Relazione
"Plants Vs Zombies"

Irene Sofia Lotti
Marco Marrelli
Margherita Zanchini
Sofia Caberletti

18 febbraio 2024

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il software mira alla costruzione di un demake di "Plants VS Zombies", un famoso videogioco tower defense del 2009, sviluppato da PopCap Games. Il giocatore deve difendere la propria casa tramite l'uso di piante con abilità difensive uniche, per sconfiggere le orde di zombies.

Requisiti funzionali

- Il gioco inizia mostrando la schermata di avvio, che presenta la possibilità di cominciare la partita.
- Iniziata la partita, viene mostrata l'area di gioco: questa consiste in una griglia in cui il giocatore può posizionare le proprie piante. A sinistra della griglia si trova la casa da proteggere, mentre da destra giungono gli Zombies.
- Gli Zombies sono le entità attive del gioco, che procedono lungo le righe della griglia con lo scopo di giungere alla casa, uccidendo le piante che incontrano. I Non-Morti vengono generati periodicamente e casualmente tra le righe disponibili.
- Le piante possono essere posizionate in ogni cella della griglia e non più spostate. Per essere posizionate, è necessario cliccare la targhetta corrispondente alla pianta scelta, per poi selezionare la cella desiderata. Una volta inserite, sparano agli Zombies presenti nella propria riga. Dopo un certo numero di colpi, lo Zombie verrà abbattuto ed eliminato dalla mappa di gioco.

- È possibile posizionare le piante grazie al "Sistema dei Soli". Ogni pianta ha un proprio costo e i Soli sono la moneta del gioco. Essi vengono generati periodicamente. Quando il Sole compare nella mappa di gioco, Il giocatore lo acquisisce cliccandoci sopra per inserirlo nel proprio deposito.
- Il Boss di fine livello consiste in un'orda di Zombies, ossia l'apparizione contemporanea di una moltitudine di Zombies.
- La vittoria si raggiunge sconfiggendo tutti gli Zombies. Invece, la sconfitta avviene nel caso in cui i nemici raggiungano la casa.

Requisiti funzionali opzionali

- Nella schermata di avvio, è fornita la possibilità di scelta di uno tra i livelli disponibili.

Requisiti non funzionali

- L'interfaccia utente del gioco deve essere intuitiva e facile da navigare, garantendo un'esperienza di gioco piacevole per gli utenti.
- Il gioco deve essere progettato per avere una grafica che si adatta a seconda del dispositivo.
- Il software deve garantire una discreta esperienza di gioco.

1.2 Analisi e modello del dominio

L'interfaccia Mondo mette a disposizione del giocatore (tramite una schermata iniziale) la possibilità di creare un nuovo Game, partendo da un Livello, utilizzato per settare i propri parametri. La partita di gioco, infatti, genera e gestisce le diverse entità che interagiscono tra di loro. Gli Zombie hanno il compito di attaccare ed uccidere le piante: nel momento in cui lo Zombie tocca la Pianta, la mangia in un breve numero di secondi e questa muore. Le Piante, invece, si attivano alla presenza di uno Zombie nella propria riga, sparando periodicamente proiettili. Una volta colpiti e uccisi gli Zombie, le piante sospendono la loro attività, attendendo l'arrivo di ulteriori nemici. L'ultima entità presente è il Sole, che si genera periodicamente nel campo di gioco e viene utilizzata come moneta del gioco per sbloccare il posizionamento delle Piante, in base al loro costo. Il finale di gioco è gestito dal Game, che

mostra una schermata di fine partita. Gli elementi costitutivi del problema sono sintetizzati in Figura 1.1.

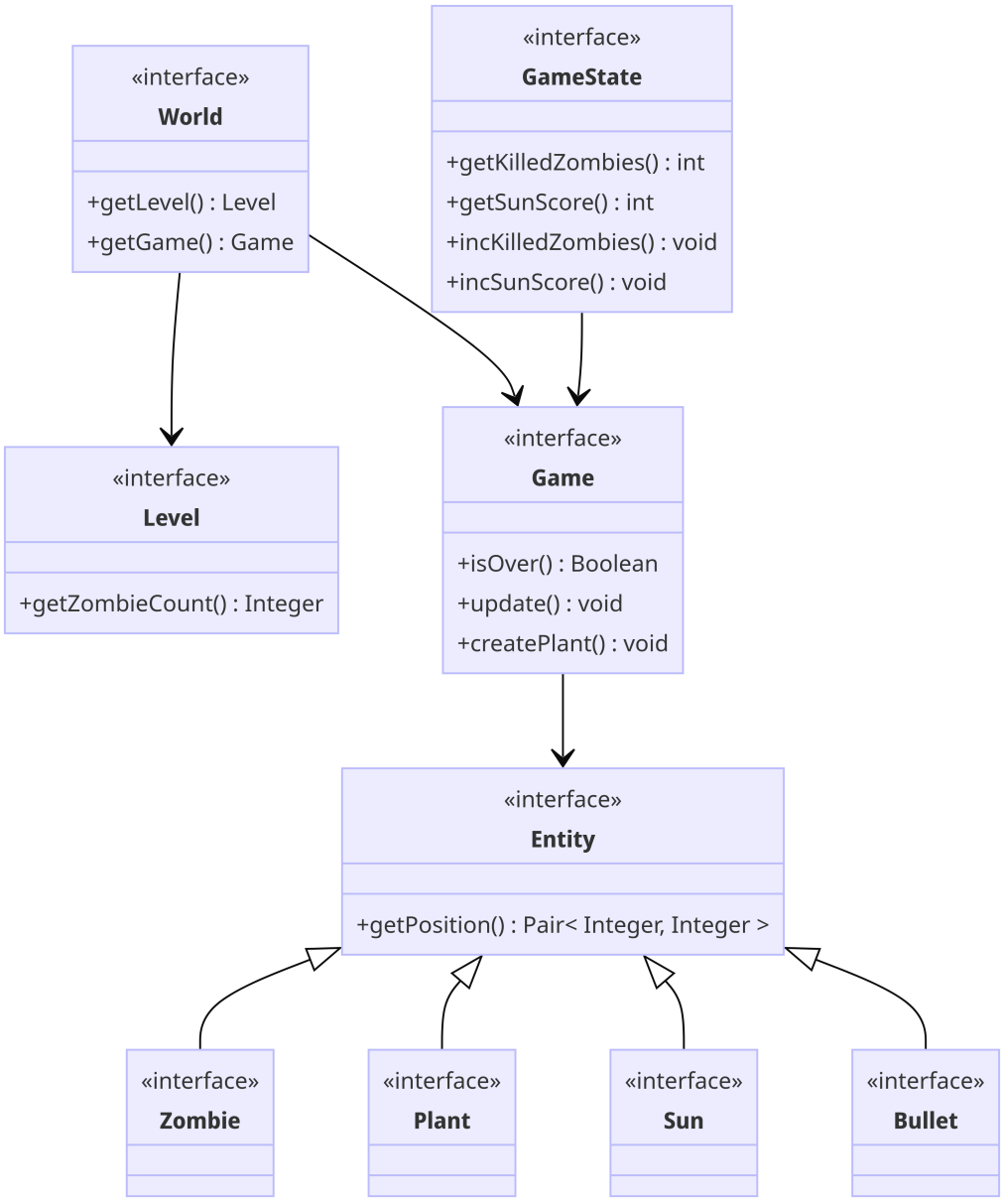

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Plants Vs Zombies adotta il pattern architetturale MVC. Il pattern separa l'applicazione in tre componenti logici (Model, View, Controller) interconnessi ma al tempo stesso indipendenti. I rispettivi moduli, nel dettaglio, rappresentano:

- **Controller**: il ruolo di questo componente, affidato ad un'omonima interfaccia, permette l'inizializzazione dell'applicazione. Inoltre, fa partire e gestisce il GameLoop, che consente l'aggiornamento di ogni entità e della sua corrispettiva rappresentazione. PvzStart è l'entry point del Controller.

- **Model**: il ruolo di questo componente è affidato all'interfaccia "Game", contenuto dentro l'interfaccia "World". GameImpl, implementazione di Game, è l'entry point del Model. Esso si occupa di gestire generazione e aggiornamento delle entità dal punto di vista logico.

- **View**: il ruolo della View viene delegato ad un'interfaccia con lo stesso nome. Tale interfaccia si occupa di mostrare graficamente le diverse entità e le schermate di gioco.

