Piante VS Zombie

Relazione per il progetto di *Programmazione ad Oggetti*
A.A. 2024/25

Leonardo Damario
leonardo.damario@studio.unibo.it

Luca Fabbri
luca.fabbri53@studio.unibo.it

Matteo Onofri
matteo.onofri4@studio.unibo.it

Maddalena Prandini
maddalena.prandini@studio.unibo.it

17 giugno 2025

Indice

Capitolo 1

Analisi

1.1 Descrizione del Software

Il software realizzato propone una versione semplificata di Plants vs. Zombies in cui l'utente veste i panni di un giardiniere-difensore chiamato a proteggere la propria casa da un'orda di **zombie** in avanzamento. Prima di ogni partita, si sceglie tra tre modalità di difficoltà selezionabile (Facile, Normale e Difficile), che regolano la capacità dei nemici di attaccare le piante. Una volta avviato il gioco, sullo schermo compare una **griglia 5×9**: ogni casella rappresenta un punto in cui posizionare una delle **tre piante** disponibili. Per piazzare un'unità è necessario spendere un certo numero di **soli**, la moneta di gioco, che i Sunflower producono automaticamente a intervalli regolari. Il Peashooter provvede invece all'attacco, lanciando proiettili in grado di danneggiare gli zombie, mentre il Wall-nut assorbe i primi danni, sacrificandosi dopo aver neutralizzato il primo assalitore che lo investe. Gli zombie emergono casualmente dal lato opposto della griglia e, avanzando verso la casa, si fermano per divorare le piante che incontrano; se riescono a raggiungere la fine di una corsia, attivano un tosaerba che ripulisce l'intera fila, ma rimane inutilizzabile in seguito. Un secondo zombie sfuggito al tosaerba condanna immediatamente il giocatore alla sconfitta. Il successo si ottiene eliminando un numero di zombie compreso tra 20 e 45, a seconda della difficoltà scelta; in caso contrario, la partita termina quando un nemico riesce ad attraversare le difese fino a colpire la casa

Requisiti funzionali

- **Menu di avvio del gioco**: all'apertura dell'applicazione compare una schermata che consente di selezionare il livello di difficoltà e la risoluzione con cui si vuole giocare la partita.

- **All'apertura del gioco**:
  - l'utente troverà una griglia di dimensione 5x9 in cui collocare le piante
  - l'utente potrà scegliere tra tre diversi tipi di piante (presenti nella barra apposita) quale posizionare nella griglia purchè abbia abbastanza soli per farlo
  - è possibile vedere se il giocatore può permettersi l'acquisto della pianta desiderata grazie ad una barra di aggiornamento che riporta il numero di monete (i soli) possedute attualmente e il numero di uccisione dei nemici
  - le piante che il giocatore può trovare sono
    * Sunflower: pianta di supporto che genera automaticamente nuovi soli a intervalli regolari, permettendo di accumulare risorse per piazzare ulteriori difese (senza risorse non è possibile acquistare nuove piante e quindi il gioco termina).
    * Peashooter: pianta offensiva che spara semi lungo la propria corsia, infliggendo danni continui agli zombie.
    * Wall-nut: barriera protettiva che blocca il primo zombie, sacrificandosi dopo averlo neutralizzato.
  - L'utente una volta posizionata la pianta impedisce l'avanzata degli Zombie: nemici che compaiono casualmente all'inizio di ciascuna corsia e avanzano verso la casa; se incontrano una pianta, la attaccano fino a distruggerla, subendo a loro volta danni da proiettili e tosaerba.
- Dopo aver ucciso un determinato numero di zombie in base alla difficoltà selezionata, il giocatore vince.

Requisiti non funzionali

- L'applicazione deve essere eseguibile nativamente su Linux, Windows e macOS.

1.2 Modello del Dominio

Il gioco si apre mostrando una griglia, in cui il giocatore ha il compito di difendere la propria casa (la fine della riga sinistra) da ondate di zombie, utilizzando una serie di piante con abilità differenti. Prima dell'inizio di ogni partita, l'utente può configurare due parametri fondamentali:

- **Difficoltà**: selezionabile premendo ripetutamente il pulsante "Seleziona difficoltà". Le tre modalità (Facile, Medio, Difficile) influenzano la capacità degli zombie di mangiare le piante più velocemente e le uccisioni necessarie per finire il gioco.

- **Risoluzione dello schermo**: impostabile tramite un menù a tendina. La risoluzione modifica le dimensioni dell'interfaccia e adatta graficamente la griglia di gioco, ma non influisce sulla logica interna del gameplay.

Una volta configurata la partita, viene mostrata una **griglia di gioco**, che rappresenta il campo di battaglia. La griglia è suddivisa in celle in cui possono essere piazzate le piante. I soli, generati regolarmente dai *Sunflower*, costituiscono la valuta del gioco e servono per acquistare le piante.

Il giocatore può scegliere tra tre tipologie di piante difensive, ognuna con un comportamento distinto:

- **Sunflower**: genera nuovi soli a intervalli regolari, permettendo di accumulare risorse.

- **Peashooter**: spara proiettili in linea retta lungo la riga per colpire gli zombie.

