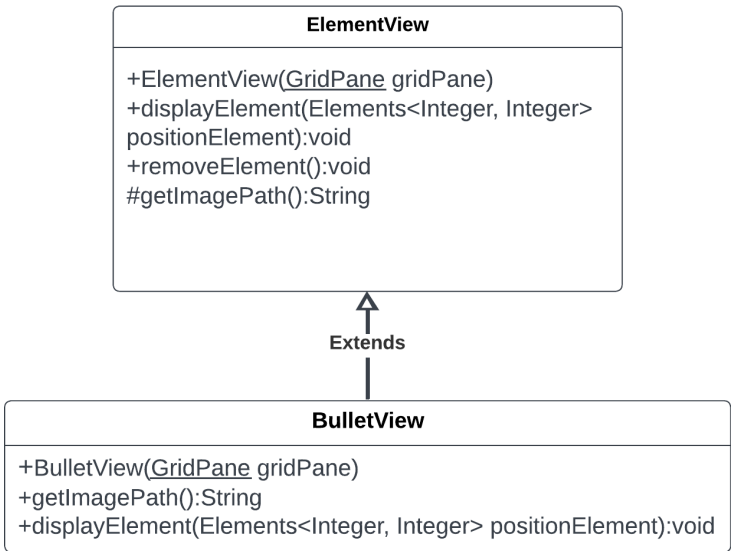

Figura 2.12: Rappresentazione UML dell'implementazione delle interfacce grafiche dei proiettili

Problema Gestione della rappresentazione grafica dei proiettili nel campo da gioco

Soluzione La gestione della visualizzazione degli elementi di gioco è un aspetto cruciale per garantire un'esperienza utente coinvolgente e intuitiva. In questo contesto abbiamo due classi fondamentali all'interno del nostro sistema e sono: `BulletView` e `ElementView`. Entrambe sono state progettate per consentire la corretta visualizzazione degli elementi di gioco all'interno dell'interfaccia grafica.

La classe `BulletView` estende la classe `ElementView` e si occupa specificamente della visualizzazione dei proiettili nel gioco. Utilizzando l'ereditarietà, `BulletView` eredita i metodi di visualizzazione di base da `ElementView` e ne aggiunge di specifici per la gestione degli oggetti. I proiettili che in modo visivo vengono rappresentati da piselli per mantenere qualche somiglianza con il gioco originale, insieme ad altri elementi del gioco, vengono visualizzati utilizzando questa classe di base.

Gestione degli elementi di gioco

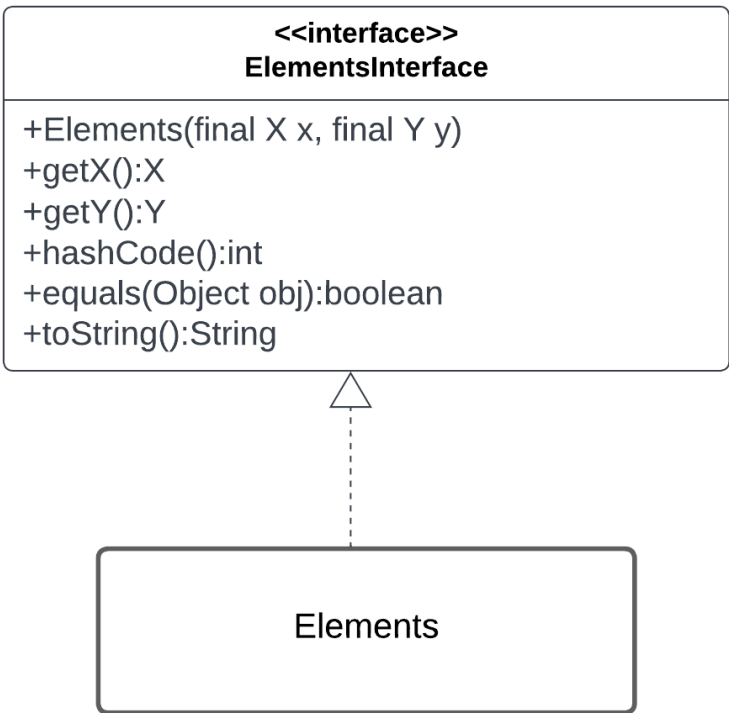

Figura 2.13: Rappresentazione UML dell'implementazione per la gestione degli elementi di gioco