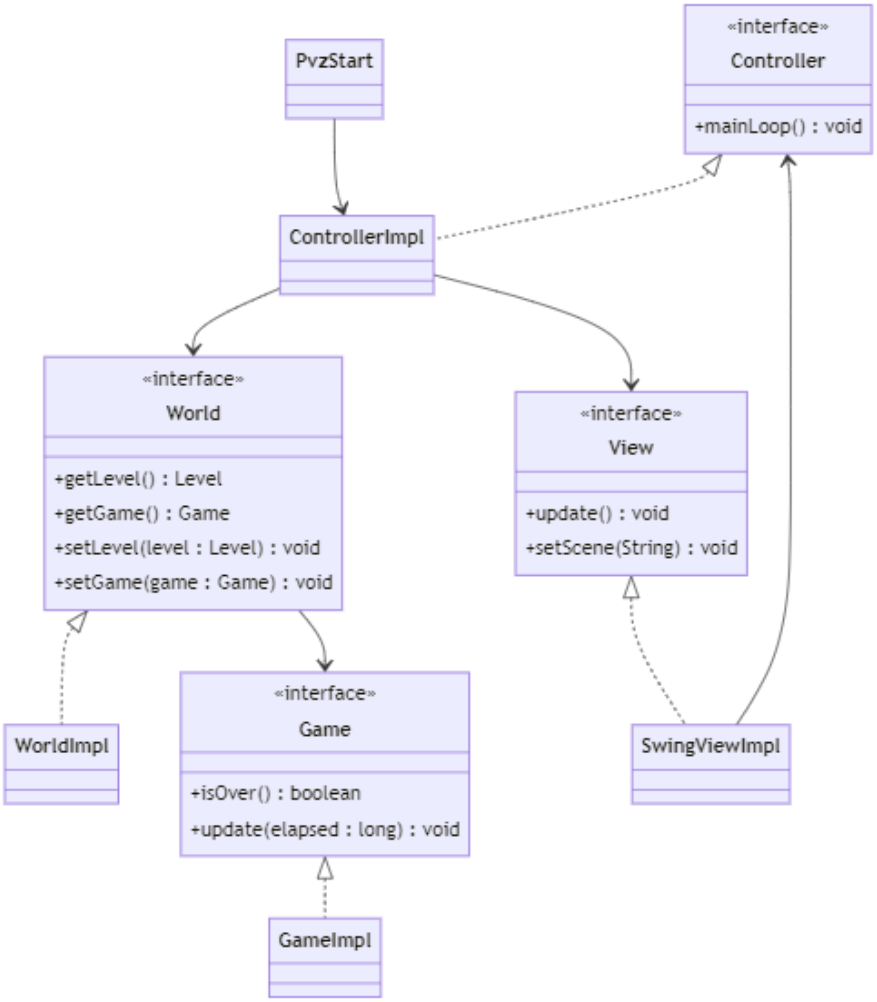

Figura 2.1: UML dell'architettura del software

2.2 Design dettagliato

2.2.1 Irene Sofia Lotti

Strutturazione dell'entità Zombie

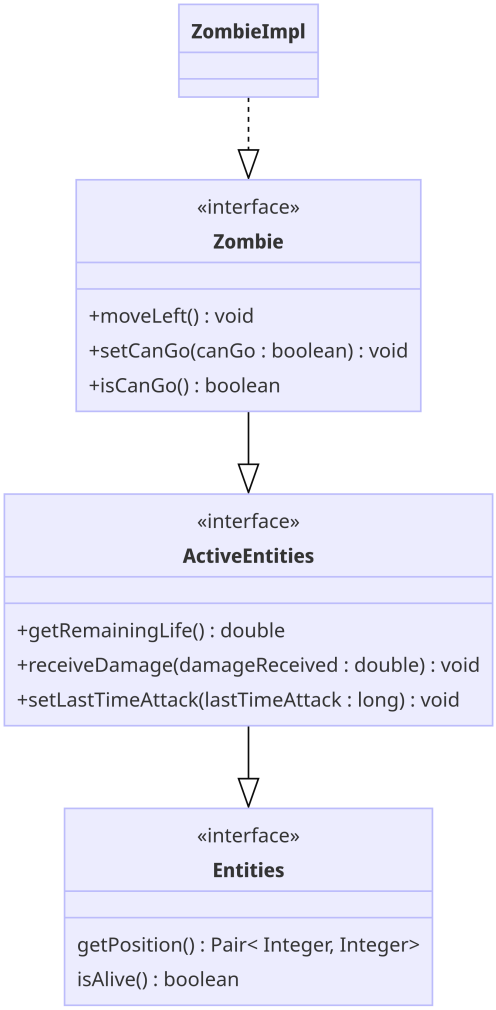

Figura 2.2: Schema UML della struttura degli Zombies

Problema: strutturazione delle classi che permettano di definire in modo conciso l'entità Zombie e altre interfacce (Soli, Piante e Bullet)

Soluzione: per la soluzione di questo problema abbiamo optato per l'ereditarietà: è stata definita una gerarchia di classi, con l'interfaccia "Entities" come radice, seguita da "ActiveEntities" e da "Zombie", per essere poi implementata da "ZombieImpl". Le funzionalità e i comportamenti specifici di

ciascuna entità sono incapsulati all'interno delle rispettive classi. Il tutto si è basato sull'applicazione del principio di buona progettazione DRY: "Don't Repeat Yourself". Lo scopo era quello di massimizzare la riduzione della duplicazione del codice, riducendo di conseguenza il rischio di errori e di bug. Il codice è di estrema facilità di comprensione. È stato rispettato, nel pieno delle sue caratteristiche, il principio KISS, "Keep It Simple, Stupid"

Creazione degli Zombie

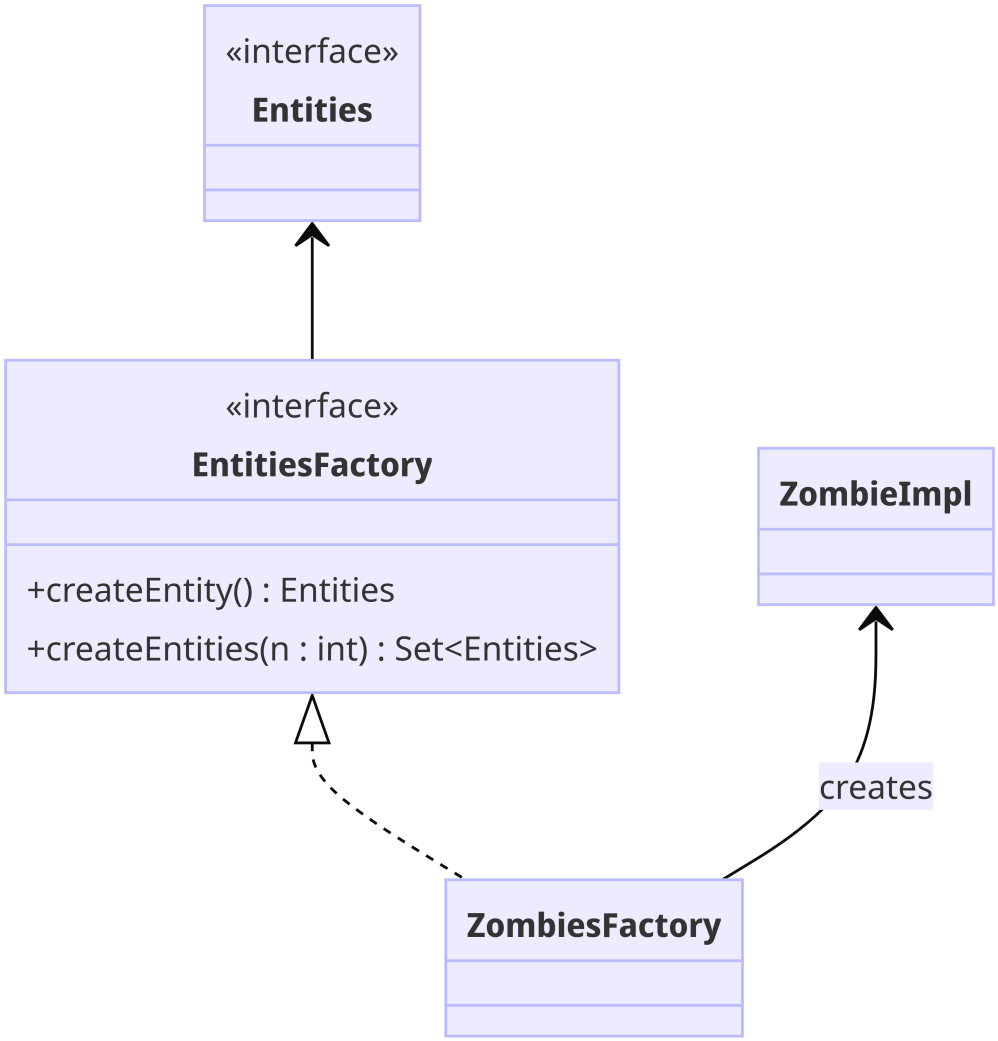

Figura 2.3: Schema UML della struttura della creazione degli Zombies

Problema: creazione delle entità di Zombie, con la possibiltà di creare più entità in una volta sola.

Soluzione: Per la creazione delle entità Sole e Zombie, è stato comunemente scelto e utilizzato il Factory-Method pattern, che consente di delegare la creazione di oggetti alle sottoclassi, consentendo una maggiore flessibilità nell'instanziazione di oggetti. Oltre a questo, abbiamo optato anche per il vero e proprio utilizzo di una forma di Strategy Pattern per la creazione di entità, dove le diverse implementazioni di EntitiesFactory rappresentano

strategie alternative per la creazione di entità. Nel mio caso, l'implementazione è ZombiesFactory. Tale classe viene richiamata dal Game per generare uno (nel caso della creazione di una singola entità durante il gioco) o più (nel caso della gestione delle orde di Zombie) entità. Tali scelte di pattern sono state fatte per rendere il codice più modulare e scalabile.