- **Wall-nut**: agisce come una barriera usa-e-getta, eliminando il primo zombie che lo colpisce e autodistruggendosi subito dopo.

Dalla parte destra della griglia, gli **zombie** iniziano ad apparire e avanzano lentamente verso sinistra, riga per riga. Ogni zombie ha punti vita e velocità differenti. Quando raggiungono una pianta, iniziano ad attaccarla finché non la distruggono, dopodiché riprendono il loro avanzamento.

Alla fine di ogni riga è presente un **tosaerba**, che viene attivato automaticamente la prima volta che uno zombie raggiunge la fine della corsia: esso elimina tutti gli zombie sulla riga, ma può essere utilizzato una sola volta per partita. Se uno zombie riesce a superare una riga dove il tosaerba è già stato attivato, la partita termina con la sconfitta del giocatore.

Il gioco termina in due condizioni:

- Il giocatore **vince** se elimina un numero sufficiente di zombie, valore che varia in base alla difficoltà selezionata.

- Il giocatore **perde** se uno zombie supera l'ultima linea di difesa in una corsia senza tosaerba.

Il sistema non prevede livelli multipli o vite, ma ogni partita è autoconclusiva e può essere ripetuta cambiando difficoltà.

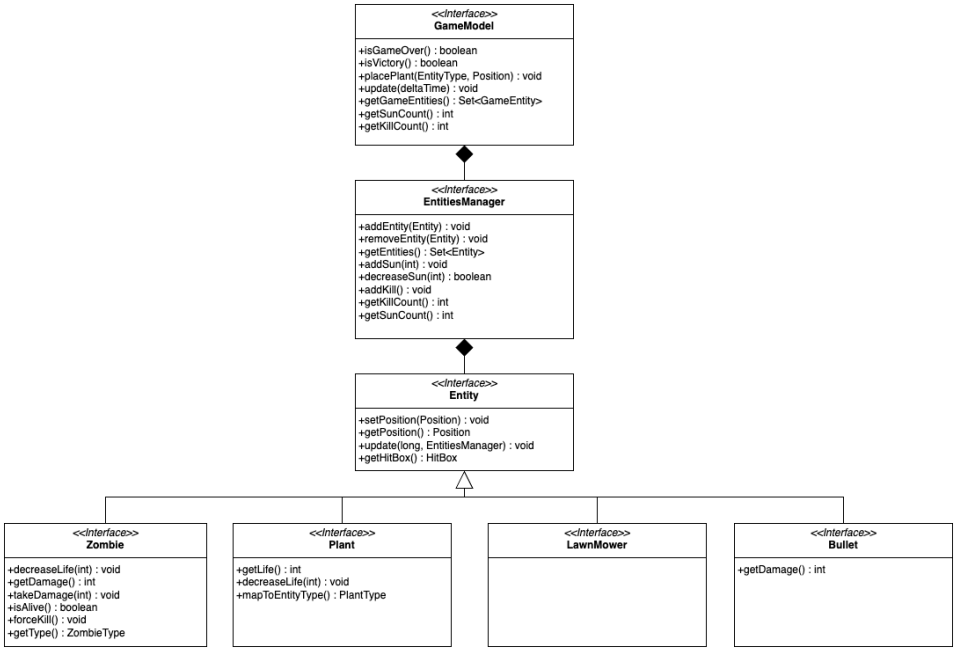


Figura 1.1: Entità del gioco

Capitolo 2

Design

2.1 Architettura

Piante contro Zombie adotta il pattern MVC. Il pattern separa i tre componenti logici (Model, View e Controller) che devono comunicare tra loro ma essere sconnessi.

- Il punto cardine del **Model** è l'interfaccia GameModel, che rappresenta la parte attiva del Model, quella che comunica con il Controller e si appoggia su una classe EntitiManager che contiene le entità della partita.
- Per il **controller** è stata creata una gerarchia per gestire tutti i layer del gioco. In particolare il controller è gestito da un parent controller (il MainController) che si suddivide a sua volta in tre controller per gestire la fine del gioco, il menù di gioco iniziale e il gioco stesso; eliminando qualsiasi dipendenza tra le classi. Sfruttando un Controller stratificato e specializzato è così possibile eliminare le dipendenze tra parti funzionali diverse dell'applicazione.
- Anche la **view** è stratificata in modo da essere utilizzata correttamente dal controller che si occupa di quella specifica funzionalità, in particolare il MenuView e l'EndGameView si occupano di mostrare i frame iniziale e finale mentre la GameView si occupa della parte del gioco