Problema Gestire coppie di valori per le posizioni

Soluzione La classe `Elements` rappresenta una coppia di elementi generici, parametrizzata con due tipi generici `X` e `Y`. Essa contiene un costruttore che accetta due parametri di tipo `X` e `Y` e li assegna ai campi `x` e `y`, rispettivamente. Inoltre, fornisce i metodi `getX()` e `getY()` per ottenere i valori dei due elementi della coppia. La classe implementa anche i metodi standard `hashCode()`, `equals(Object obj)` e `toString()`, che consentono rispettivamente di calcolare il codice hash dell'oggetto, confrontare due oggetti per uguaglianza e ottenere una rappresentazione testuale dell'oggetto.

Essendo progettata in modo generico ed essendo un'astrazione, la classe `Elements` può essere utilizzata per rappresentare diverse coppie di elementi di tipi diversi, ad esempio le coordinate (X, Y) di una posizione nella griglia di un gioco.

Implementazione del metodo handlerprofessor

Problema Gestione dell'implementazione del professore durante il gioco

Soluzione Il metodo `handleProfessors()` si trova nel `GamePlayController` ed è una parte fondamentale del progetto che gestisce il comportamento dei professori nel gioco.

Innanzitutto, il metodo crea una copia delle liste di professori presenti nel gioco, questo è utile per evitare problemi di concorrenza durante l'iterazione e la modifica delle liste originali. Successivamente viene verificato se ci sono professori presenti nel gioco, se la lista non è vuota, il metodo itera attraverso ciascuna lista di professori.

All'interno del ciclo principale, per ogni professore presente nella lista, vengono eseguite le seguenti azioni:

1. Controlla se il professore ha esaurito i punti vita (`healthPointsProf`). Se i punti vita sono inferiori o uguali a zero allora il professore viene considerato sconfitto. In tal caso, il metodo aggiorna l'energia del gioco e rimuove il professore dalla lista dei professori da gestire.

2. Se il professore non è stato sconfitto e non è in collisione con gli studenti, il metodo controlla se è il momento per il professore di sparare. Se è il momento corretto per sparare e il professore non è già stato attaccato, il metodo aggiunge un proiettile alla lista dei proiettili del gioco.

2.2.3 Anna Bartolucci

La mia partecipazione al progetto si é concentrata soprattutto sullo sviluppo delle dinamiche di gioco. Ciò include la gestione delle collisioni tra gli elementi di gioco, il controllo dell'avvio e della durata delle partite, l'implementazione di meccanismi per aumentare l'energia e il punteggio totale, insieme alla gestione dell'uscita dalla partita.