Gestione delle informazioni dello stato di gioco

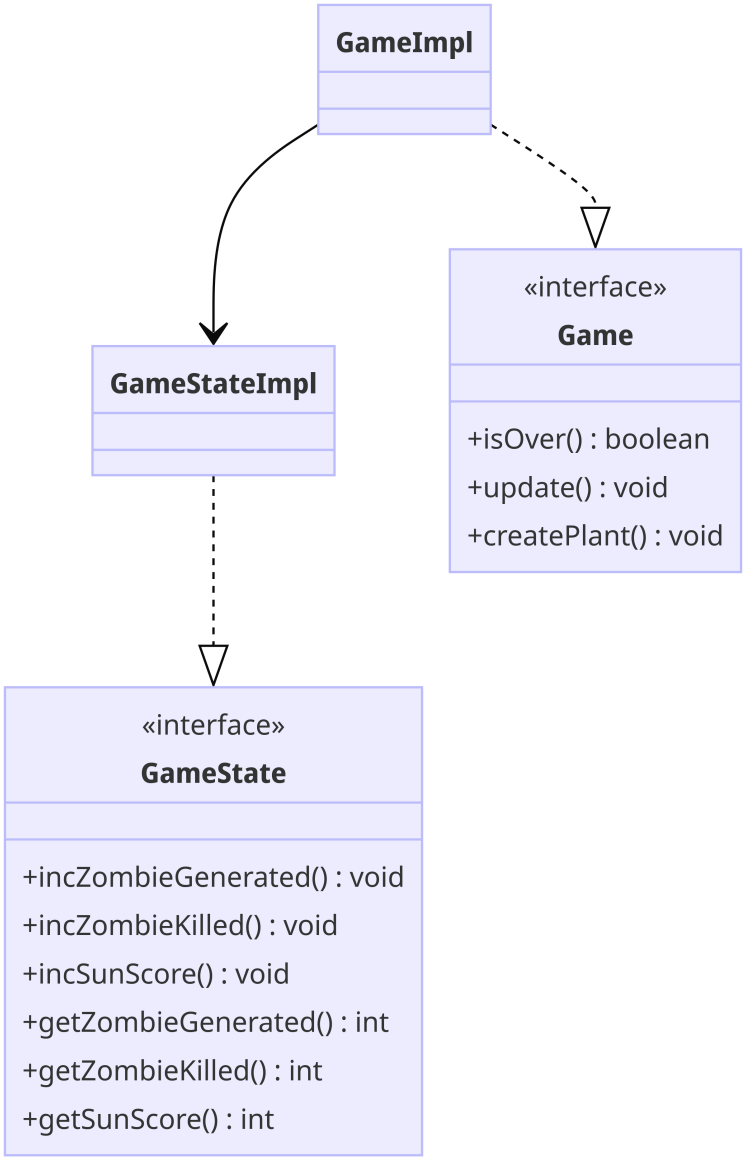

Figura 2.4: Schema UML della struttura delle informazioni di gioco

Problema: gestione dello stato di gioco
Soluzione: per gestire lo stato di gioco ho scelto di utilizzare il pattern State, il quale consente di definire una serie di stati e di modificare il comportamento dell'oggetto in base allo stato corrente. A questo proposito, GameState si occupa di tali meccaniche, quali l'incremento dei punteggi e

il riporto di essi. Il pattern usato è stato estremamente efficace, in quanto ha permesso di gestire lo stato di un oggetto in modo flessibile e modulare, consentendo di cambiare il comportamento dell'oggetto a seconda del suo stato interno.

Gestione del gameLoop

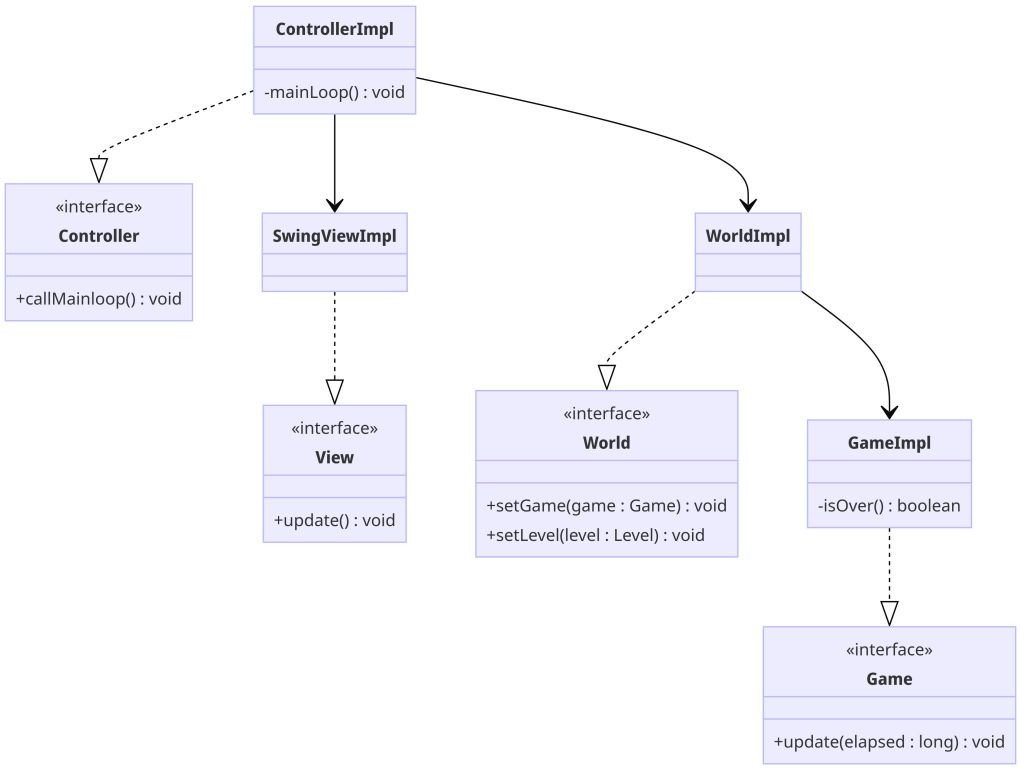

Figura 2.5: Schema UML del mainloop all'interno del Controller

Problema: Implementazione del Ciclo di Gioco che permetta di aggiornare lo stato di gioco in base

Soluzione: Per la creazione del ciclo di gioco si è scelto di realizzare una forma semplificata del pattern Game Loop, prendendo come ispirazione il codice di "Game As A Lab" del professor Alessandro Ricci, ma adattandolo e reimpostandolo a seconda delle esigenze della nostra applicazione. Proprio come il classico pattern Game Loop, la funzione aggiorna lo stato di gioco in base ai fattori esterni e del mondo. Una volta aggiornato lo stato di gioco, il Game Loop procede al rendering dei frame sullo schermo. Infine viene regolato il ritmo di gioco, per assicurarci che venga mantenuta una frequenza di aggiornamento costante. Il codice l'ho pensato mirando al conseguimento di una migliore esperienza di gioco, favorendo inoltre manutenibilità, la flessibilità e le prestazioni dell'applicazione.

Adattare le immagini in base alla scalabilità

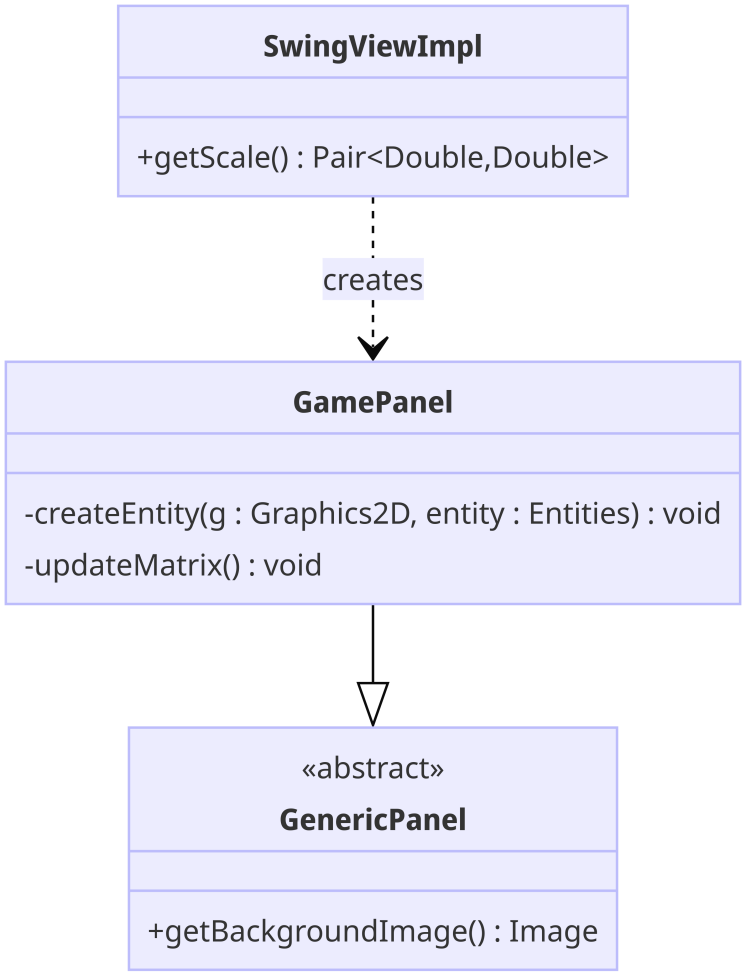

Figura 2.6: Schema UML dell'aggiornamento di matrice e entità in base alla scalabilità

Problema: aggiornare la dimensione grafica delle entità e delle celle all'interno della matrice.