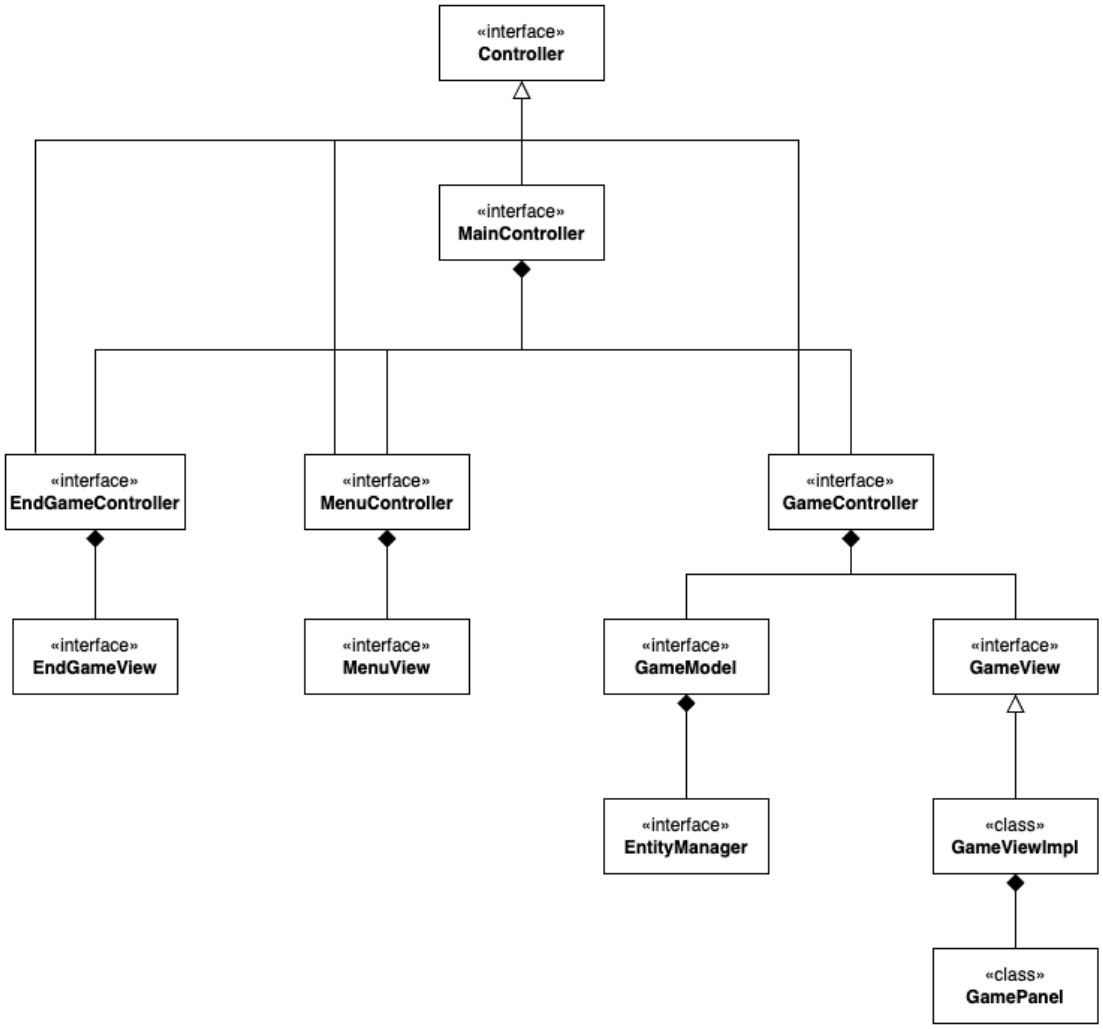


Figura 2.1: Architettura del gioco

2.2 Design dettagliato

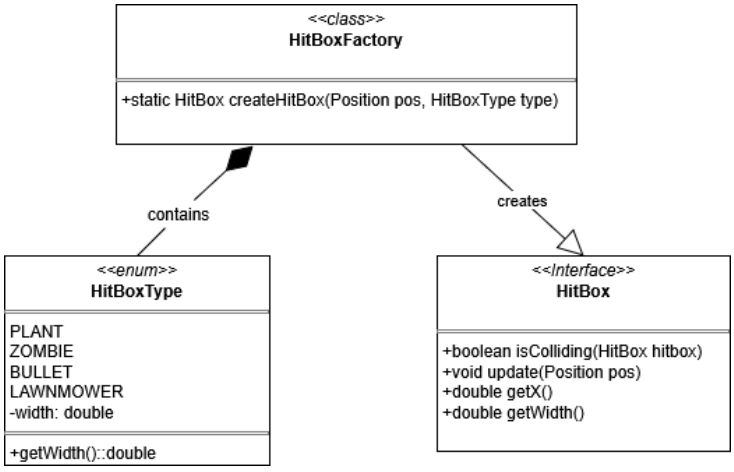
2.2.1 Leonardo D'Amario

Gestione delle collisioni

Gestione delle collisioni tra zombie, piante e proiettili.

Problema: Affinché gli oggetti possano collidere tra loro in modo più realistico, è necessario che evolvano da semplici punti rappresentati tramite una Position(Double, Double) ad entità dotate di una propria fisicità.