Gestione Menù del Campo da Gioco

Problema Gestione uscita dalla partita prima della fine del tempo

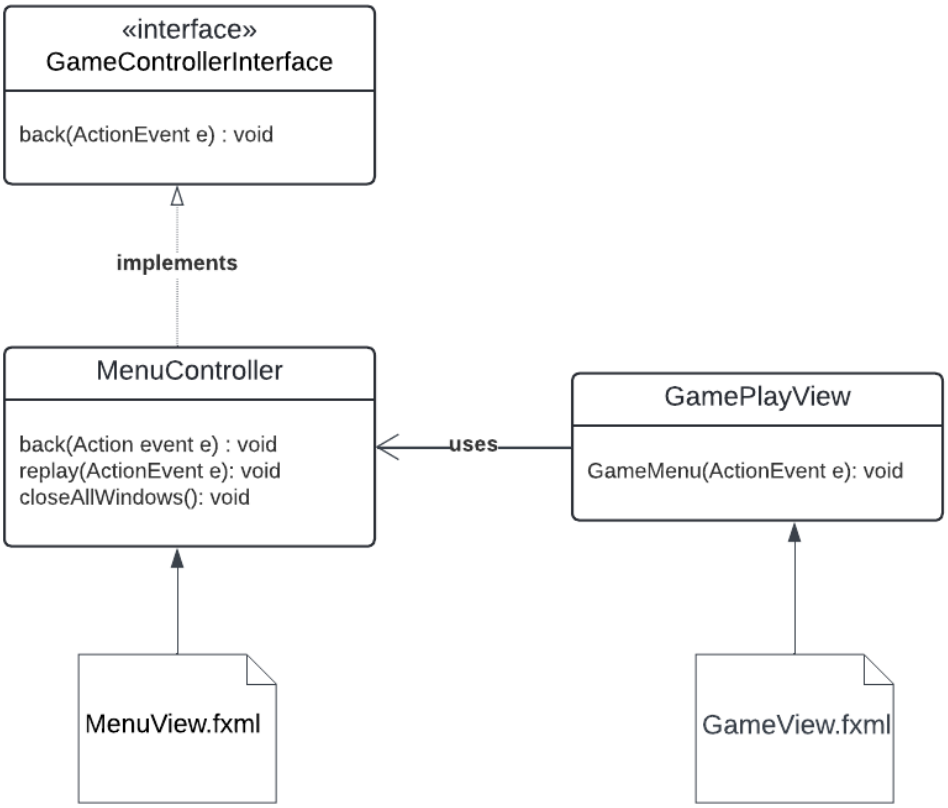

Figura 2.14: Rappresentazione UML dell'implementazione per la gestione del menù del campo di gioco

Soluzione La gestione del menù del campo di gioco consente agli utenti di poter interrompere il gioco prima della fine della partita e tornare alla schermata di menù principale o iniziare un nuovo match. Ho ritenuto opportuno quindi sviluppare la classe `MenuController`, che estende l'interfaccia `GameControllerInterface`. Il `MenuController` è stato associato al file FXML `MenuView.fxml`, il quale definisce l'aspetto grafico del menù di gioco attraverso due pulsanti, "Replay" e "Return to Main Menu". La gestione degli eventi dei pulsanti è implementata nei metodi `back(ActionEvent event)`, che gestisce il ritorno al menù principale, e `replay(ActionEvent event)`, che invece gestisce l'inizio di una nuova partita. Entrambi i metodi chiudono le finestre correnti e aprono nuove finestre in base all'azione richiesta dall'utente. Per richiamare il menù di gioco dal campo di gioco, è stata creata la funzione `GameMenu(MouseEvent event)` nel `GamePlayView`.

Loop di gioco

Problema Gestione sincronizzata del loop della partita utilizzando thread

Soluzione Per affrontare il problema della gestione del ciclo di gioco è stato utilizzato un approccio multi-threading. In questo caso la risoluzione del problema risiede nel metodo `startGame(GamePlayView gamePlayView)` nella classe `GamePlayController`. In questo metodo viene implementato il loop di gioco che gestisce in modo continuo le varie attività di gioco finché non viene raggiunta una condizione di fine gioco (*gameStatus=false*).
Durante il loop vengono avviati quattro thread che gestiscono le diverse dinamiche di gioco, tra cui il controllo di fine partita, l'aggiornamento delle posizioni dei proiettili e dei professori, il movimento degli studenti ed infine la creazione di nuove ondate di studenti.
Quando inizia la partita questi thread vengono avviati in modo asincrono e continuano ad eseguire il loro compito in loop finché lo stato del gioco rimane attivo (*gameStatus = true*). Nel thread principale viene costantemente verificato lo stato del gioco per consentire la terminazione ordinata degli altri thread una volta che la partita è conclusa. Per garantire un accesso sicuro e sincronizzato alle risorse condivise, ovvero le liste di elementi presenti sul campo da gioco, sono state utilizzate le istruzioni "synchronized", garantedo che l'accesso agli studenti, proiettili e professori avvenga in modo esclusivo ad un thread alla volta.