Soluzione: la gestione della scalabilità delle entità e delle celle della matrice nel componente avviene nel "GamePanel", utilizzando costanti per la scalatura. La logica di aggiornamento delle dimensioni è stata implementata controllando il parametro Scale preso da "SwingViewImpl". Se la scala cambia, viene aggiornata la dimensione delle entità e delle celle. Durante lo sviluppo, abbiamo seguito principi come il DRY ("Don't Repeat Yourself"), la chiarezza del codice e la separazione delle responsabilità. Questo approc-

cio ha garantito una resa grafica ottimale su schermi di diverse dimensioni, migliorando l'esperienza di gioco complessiva.

2.2.2 Marco Marrelli

Strutturazione della Creazione e Caricamento di un Livello

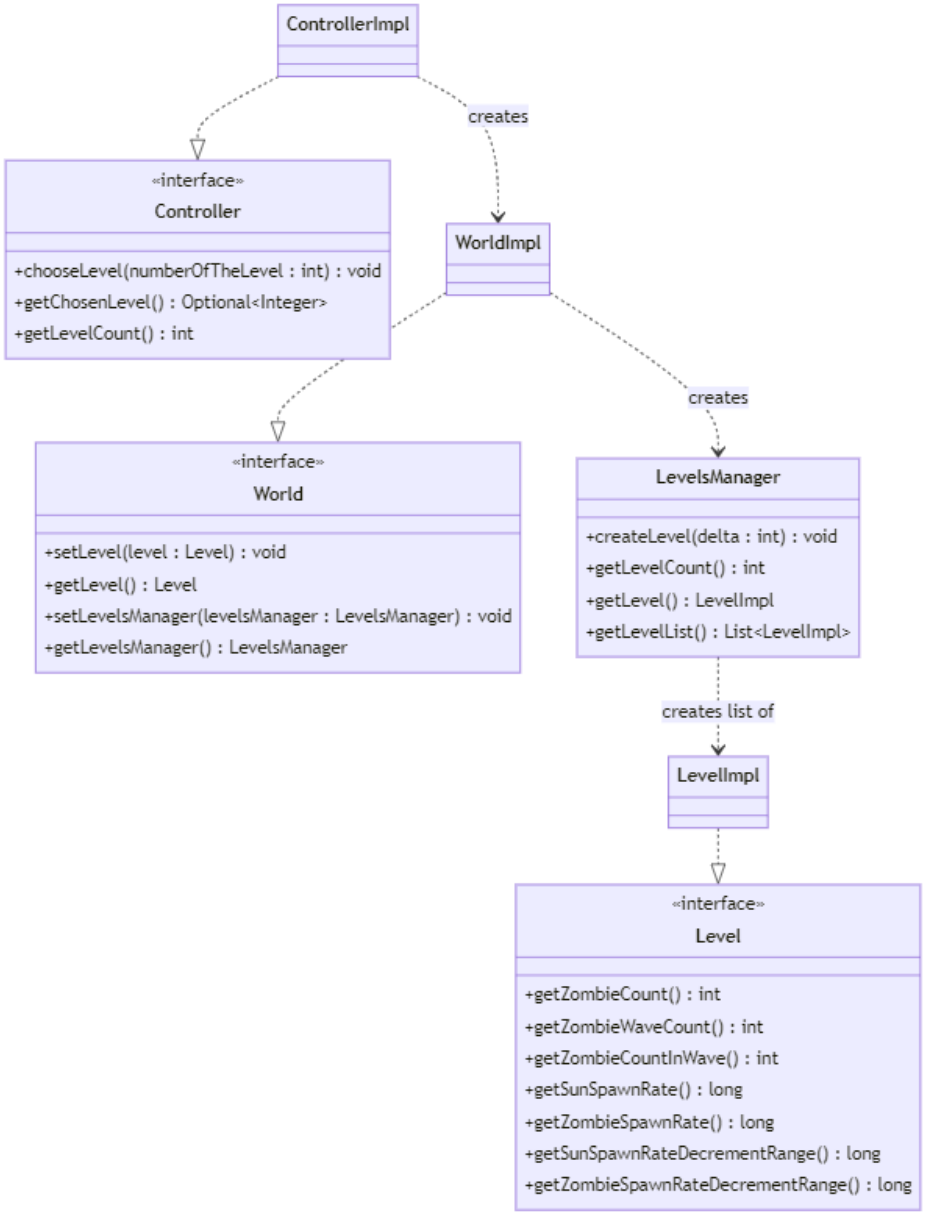

Figura 2.7: UML della struttura della gestione dei livelli

Problema: L'applicazione deve creare un certo numero di livelli, di difficoltà crescente, dando poi all'utente la possibilità di selezionarne uno solamente, per infine caricarlo nell'interfaccia di gioco.

Soluzione: La classe LevelsManager crea una lista di livelli, di difficoltà crescente, e restituisce essa totalmente o un elemento singolo (Level) dato un index. Nel nostro caso, verrà restituito (tramite un input click-event dell'utente, in un'interfaccia dedicata) solo un livello al World, che lo setterà nel Game tramite la funzione setLevel(Level level).

Strutturazione della Gestione delle Scene

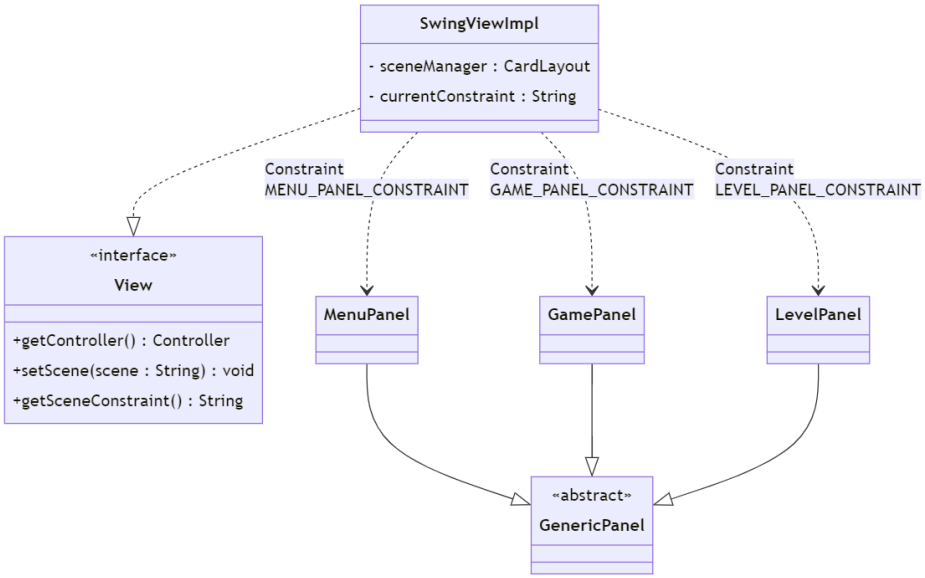

Figura 2.8: UML della struttura della gestione delle scene (panels)

Problema: L'applicazione deve gestire più interfacce grafiche, ognuna avente uno scopo ben preciso. Le principali scene che dovranno essere rappresentate all'interno dell'applicazione saranno:

- Menù iniziale
- Schermata per la selezione del Livello
- Schermata di Gioco

Soluzione: Tramite l'utilizzo del design pattern Façade, viene creato un menù iniziale (l'entry point dell'utente), dove da lì potrà selezionare le varie interfacce grafiche (selezione livelli e pannello di gioco) o eseguire l'azione di chiusura dell'applicazione. Ogni scena ha una constraint testuale identificativa, che verrà passata alla View, tramite la funzione setScene(String scene).

Strutturazione dell'evento GameOver

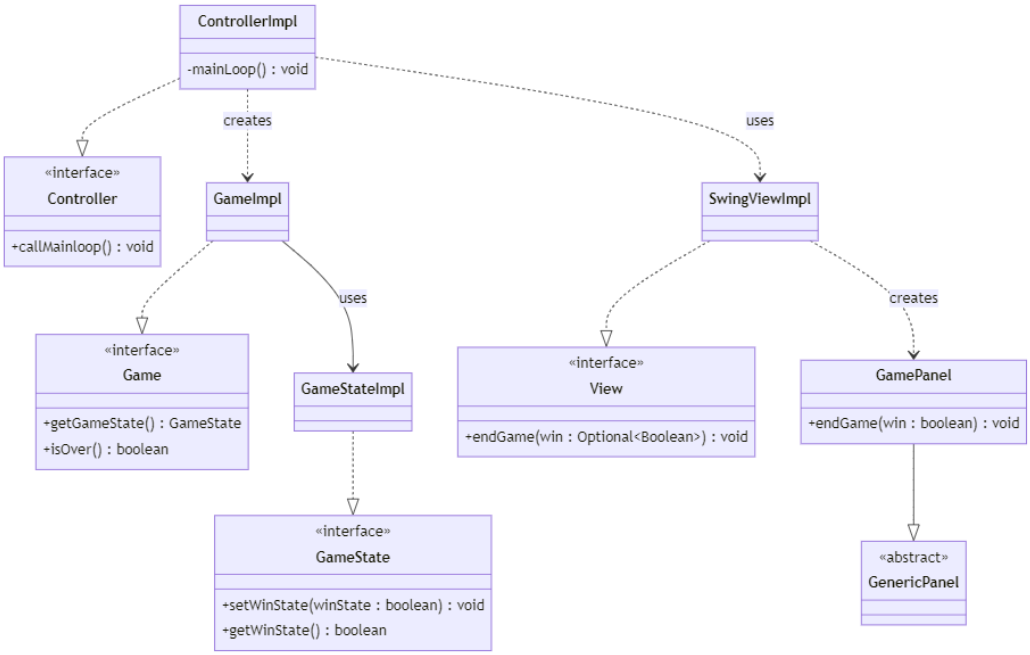

Figura 2.9: UML della struttura dell'evento GameOver (model + view)

Problema: Il gioco deve terminare restituendo un output visivo congruo allo stato del game over (che può essere di vittoria o di sconfitta),
Soluzione: Il gameloop viene terminato quando la funzione isOver() di GameImpl restituirà true. Intanto, il suo state (GameStateImpl) verrà aggiornato e tramite la funzione getWinState() si restituirà il caso di vittoria o di sconfitta alla View, nel pannello di gioco. In base all'esito del game over, la schermata mostrerà un messaggio diverso, tramite la funzione endGame(boolean win) del GamePanel, richiamata nella View.