Soluzione: La soluzione implementata prevede l'utilizzo di un oggetto Hitbox, centrata rispetto alla posizione dell'entità, al fine di modellarne la fisicità. La Hitbox avrà un'estensione solo sull'asse x e non sull'asse y, poiché il gioco prevede collisioni soltanto tra entità presenti sulla stessa ordinata. Essa permetterà di verificare se è in collisione con un'altra Hitbox appartenente a un'altra entità, tramite il metodo isColliding(Hitbox). Ogni entità dovrà aggiornare la propria Hitbox a ogni movimento, ovvero a ogni variazione della propria Position, attraverso il metodo update(Position). Per generare Hitbox differenti in base al tipo di entità, si utilizza una HitBoxFactory, la quale si basa su modelli predefiniti contenuti nell'enumerazione HitboxType (PLANT, ZOMBIE, BULLET), che si distinguono principalmente per larghezze differenti.



Pro:

- Permette una gestione più realistica e modulare delle collisioni rispetto all'uso delle sole coordinate.

- La centralità rispetto alla posizione dell'entità semplifica i calcoli di collisione.
- Possibilità di definire hitbox differenti per tipo di entità tramite `HitBoxFactory`.

Contro:

- Maggiore complessità rispetto alla rappresentazione tramite semplici coordinate.
- Le hitbox hanno estensione solo orizzontale: ciò potrebbe limitare eventuali sviluppi futuri con collisioni bidimensionali.
- Necessità di aggiornare manualmente la hitbox a ogni cambiamento di posizione.

Problema: Gestire gli scontri effettivi tra zombie e piante, tra proiettili e zombie, e tra Wall-nut e zombie.

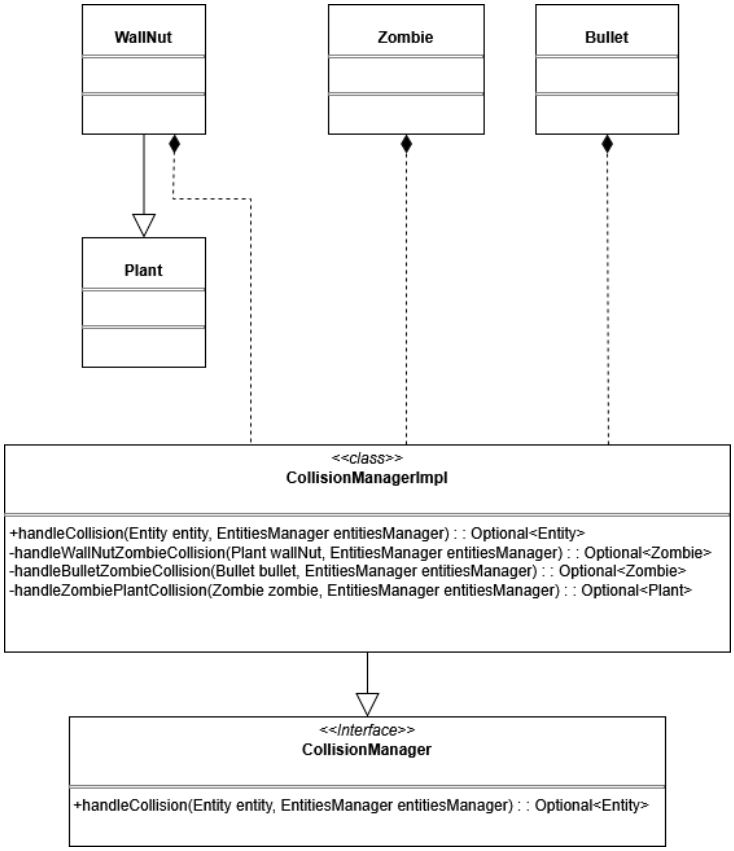
Soluzione: Per affrontare questo problema è stata progettata una classe dedicata alla gestione delle collisioni, denominata CollisionManager, utilizzata all'interno dei metodi di aggiornamento (update) delle singole entità. CollisionManager espone un solo metodo pubblico, handleCollision(Entity, EntitiesManager), che prende come input l'entità che lo invoca e restituisce l'oggetto con cui è avvenuta una collisione. Questo metodo identifica il tipo dell'entità chiamante e chiama il metodo privato corrispondente, che gestisce le collisioni specifiche per quella tipologia di entità.

Pro:

- Centralizzazione della logica di gestione delle collisioni, che favorisce la separazione delle responsabilità.
- Facilità di estensione per l'aggiunta di nuove regole di collisione specifiche per tipo di entità.
- Riduzione del codice duplicato all'interno delle entità.

Contro:

- Il metodo `handleCollision` deve distinguere e gestire manualmente ogni tipo di entità, rendendo il codice più lungo e potenzialmente soggetto a errori.
- L'architettura può diventare più difficile da mantenere se il numero di entità o tipi di interazione aumenta significativamente.



- Ogni entità deve comunque invocare esplicitamente il `CollisionManager`, introducendo un legame tra le entità e il gestore delle collisioni.