Gestione del campo da gioco

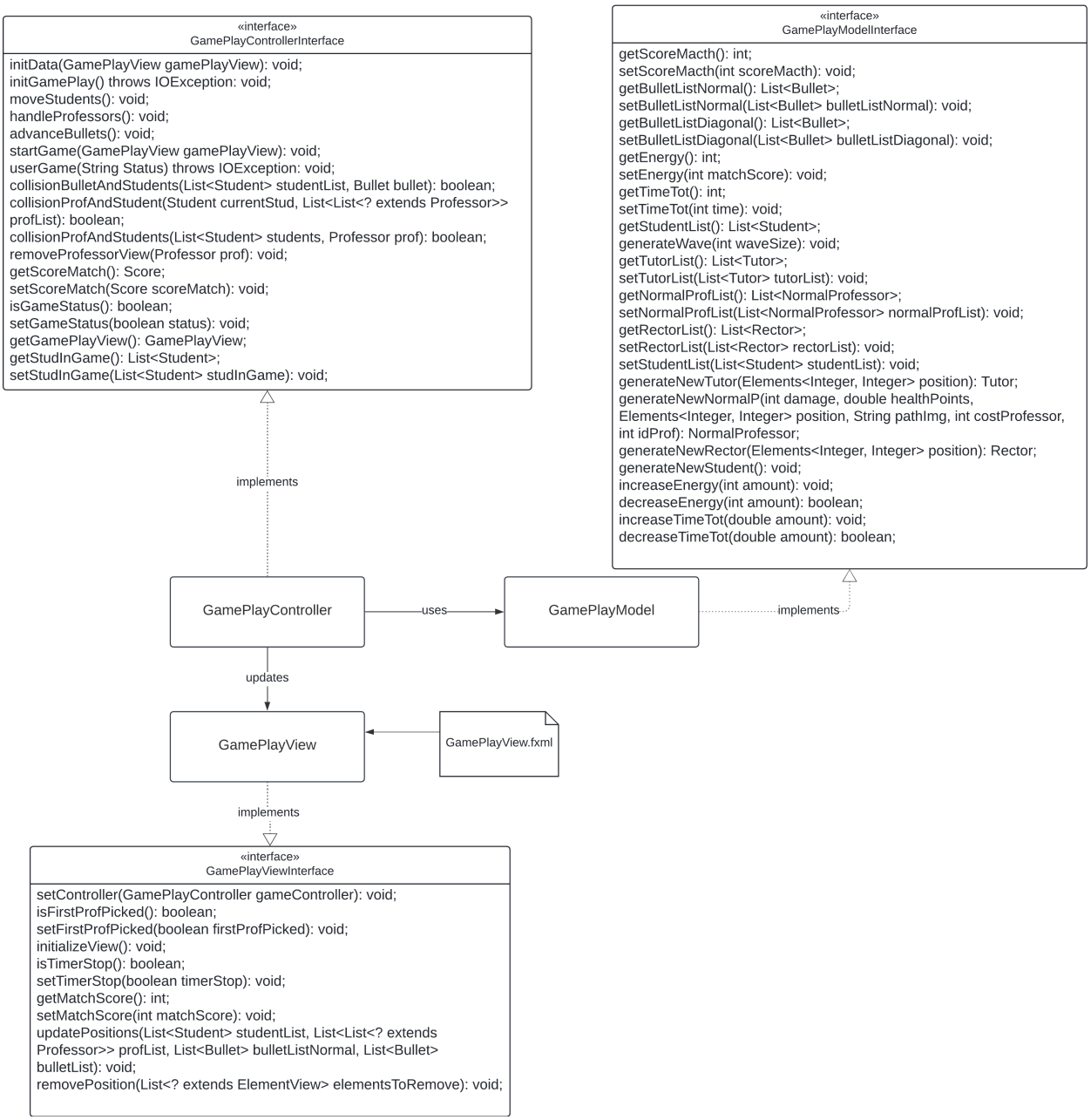

Figura 2.15: Rappresentazione UML dell'implementazione per la gestione del campo da gioco