Strutturazione della Griglia di Gioco

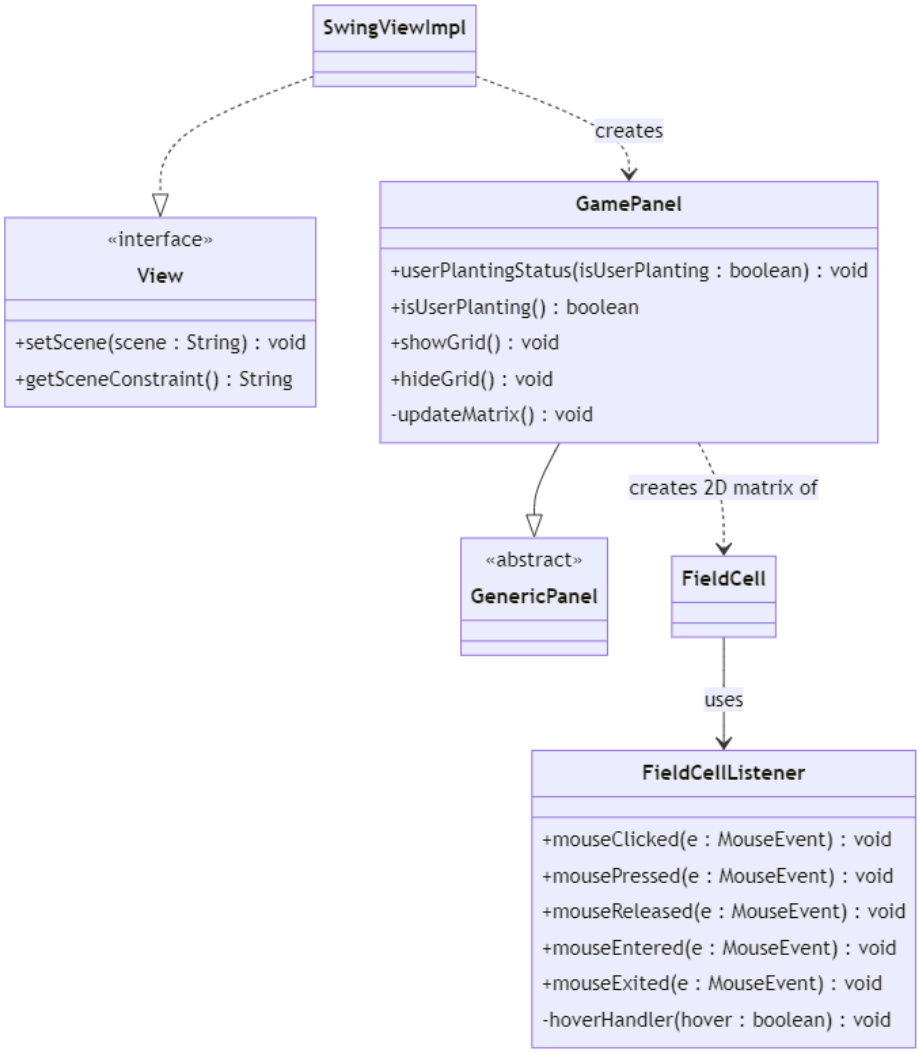

Figura 2.10: UML della struttura della griglia nel campo di gioco

Problema: Il background della schermata di gioco presenta una griglia dove si possono piazzare le entità piante. Bisogna realizzare una griglia per poter gestire gli input click-event dell'utente e tutto ciò che ne sussegue nel model.
Soluzione: Il pannello di gioco crea una matrice di FieldCell, una classe che estende il JButton e utilizza un action listener creato appositamente, la classe FieldCellListener (che ovviamente estende MouseListener). Questa cella della griglia gestisce gli eventi di hover (mouseEntered e mouseExited)

e il click (tramite mouseRelease, poiché è un evento che comprende sia il clicked che il pressed).

2.2.3 Margherita Zanchini

Strutturazione delll'entità pianta

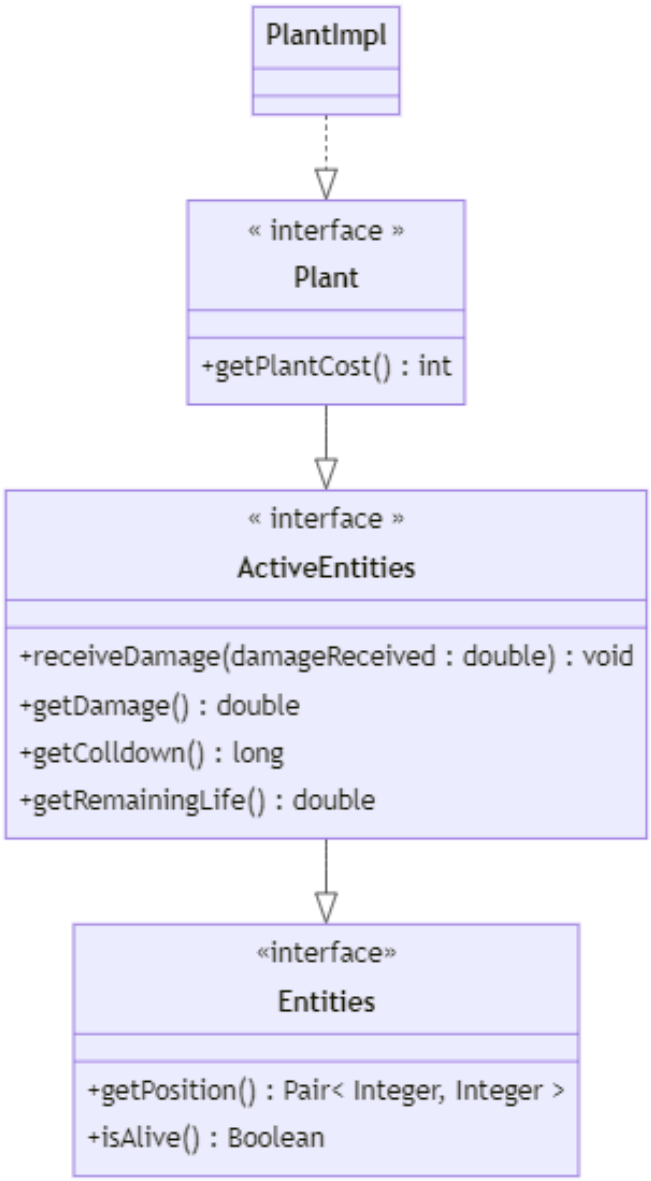

Figura 2.11: UML della struttura delle piante

Problema: Rendere il codice flessibile e facilmente modificabile, così da evitare anche il riuso del codice, quando si andranno a creare diversi tipi di entità ma con metodi simili.

Soluzione: Per risolvere questo problema è stato usato il pattern Strategy, e applicato il principio DRY. Per prima cosa è stato deciso da tutto il gruppo di creare un'interfaccia *Entities* che raggruppa tutte le entità, e poi un'interfaccia *ActiveEntities*, focalizzata sulle entità che possono attaccare e venire a loro volta attaccate, ovvero zombie e piante. Il pattern strategy ha quindi permesso di separare le responsabilità, evitando un'inutile ripetizione del codice.

Strutturazione della gestione del Proiettile

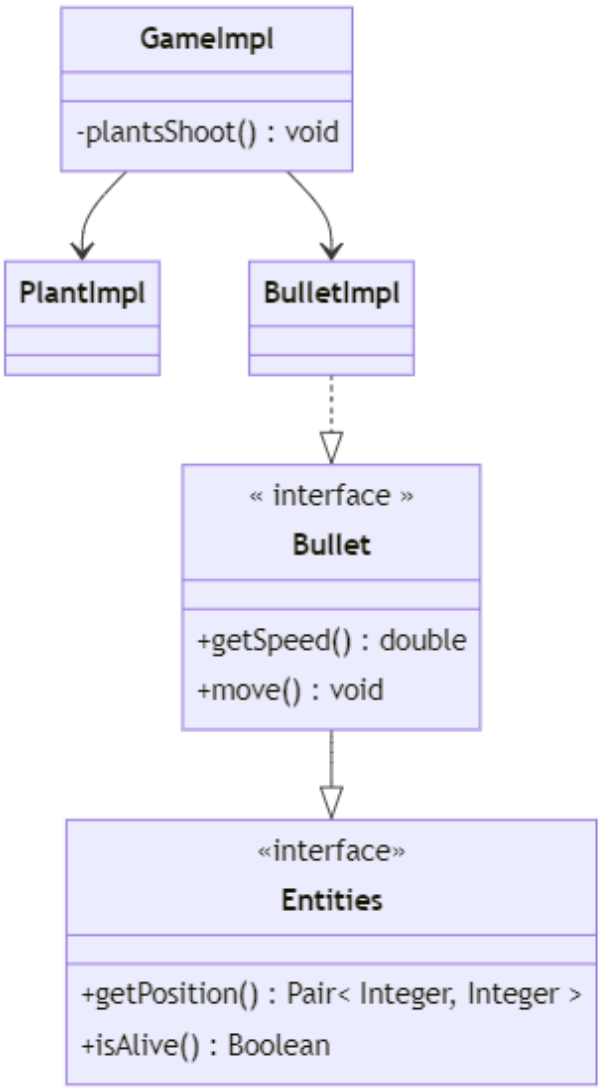

Figura 2.12: UML della gestione del proiettile

Problema: Ogni pianta deve sparare un proiettile.