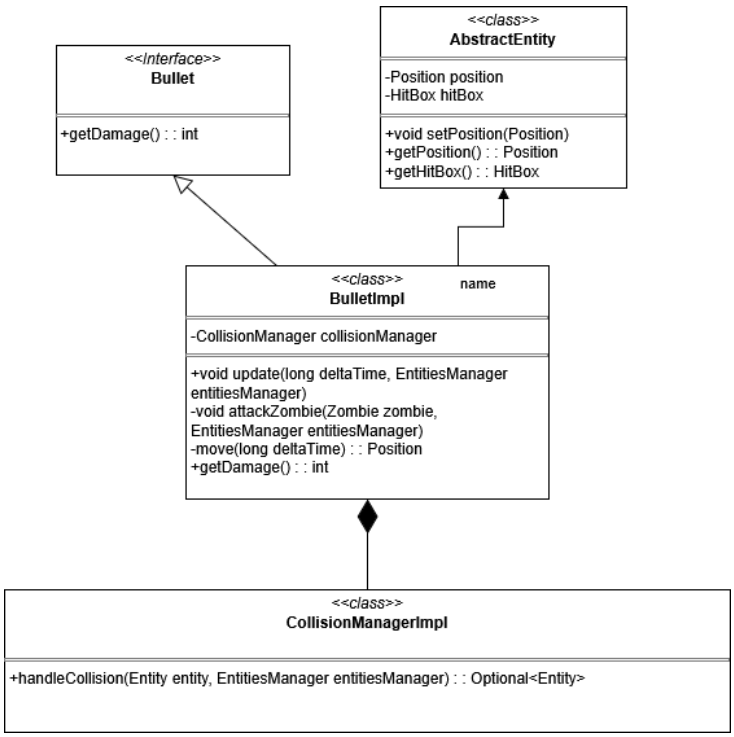
Gestione dei proiettili

Problema: È necessario modellare il comportamento dei proiettili nel gioco, in particolare:

- movimentazione orizzontale dei proiettili nel tempo;
- rilevamento delle collisioni con gli zombie;
- applicazione del danno in caso di collisione;
- rimozione del proiettile e dello zombie eventualmente eliminato.

Soluzione: La soluzione prevede l'implementazione della classe `BulletImpl`, che estende `AbstractEntity` e implementa l'interfaccia `Bullet`. Il proiettile

si muove orizzontalmente utilizzando una velocità costante definita da una variabile statica. Ad ogni aggiornamento (`update`), la sua posizione viene aggiornata e la `Hitbox` sincronizzata. La gestione della collisione con uno zombie è delegata al `CollisionManager`, il quale restituisce l'entità eventualmente collisa. In caso di collisione, il proiettile infligge un danno fisso allo zombie. Se lo zombie muore, viene rimosso dal `EntitiesManager` e viene incrementato il conteggio delle uccisioni. Il proiettile viene infine rimosso dal gioco.



Pro:

- Chiarezza e coerenza della logica del proiettile, focalizzata unicamente sul proprio comportamento.
- Uso di `CollisionManager` per separare la logica di collisione, migliorando modularità e manutenibilità.
- Velocità e danno rappresentati come costanti, facilmente configurabili.
- Movimento preciso tramite `BigDecimal`, adatto anche a step temporali piccoli o irregolari.

- Utilizzo sicuro di `Optional` per evitare eccezioni legate a valori nulli.

Contro:

- L'istanza del `CollisionManager` è creata internamente, limitando testabilità e inversione delle dipendenze.
- Il danno è fisso: non è prevista una configurazione o un comportamento differenziato tra proiettili.
- In caso di collisioni multiple simultanee, solo la prima viene gestita.
- Ogni proiettile istanzia un proprio `CollisionManager`, con possibile overhead.

2.2.2 Luca Fabbri

Gestione Controller

Problema: Gestione delle chiamate al controller principale e comportamento dei controller subordinati

Soluzione: I controller sono gestiti attraverso una Chain of Responsibility. Un'interfaccia controller generalizza le funzioni più comuni per ogni controller. Il MainController gestisce le relazioni tra i controller secondari (GameController, MenuController ed EndGameController), ai quali passa una copia di sé stesso quando vengono creati. In questo modo, quando c'è un passaggio di stato dell'applicazione (Menu, Game, EndGame), il controller corrente fa una chiamata al controller padre, il quale si occupa di gestire il cambio di stato.

Pro: La soluzione proposta evita ripetizioni di codice con l'utilizzo delle interfacce (DRY) e fornisce una separazione chiara delle responsabilità (SRP) dei singoli controller e permette di aggiungere nuovi controller senza modificare quelli esistenti grazie alla chain of responsibility.

Contro: La soluzione presenta interfacce multiple e dunque una maggiore complessità.