Problema Gestione del campo da gioco

Soluzione Per gestire il campo da gioco, ho adottato un approccio basato sul pattern architetturale Model-View-Controller (MVC). Il Modello (Model), rappresentato dalla classe `GamePlayModel`, contiene la logica e i dati del gioco, gestendo il tempo di gioco, il punteggio, l'energia e le liste di professori, studenti e proiettili presenti sul campo. La Vista (View) consiste nel file FXML e nella classe `GamePlayView`. Questa interfaccia grafica consente di visualizzare le varie componenti del gioco grazie alle funzioni `initializeView()` e `updatePositions()`, che si occupano anche di aggiornare gli elementi del gioco sullo schermo.
Infine, il controller è gestito dalla classe `GamePlayController` che coordina le interazioni tra la Vista e il Modello del campo da gioco. Una parte significativa del codice riguarda la gestione delle collisioni tra gli elementi del gioco, cioè studenti, proiettili e professori. Le funzioni dedicate alla gestione delle collisioni (`collisionBulletAndStudents(List<Student> studentList, Bullet bullet)`, `collisionProfAndStudents(List<Student> students, Professor prof)`, `collisionProfAndStudent(Student currentStud, List<List<? extends Professor>> profList)`) si occupano di controllare se gli elementi interagiscono tra loro.

Avanzamento dei proiettili e degli studenti

Problema Gestione avanzamento dei proiettili e degli studenti sulla griglia da gioco

Soluzione Per risolvere il problema dell'avanzamento dei proiettili e degli studenti all'interno della griglia del gioco ho creato le funzioni `moveStudents()` e `advanceBullets()` dentro la classe `GamePlayController`. La prima è stata progettata per gestire e coordinare lo spostamento degli studenti lungo il percorso prestabilito sulla griglia ed a gestire un eventuale collisione con i professori. L'ultima funzione invece si occupa dell'avanzamento dei proiettili, gestendo anch'essa le eventuali collisioni che possono effettuarsi con gli studenti.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Tutti i test sono automatizzati, avendo utilizzato la suite JUnit. In particolare vengono fatti i seguenti test:

- **AbstractGameElementTest:** vengono testati la corretta funzionalità dei metodi per ottenere e impostare la posizione e i danni degli elementi di gioco.
- **ScoreTest:** vengono verificati se il punteggio iniziale è zero, se aumenta correttamente dopo l'aggiunta di punti e se viene resettato correttamente a zero.
- **StudentTest:**vengono verificati che i valori predefiniti per la salute e l'energia siano corretti e che sia possibile impostare e ottenere tali valori. Viene anche controllato che i valori predefiniti per danni, salute, energia e posizione dello studente siano corretti.
- **TutorTest:** vengono verificati che la gestione dei proiettili associati al tutor funzioni correttamente, inclusa la corretta impostazione e ottenimento dei proiettili e la gestione della velocità dei proiettili.
- **ProfessorTest:** vengono verificati il corretto funzionamento dei metodi che gestiscono lo stato del professore, la sua salute, energia, danni e posizione.
- **GamePlayModelTest:** viene verificato che i valori iniziali per il punteggio della partita, l'energia, il tempo totale e le liste di studenti, tutor, professori normali, rettori e proiettili siano correttamente impostati.

Successivamente, viene testata la gestione dell'energia e del tempo totale, sia nel caso di aumento che di diminuzione. Si controlla anche che non sia possibile diminuire l'energia o il tempo totale oltre il valore disponibile.
Infine, vengono testati i metodi per generare onde di studenti, nuovi studenti, tutor, professori normali e rettori, e viene verificato che le liste associate a ciascuna di queste entità vengano aggiornate correttamente.