Soluzione: Per risolvere questo problema invece di dare un metodo alla pianta che si occupasse del proiettile, ho fatto in modo che se ne occupasse il *GameImpl*. Il proiettile è quindi un'entità indipendente dalla pianta. In più gni pianta ha un cooldown, cioè il tempo che deve passare tra un attacco e un altro.

Strutturazione della gestione delle collisioni

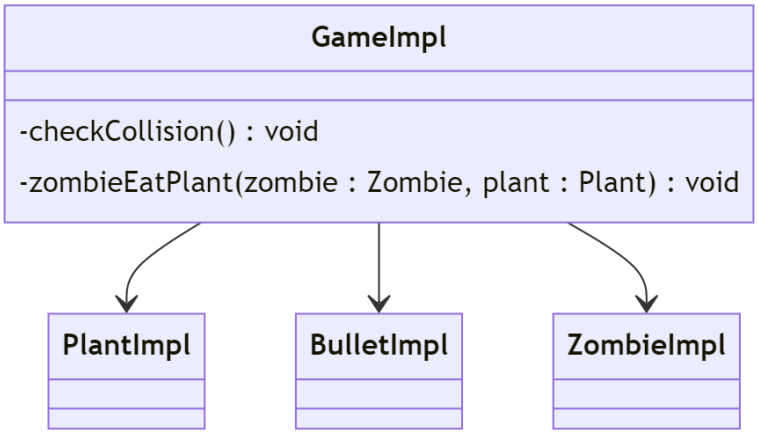

Figura 2.13: UML gestione collisioni

Problema: Gestione delle collisioni tra zombie e pianta e tra zombie e proiettile e della conseguente eliminazione delle entità.

Soluzione: Ho gestito le collisioni con un ulteriore metodo in *GameImpl*, ma mi rendo conto che sarebbe stato più efficace creare una classe a parte che si occupasse solo di questo, per evitare che la logica della gestione delle collisioni si mischi con la logica principale del gioco. Subito dopo che si è effettuata la collisione controllo che l'entità sia ancora viva (tranne il proiettile che viene tolto immediatamente) e se non lo è la rimuovo immediatamente.

Gestione della scalabilità della GUI

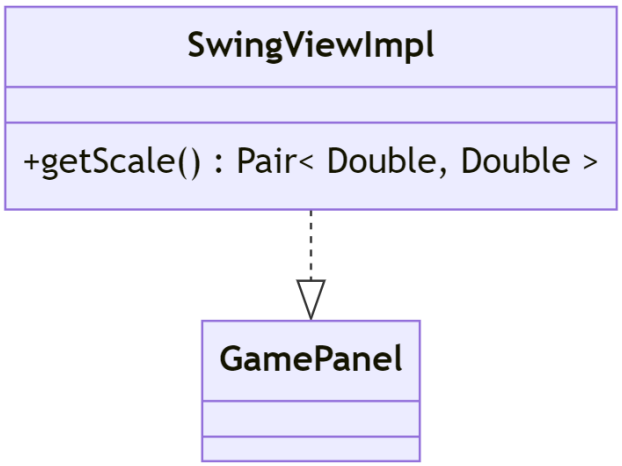

Figura 2.14: UML scalabilità GUI

Problema: Rendere la GUI scalabile, ovvero che non abbia dimensione fissa

Soluzione: Per risolvere questo problema sono state usate due classi *SwingViewImpl* e *GamePanel*. Io mi sono occupata della parte in *SwingViewImpl*. La classe ha un parametro chiamato *scale*, che indica quanto sono cambiate le proporzioni rispetto a quelle iniziali. Inoltre il pannello ha un Component Listener, che si occupa di capire quando la finestra viene ridimensionata e di conseguenza assegnare il nuovo valore al paramentro *scale*. Questo parametro verrà poi passato al *GamePanel* che si occuperà del resto.

2.2.4 Sofia Caberletti

Strutturazione dell'entità Sun

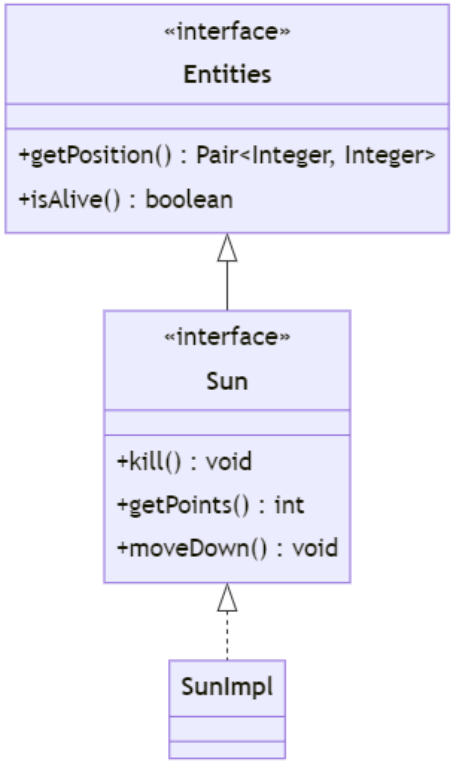

Figura 2.15: Schema UML della struttura dell'entità Sun

Problema: Creare una struttura per l'entità Sun che permetta di riutilizzare il codice in caso di futura creazione di Entities di tipo Sun con caratteristiche diverse.
Soluzione: La soluzione di questo problema sfrutta il pattern Strategy e il principio DRY. Abbiamo infatti deciso di definire una gerarchia di classi per la strutturazione delle entità che partisse dall'interfaccia Entities i cui metodi, che implementano azioni base di un'entità, vengono ereditati dalle interfacce Zombie, Plant e nel mio caso particolare Sun. In questo modo siamo riusciti a minimizzare inutile ripetizione di codice e a separare le caratteristiche e comportamenti specifici di ogni tipo di entità nelle rispettive classi.

Creazione delle entità Sun

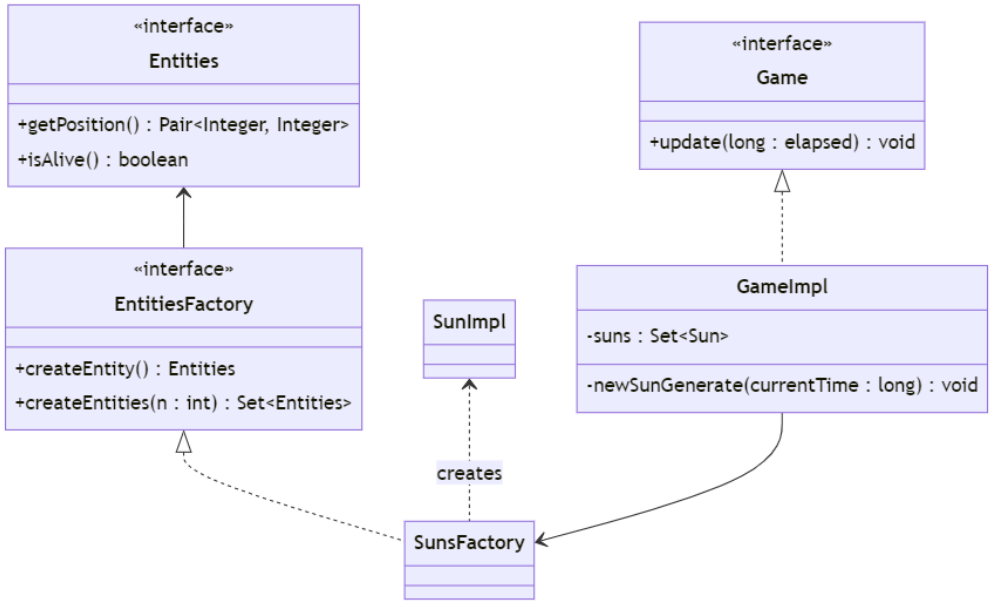

Figura 2.16: Schema UML di SunsFactory

Problema: Creazione delle entità di tipo Sun.
Soluzione: La soluzione di questo problema sfrutta il pattern Factory-Method. Inoltre SunsFactory, come anche ZombiesFactory, è un'implementazione specifica dell'interfaccia EntitiesFactory, il che vuol dire che esistono due diverse strategie per la creazione di Entities e quindi che ci troviamo di fronte a un caso di utilizzo del pattern Strategy. La specifica implementazione SunsFactory di EntitiesFactory, viene utilizzata nella classe GameImpl dentro al metodo newSunGenerate() che crea un sole ogni tot di tempo passato.