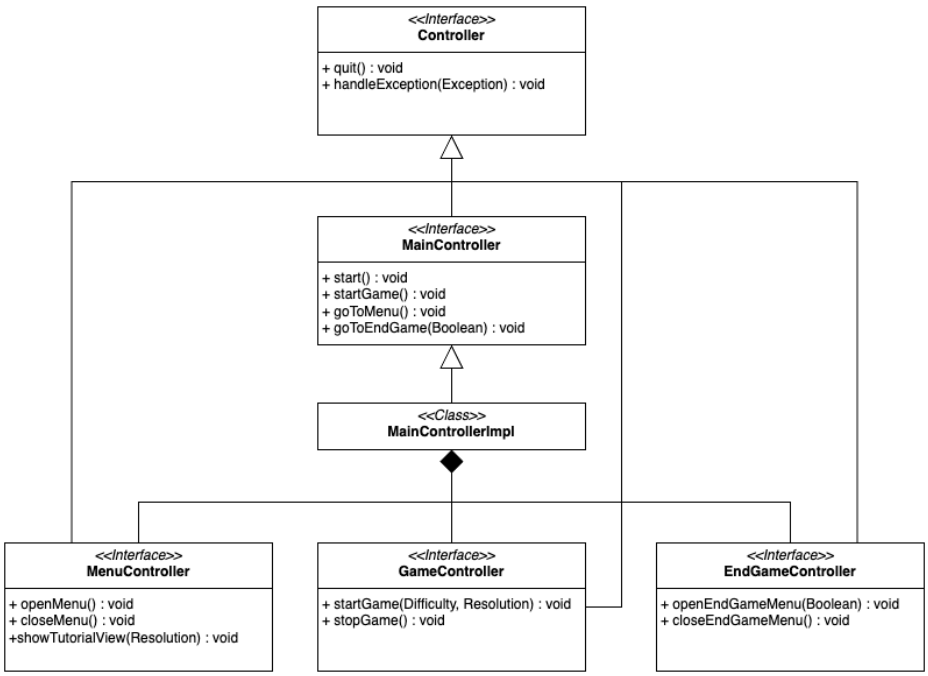


Figura 2.2: Rappresentazione dei vari controller del gioco

Entities Manager

Problema: Gestione delle entità di gioco

Soluzione: La gestione delle entità di gioco è affidata a un EntitiesManager che funge da classe "Data Container". La logica del gioco e le chiamate alle funzioni avvengono nel GameModel, che si appoggia sull'EntitiesManager per il conteggio dei soli, delle uccisioni e per la gestione delle entità presenti in gioco.

Pro: La soluzione proposta rispetta i principi SOLID, il che contribuisce a rendere il codice semplice da leggere, manutenere e utilizzare. Inoltre, l'elevato grado di modularità permette di riutilizzare il componente anche in contesti con logiche di gioco differenti, favorendo la scalabilità e l'estendibilità del codice.

Contro: L'estensione seguita dall'aggiunta di troppe informazioni riduce la coesione e richiede la creazione di un'altra classe "Container" per evitare

incoerenza, inoltre la mancanza di logica è vulnerabile a dati incoerenti o errati.

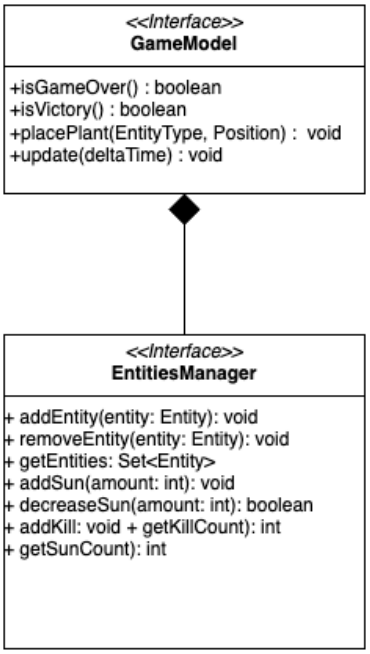


Figura 2.3: Rappresentazione UML di EntityManager e GameModel

Gestione dei Tosaerba

Problema: Nel gioco, i LawnMower rappresentano una meccanica difensiva "estrema": devono essere attivati automaticamente quando uno zombie raggiunge l'inizio della corsia, avanzare lungo la riga, eliminare tutti gli zombie incontrati e poi essere rimossi dal campo.

Soluzione: L'interfaccia LawnMower con la sua implementazione permette ad ogni tosaerba di gestire in autonomia la propria logica di movimento e la gestione delle collisioni con gli zombie. La sua attivazione e rimozione è invece compito del GameModel che è l'unico a conoscere le dimensioni della griglia di gioco, dunque lo aggiunge alle entità di gioco quando uno zombie arriva alla prima casella di qualsiasi riga e lo rimuove una volta uscito dal campo di gioco.

Pro: Facilità nel manutenere ed estendere aggiungendo nuove tipologie di LawnMower che possono essere aggiunte senza modificare il resto del model o il controller. La soluzione rispetta il principio SRP (Single Responsibility Priciple) che la rende anche facilmente testabile.

Contro: L'autonomia dei LawnMower comporta una maggiore complessità nelle interazioni tra le entità di gioco, specialmente nella gestione delle collisioni e della loro rimozione dal campo. La gestione dell'attivazione e rimozione affidata al gamemodel può portare a potenziali problemi di sincronizzazione.