3.2 Note di sviluppo

3.2.1 Alesja Delja

Utilizzo della libreria JavaFx, usata nelle classi componenti la view, soprattuto nel main menu

Utilizzata in vari punti un esempio è `https://github.com/Eleonora-f/OOP23-prof-vs-students/blob/084ccf3e5c6de9f33ef9955e6a6094c8feff115b/src/main/java/_OOP_develop_gradle/controller/StageChangeController.java#L17-L23`

Utilizzo di Lambda expressions

Permalink:`https://github.com/Eleonora-f/OOP23-prof-vs-students/blob/084ccf3e5c6de9f33ef9955e6a6094c8feff115b/src/main/java/_OOP_develop_gradle/model/MainMenuImp.java#L24-L28`

Utilizzo di Pair chiamato da noi Elements

Permalink:`https://github.com/Eleonora-f/OOP23-prof-vs-students/blob/084ccf3e5c6de9f33ef9955e6a6094c8feff115b/src/main/java/_OOP_develop_gradle/model/Student.java#L25-L29`

3.2.2 Eleonora Falconi

Utilizzo della libreria JavaFx

Utilizzata in alcuni punti. Un esempio é:`https://github.com/Eleonora-f/OOP23-prof-vs-students/blob/084ccf3e5c6de9f33ef9955e6a6094c8feff115b/src/main/java/_OOP_develop_gradle/view/NormalProfView.java#L15-L17`

Utilizzo di Pair chiamato da noi Elements

Utilizzato in alcuni punti. Un esempio é:https://github.com/Eleonora-f/OOP23-prof-vs-students/blob/391209750ac6d2484dd85e20c3937a4887c446a9/src/main/java/_OOP_develop_gradle/model/Bullet.java#L56

Utilizzo di Lambda expressions

Utilizzate in alcuni punti. Un esempio éhttps://github.com/Eleonora-f/OOP23-prof-vs-students/blob/084ccf3e5c6de9f33ef9955e6a6094c8feff115b/src/main/java/_OOP_develop_gradle/controller/GamePlayController.java#L205

Utilizzo di Stream

Permalink:https://github.com/Eleonora-f/OOP23-prof-vs-students/blob/084ccf3e5c6de9f33ef9955e6a6094c8feff115b/src/main/java/_OOP_develop_gradle/view/GamePlayView.java#L511

3.2.3 Anna Bartolucci

Utilizzo della libreria JavaFx

Utilizzata in vari punti. Un esempio éhttps://github.com/Eleonora-f/OOP23-prof-vs-students/blob/391209750ac6d2484dd85e20c3937a4887c446a9/src/main/java/_OOP_develop_gradle/controller/MenuController.java#L59-L73

Utilizzo di Lambda expressions

Utilizzate in alcuni punti. Un esempio é:https://github.com/Eleonora-f/OOP23-prof-vs-students/blob/391209750ac6d2484dd85e20c3937a4887c446a9/src/main/java/_OOP_develop_gradle/controller/GamePlayController.java#L230

Utilizzo di Stream

Utilizzati in alcuni punti. Un esempio é:https://github.com/Eleonora-f/OOP23-prof-vs-students/blob/391209750ac6d2484dd85e20c3937a4887c446a9/src/main/java/_OOP_develop_gradle/controller/GamePlayController.java#L432

Utilizzo di Optional

Utilizzati in alcuni punti. Un esempio é:https://github.com/Eleonora-f/OOP23-prof-vs-students/blob/391209750ac6d2484dd85e20c3937a4887c446a9/src/main/java/_OOP_develop_gradle/controller/GamePlayController.java#L432

Utilizzo di Thread

Utilizzati nel GamePlayController. Un esempio é:https://github.com/Eleonora-f/OOP23-prof-vs-students/blob/391209750ac6d2484dd85e20c3937a4887c446a9/src/main/java/_OOP_develop_gradle/controller/GamePlayController.java#L276-L285

Utilizzo di Pair chiamato da noi Elements

Permalink:https://github.com/Eleonora-f/OOP23-prof-vs-students/blob/391209750ac6d2484dd85e20c3937a4887c446a9/src/main/java/_OOP_develop_gradle/controller/GamePlayController.java#L118