Aggiornamento delle posizioni delle Entities nel Model e nella View

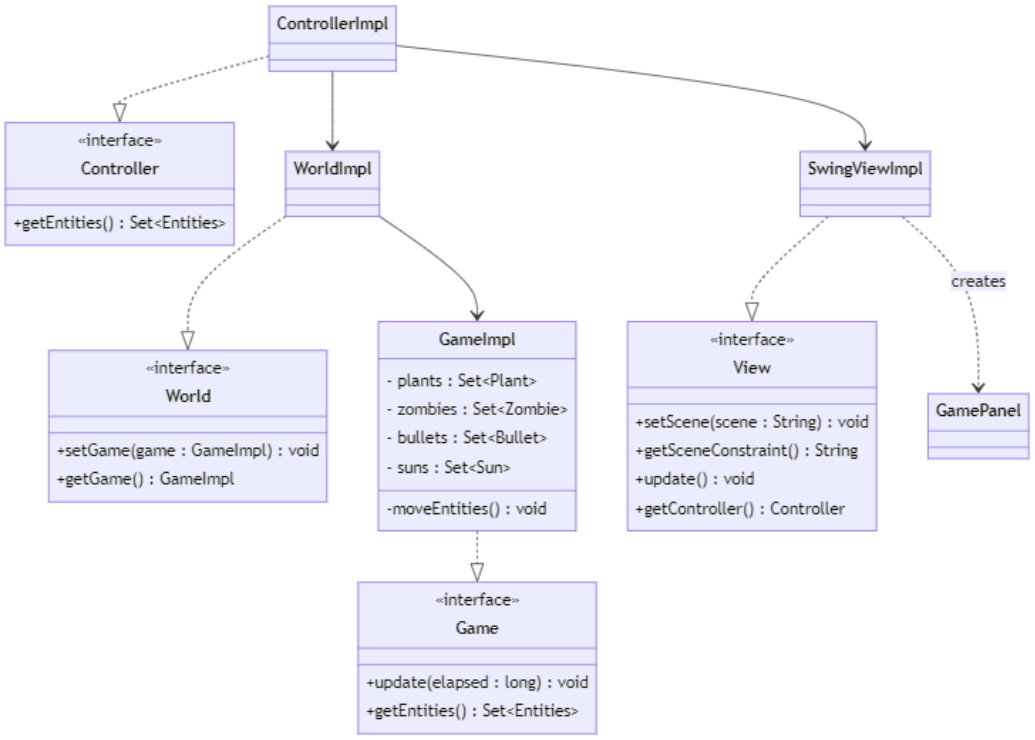

Figura 2.17: Schema UML dell'aggiornamento delle posizioni delle Entities

Problema: Aggiornare le posizioni delle entità nel Model e trasmettere tale aggiornamento alla View.
Soluzione: All'interno della classe GameImpl ho creato dei campi di tipo Set per ogni entità, questi Set contengono le entità presenti al momento nel gioco. Ho poi implementato un metodo di nome moveEntities(), richiamato dentro al metodo update(), che fa in modo che per ogni entità appartenente ai vari Set sia richiamato il rispettivo metodo di movimento. La View può reperire le varie entità con le posizioni aggiornate richiamando nella classe GamePanel il metodo getEntities() del ControllerImpl che a sua volta richiama il metodo getEntities() della classe GameImpl. Il metodo getEntities() che ho implementato in GameImpl restituisce un unico Set di Entities che contiene gli elementi dei Set plants, zombies, bullets e suns.

Avvio del Game Loop tramite comando ricevuto dalla View

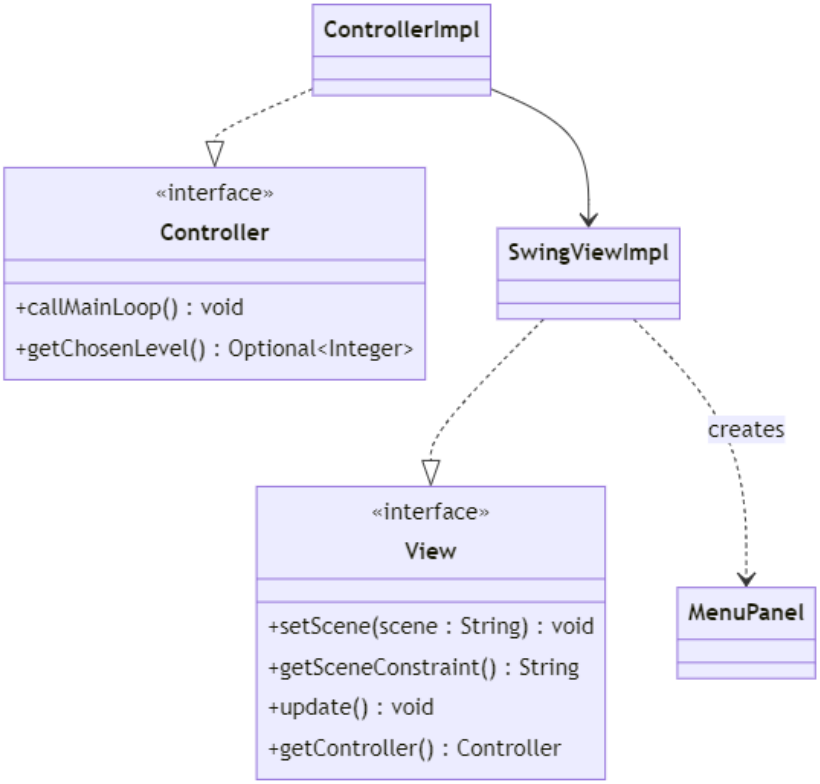

Figura 2.18: Schema UML dell'avvio del GameLoop

Problema: Avviare il Game Loop esattamente nel momento in cui viene incominciata la partita dall'utente tramite interfaccia grafica.
Soluzione: Ho implementato una classe MenuPanel che si occupa di mostrare graficamente il menu d'avvio del gioco. Il gioco viene avviato solo quando il giocatore preme il bottone di avvio della partita. Ciò è stato implementato grazie all'Action Listener associato al bottone di avvio che richiama il metodo del ControllerImpl callMainLoop() per far partire il Game Loop della partita. Quest'ultima funzione viene richiamata solo nel caso che il livello della partita sia già stato scelto e quindi richiamando il metodo getChosenLevel() l'Optional restituito non dovrà risultare empty.

Associazione del numero del livello scelto nella View al Model

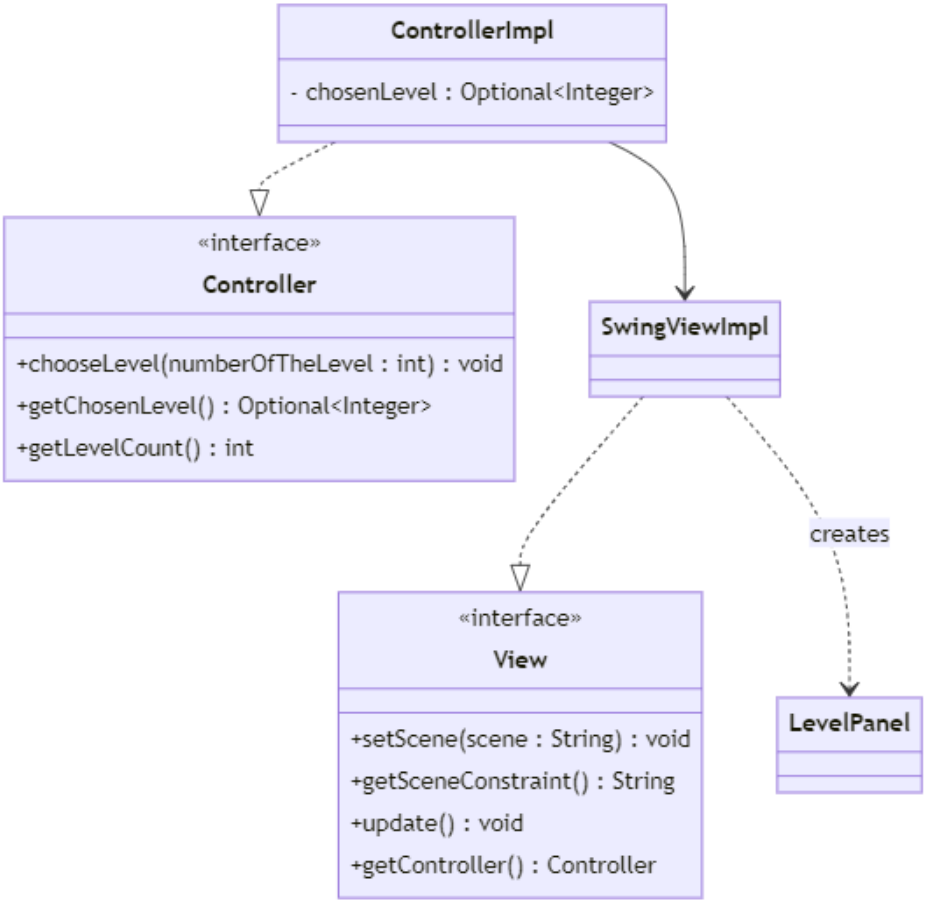

Figura 2.19: Schema UML della scelta del livello nella View e dell'associazione di tale informazione nel Model

Problema: Fornire al Model l'informazione del livello scelto dall'utente tramite interfaccia grafica.
Soluzione: Ho implementato una classe LevelPanel adibita a mostrare graficamente la scelta del livello. Per ogni livello è mostrato un botttone e il numero di livelli disponibili si conosce grazie al metodo getLevelCount() di ControllerImpl. Ad ogni bottone viene associato un Action Listener che richiama il metodo chooseLevel() del ControllerImpl. Nella classe ControllerImpl ho deciso di creare un campo di tipo Optional<Integer> per contenere il numero del livello scelto così da poter indicare anche la momentanea mancata presenza della scelta di un livello. In pratica quello che fa il chooseLevel() una volta richiamato è assegnare a chosenLevel il numero del livello fornito

dalla View. Grazie a questa tecnica il World potrà poi settare il proprio Level grazie al LevelsManager che restituirà il Level scelto in base al valore contenuto nel campo chosenLevel.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Tutti i test sono automatizzati, avendo utilizzato la suite JUnit. Abbiamo scelto di concentrarci sui test dei componenti principali del model evitando di dedicare test alla parte grafica. Per ciascuna entità è stato effettutato un test per verificarne le funzionalità, ad esempio la verifica del decremento della vita e di ulteriori parametri. Inoltre ci siamo assicurati che Soli, Zombie e proiettili si muovessero correttamente. Abbiamo inoltre testato che la classe responsabile dei livelli non contenesse alcun errore. Infine abbiamo verificato la corretta esecuzione del World, e testato solo alcune funzionalità del Game.