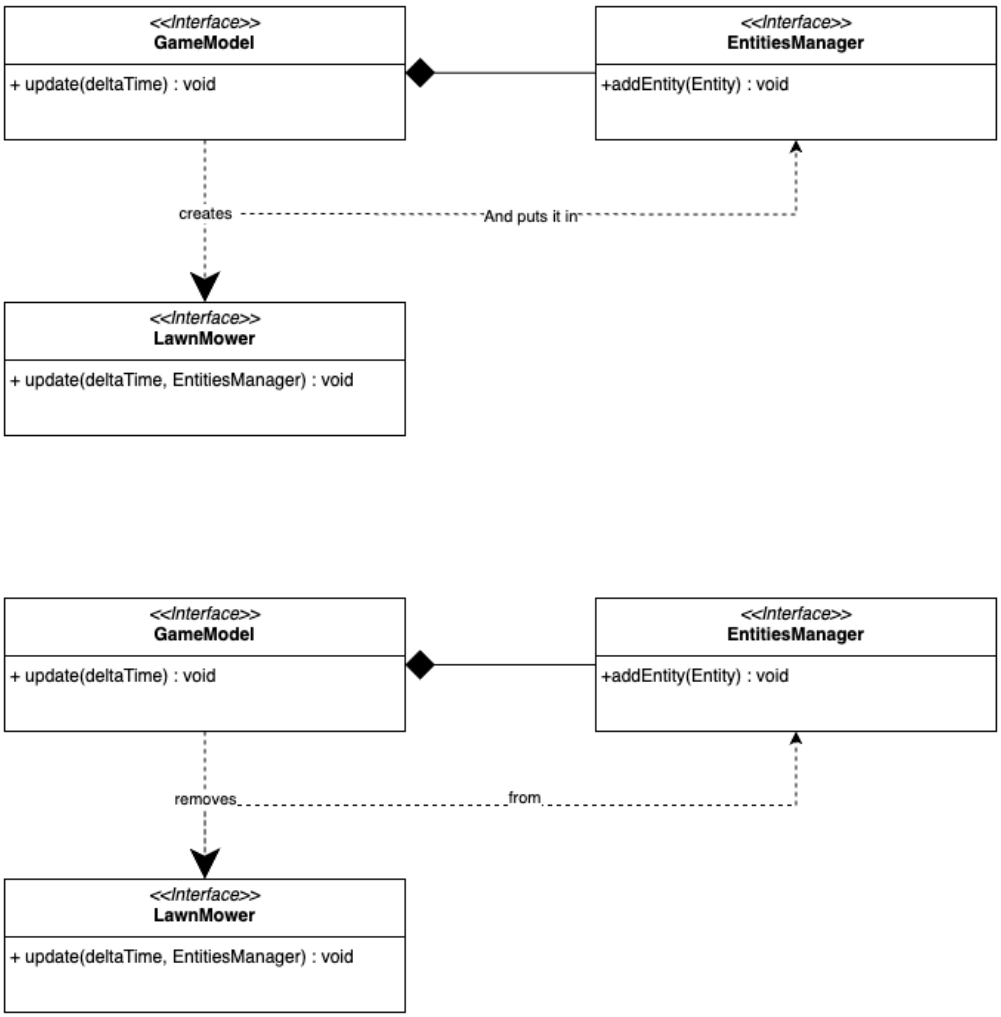


Figura 2.4: Rappresentazione UML dell'azione di GameModel su LawnMower

2.2.3 Matteo Onofri

Architettura delle Classi Zombie (Template Method Pattern)

Problema: Era necessario centralizzare i comportamenti comuni a tutti gli zombie (come il movimento, la gestione della vita e la logica di attacco) in un unico punto per evitare duplicazioni del codice. Allo stesso tempo, ogni tipo di zombie (Basic, Fast, Strong, Beast) doveva avere attributi unici e specifici (punti vita, velocità, danno).

Soluzione: Per risolvere questa dualità è stato implementato il **Template Method Pattern**

- Il fulcro è la classe astratta AbstractZombie che funge da "template". Essa implementa l'interfaccia Zombie e contiene tutta la logica condivisa:
  - gestione della vita, velocita e stato alive.
  - il meotodo update(), definisce il comportamento: controlla le collisioni e, in base al risultato, attacca o si muove. La logica di attacco, incluso il cooldown(ATTACK RATE), è definita qui.
  - il metodo move() che calcola lo spostamento.

- La classe definisce anche metodi abstract come **getDamage()** e **getType**, che le sottoclassi sono obbligate ad implementare

- Le classi concrete come BasicZombie, FastZombie, StrongZombie e BeastZombie estendono AbstractZombie. Ognuna di esse:
  - Definisce i propri attributi (vita, velocità, danno) come costanti private.
  - Passa questi attributi unici al costruttore della classe madre super().
  - Fornisce l'implementazione concreta dei metodi astratti.

Pro:

- **Massimo riutilizzo del codice**: La logica complessa e condivisa è scritta una sola volta in AbstractZombie.

- **Struttura coerente**: Il pattern garantisce che tutti gli zombie seguano lo stesso ciclo di vita e comportamento di base, riducendo la possibilità di errori.

- **Estendibilità**: Creare un nuovo tipo di zombie richiede solo l'estensione di AbstractZombie e fornire le implementazioni specifiche.