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Alesja Delja

Sono abbastanza soddisfatta del lavoro svolto in gruppo e anche da me. Mi ritengono abbastanza soddisfata anche della parte che ho preso in carico dopo l'abbandono di uno dei nostri membri, anche se avrei voluto vedere meglio la grafica dei studenti e delle loro animazioni.
Ci sono state alcune scelte di design che rimpiango, per quanto riguarda il quadro generale, che a mio parere sono il risultato di una concentrazione scarsa da parte di tutti sulla parte preliminare di design e pre-structuring, soprattuto della gestione del gioco principale, le quali hanno poi impattato negativamente alcune parti del software, compresa la mia, e che hanno causato un continuo riadattamento di varie sezioni del mio codice.

4.1.2 Eleonora Falconi

Al termine di questo progetto posso ritenermi soddisfatta di quanto abbiamo realizzato. Poichè bisogna tener conto che, almeno per quanto mi riguarda era la prima volta che programmavo in questo linguaggio e con il paradigma ad oggetti. Premetto che la mia parte sicuramente poteva essere progettata meglio fin da subito, come anche le restanti parti del progetto, nonostante abbiamo comunque cercato di utilizzare il pattern *MVC(Model-View-Controller)*. Alla luce di ciò ci siamo trovate ad avere un'unica classe che gestisse tutta la parte `controller` del gioco. Sicuramente con l'esperienza acquisita dopo questo progetto, avrò una consapevolezza maggiore sul fatto di dover dedicare più tempo alla parte di progettazione iniziale, sia dal punto di vista del codice, ma anche per quanto concerne i pattern de-

sign, che certamente possono far apparire il codice meglio sviluppato e più professionale.

4.1.3 Anna Bartolucci

Complessivamente, ritengo soddisfacente il lavoro svolto da me e dalle mie colleghe. Questa esperienza, di fatto, mi ha aiutato a comprendere meglio ciò che implica lavorare su un progetto di dimensioni considerevoli all'interno di un team.
Riconosco che sono presenti carenze e lacune sia in generale nel progetto, ma anche nella parte da me sviluppata, dovute, a parer mio, dalla errata gestione del tempo e dalla poca esperienza. Sicuramente avrei potuto effettuare delle scelte di design migliori e più ottimali, ma la scarsa coordinazione e comunicazione iniziale ha penalizzato la fase preliminare di design e di pre-structuring, inducendomi a portare avanti un design non ideale.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Alesja Delja

Oltre alle difficoltà che si incontrano normalmente in un progetto, la mia difficoltà più grande è stato il lavoro fatto sullo studente dato che inizialmente era in carico al collega che ha poi abbandonato trovandomi così senza una base da cui partire.
Nonostante tutto però questa difficoltà è stata superata anche grazie all'aiuto delle altre colleghe del gruppo.

4.2.2 Anna Bartolucci

Concordo nel dire che l'abbandono da parte del nostro collega ha arrecato maggiore difficoltà a tutte noi, sopratutto perchè facevamo affidamento su di lui per certi aspetti iniziali che lui stesso ci aveva assicurato di occuparsene ma ha sempre rimandato.

4.2.3 Eleonora Falconi

Sicuramente portare avanti un progetto in tre piuttosto che in quattro come inizialmente pattuito è stata un'ulteriore difficoltà, oltre a quelle che si sono

presentate durante tutto lo sviluppo del codice. Sia perchè questo imprevisto ci ha portate ad allungare i tempi di consegna, sia perchè siamo state sovraccaricate di lavoro che non ci spettava. Ad ogni modo siamo comunque contente di aver terminato al meglio che potevamo.

Appendice A

Guida utente

La guida per l'utente si trova nel menu principale di gioco facendo click in "How To Play"

Bibliografia

Le immagini sono tutte state prese da google.

Per il file fxml del campo da gioco (GameView.fxml) si è presa ispirazione da:
`https://github.com/ria18405/Plants-Vs-Zombies/blob/master/src/sample/gamepage2.fxml`