3.2 Note di sviluppo

3.2.1 Tutti i Componenti

Tutti i componenti si sono impegnati nell'utilizzo di un DVCS (distributed version control system, in questo caso `https://github.com`).
Il DVCS segue il model più semplice esistente per quanto riguardi il git branching, ovvero composto da un branch master e un branch dev(elopment). Sulla repository di GitHub si può trovare anche un terzo branch `"view_refactoring"`, in stato di abbandonno e pronto all'eliminazione, ma mantenuto come testimonianza del lavoro e dell'idea di un refactoring dell'implementazione della View. Inoltre, la repository presenta una licenza MIT e possiede la documentazione javadoc all'interno del Pages personale (consultabile su `https://marcomarrelli.github.io/OOP23-pvz-clone/`).

3.2.2 Irene Sofia Lotti

Uso di stream

Utilizzati in vari punti. Questo è un esempio.
Permalink: https://github.com/marcomarrelli/OOP23-pvz-clone/blob/788943456b8542b5d65e12aeec38a2ee60f8d1fa/src/main/java/pvzclone/model/impl/GameImpl.java

Uso di method reference

Permalink: https://github.com/marcomarrelli/OOP23-pvz-clone/blob/788943456b8542b5d65e12aeec38a2ee60f8d1fa/src/main/java/pvzclone/controller/impl/ControllerImpl.java#L47

3.2.3 Marco Marrelli

Uso di lambda expressions

Utilizzate in vari punti. Un esempio si può trovare al seguente permalink:
https://github.com/marcomarrelli/OOP23-pvz-clone/blob/788943456b8542b5d65e12aeec
src/main/java/pvzclone/view/impl/GamePanel.java#L249-L255

Uso di lambda-like switch

Permalink: https://github.com/marcomarrelli/OOP23-pvz-clone/blob/788943456b8542b5d65e12aeec38a2ee60f8d1fa/src/main/java/pvzclone/view/impl/GamePanel.java#L264-L270

Uso di `Optional`

Permalink: https://github.com/marcomarrelli/OOP23-pvz-clone/blob/788943456b8542b5d65e12aeec38a2ee60f8d1fa/src/main/java/pvzclone/view/impl/SwingViewImpl.java#L171-L177

3.2.4 Margherita Zanchini

Uso di lambda expressions

Un esempio:
https://github.com/marcomarrelli/OOP23-pvz-clone/blob/788943456b8542b5d65e12aeec
src/main/java/pvzclone/model/impl/GameImpl.java#L220

Uso di stream

Un esempio:
https://github.com/marcomarrelli/OOP23-pvz-clone/blob/788943456b8542b5d65e12aeec
src/main/java/pvzclone/model/impl/GameImpl.java#L274

3.2.5 Sofia Caberletti

Utilizzo di lambda expressions

Permalink: https://github.com/marcomarrelli/OOP23-pvz-clone/blob/
788943456b8542b5d65e12aeec38a2ee60f8d1fa/src/main/java/pvzclone/
view/impl/MenuPanel.java#L58-L74

Uso di `Optional`

Permalink: https://github.com/marcomarrelli/OOP23-pvz-clone/blob/
788943456b8542b5d65e12aeec38a2ee60f8d1fa/src/main/java/pvzclone/
controller/impl/ControllerImpl.java#L106

3.2.6 Codice Reperito Online

Classe `Pair`

Risorsa ottenuta dal laboratorio del corso e successivamente modificata. Permalink: https://github.com/marcomarrelli/OOP23-pvz-clone/blob/788943456b8542b5d65
src/main/java/pvzclone/model/impl/Pair.java
Crediti al Professor. Mirko Viroli.
Sito Web: https://www.unibo.it/sitoweb/mirko.viroli

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Irene Sofia Lotti

Sono compiaciuta del lavoro svolto. La coordinazione all'interno del gruppo si è svolta con serenità per l'intera durata del progetto. Nonostante le poche giornate rimanenti a disposizione, ci siamo tutti rimboccati le maniche per dare vita a un progetto di cui ci potessimo ritenere soddisfatti. Sicuramente, alcune implementazioni e idee di codice potevano essere elaborate in maniera differente. Con occhio critico, mi rendo conto di scelte compiute che non siano state completamente ottimali, come ad esempio la mancata realizzazione di classi che avessero come scopo quello di "alleggerire" il carico dei file con molte righe di codice. Questa è stata la mia prima esperienza di gruppo, e sono estremamente contenta di aver dato il massimo. Le sfide incontrate durante il progetto sono state varie, e alcune di elevata difficoltà, ma insieme siamo riusciti a gestirle e a superarle.

4.1.2 Marco Marrelli

Mi ritengo soddisfatto del lavoro svolto, singolarmente e insieme al resto dei componenti del gruppo. Sicuramente si sarebbe potuto dare di più, ma rimango appagato dal risultato ottenuto nonostante conoscenze e tempistiche limitate. È stata molto educativa e motivazionale come esperienza. Prima di questo corso non avrei mai pensato a Java come un valido linguaggio di programmazione per progetti complessi e aventi una `GUI`, ma questo progetto ha dato la possibilità di ricredermi fortemente. Da qualche anno utilizzo `QML` (insieme a `C++` e `Javascript`) in ambito lavorativo e per progetti personali, però mi sento spronato nell'approfondire le capacità del linguaggio Java e

vorrei in futuro creare un progetto con esso. Infine, voglio ringraziare le mie compagne di gruppo per aver dato il massimo delle loro capacità nonostante la presenza di altri esami importanti nella sessione, soprattutto durante lo `"sprint"` finale. L'ottima comunicazione e l'impegno dato da tutti nel cercare di migliorare le proprie capacità ha permesso a questo progetto di concludersi rispettando le tempistiche e le premesse date.

4.1.3 Margherita Zanchini

In merito alla mia parte svolta nel progetto, riconosco la presenza di molti difetti, tuttavia mi ritengo soddisfatta. Dovendo gestire più esami nell stesso periodo, il tempo a disposizione è stato limitato, e per questo motivo ritengo che il lavoro svolto sia discreto. Sono consapevole che ci siano molte cose da migliorare, ad esempio sarebbe stato oppurtuno dividere il GameImpl in diverse classi in base ai vari compiti. In più la separazione delle mansioni tra i membri del gruppo non è netta, nonstante l'ottima comunicazione tra i membri, infatti ci sono state molte parti svolte in collaborazione. Certamente avrei potuto fare di meglio. Tuttavia, considerando che è la mia prima esperienza con questo tipo di progetto, mi ritengo appagata e con le conoscenze apprese so che la prossima volta otterrò un risultato migliore.

4.1.4 Sofia Caberletti

In generale mi ritengo abbastanza soddisfatta del progetto svolto. Ora ho una visione più chiara di quali sono i vari step per creare un gioco partendo da zero e ritengo che questo progetto mi abbia anche aiutato a sviluppare le mie capacità di problem solving e a sentirmi più a mio agio a lavorare con strumenti e tecniche con cui non avevo dimistichezza. Credo che la parte più difficile sia stata l'assegnazione delle varie parti del progetto ai singoli componenti del gruppo, non tanto per un problema di comunicazione fra i vari membri, che ritengo sia stata ottima, ma più che altro a livello di progettazione iniziale. Mi sono infatti accorta di quanto sia essenziale basare il proprio programma su una struttura iniziale ben progettata e sono convinta che il codice potrebbe essere migliorato e reso più efficente e riusabile tramite una più attenta analisi e migliore conoscenza di pattern. Penso che sia stata soprattuto la classe GameImpl a risentire di questa cosa, in quanto ci siamo ritrovati ad avere una classe con tanti metodi implementati da diverse persone. Per il resto sono molto felice di questo progetto perchè ho apprezzato il processo del lavoro di gruppo e ho applicato le mie capacità in un lavoro concreto.

Capitolo 5

Esercitazioni di laboratorio

5.0.1 margherita.zanchini@studio.unibo.it

- Laboratorio 07:
  https://virtuale.unibo.it/mod/forum/discuss.php?d=147598#p209355
- Laboratorio 08:
  https://virtuale.unibo.it/mod/forum/discuss.php?d=148025#p210234
- Laboratorio 09:
  https://virtuale.unibo.it/mod/forum/discuss.php?d=149231#p211544
- Laboratorio 10:
  https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212695

5.0.2 sofia.caberletti@studio.unibo.it

- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=148025#p210356