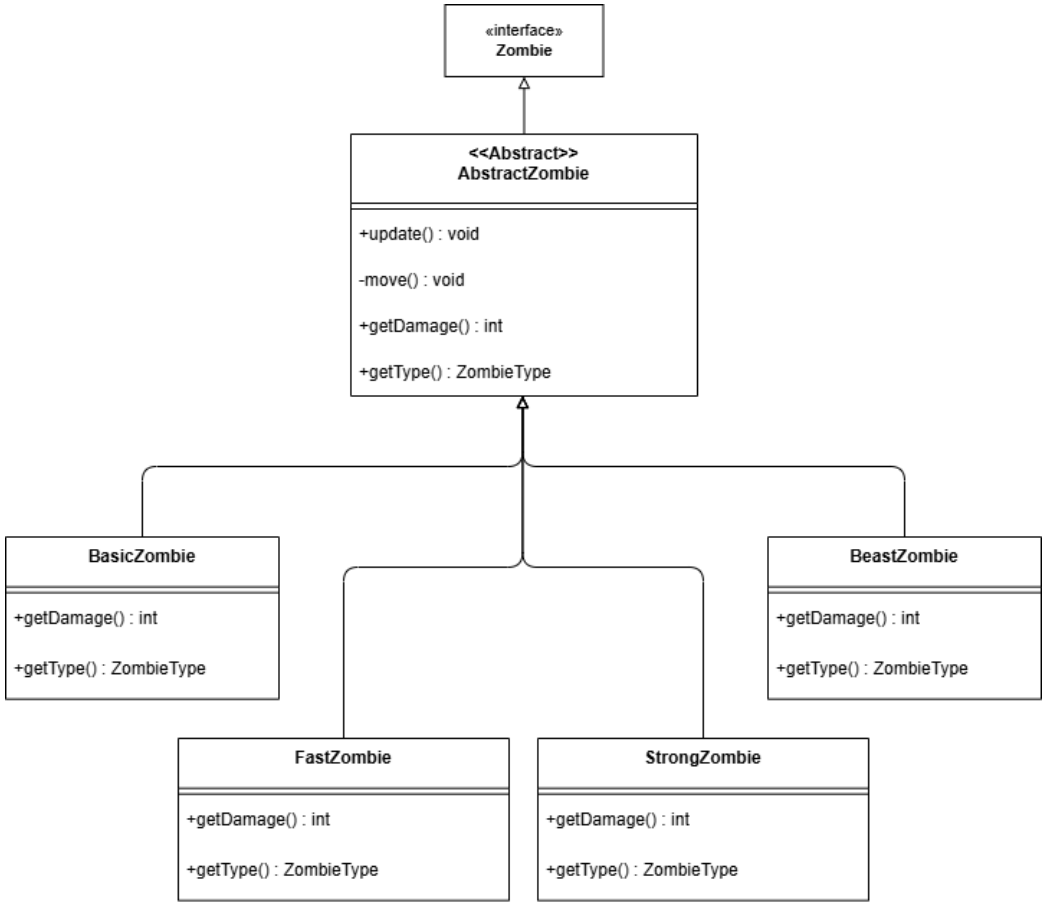


Figura 2.5: UML del Template Method per le classi Zombie.

Contro:

- **Rigidità**: Il template definisce una struttura fissa. Se un nuovo zombie richiedesse una logica di update radicalmente diversa, questo pattern risulterebbe limitante.

Creazione Centralizzata degli Zombie (Factory Pattern)

Problema: Il codice che necessitava di creare zombie (come la logica di spawn) non doveva essere accoppiato alle classi concrete (new BasicZombie(), new FastZombie(), ecc.). Era necessario un punto di accesso centralizzato.

Soluzione: È stato implementato il **Factory Pattern** attraverso la classe ZombieFactory.

- Offre un unico metodo statico createZombie(String type, Position position).
- All'interno del metodo, uno switch statement basato sulla stringa type si occupa di istanziare e restituire l'oggetto AbstractZombie corretto.
- Se viene fornito un tipo non valido, la factory lancia un'IllegalArgumentException, rendendo il sistema robusto.

Pro:

- **Disaccoppiamento**: Il client (in questo caso ZombieSpawnUtil) non conosce le classi concrete degli zombie, ma solo la factory e i tipi di stringa.
- **Centralizzazione**: La logica di creazione è tutta in un unico posto, rendendo più semplice la manutenzione o la modifica del processo di istanziazione.

Contro:

- **Uso di stringhe**: La factory si basa su stringhe per la selezione del tipo quindi meno sicuro di un approccio basato su enum.

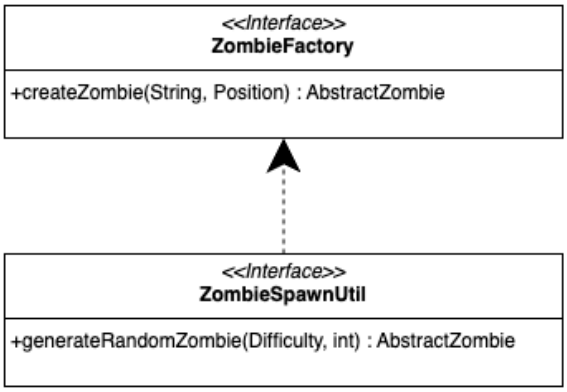


Figura 2.6: UML della relazione ZombieSpawnUtil e ZombieFactory

Logica di Generazione Casuale (Separazione delle Responsabilità)

Problema: La generazione di zombie non è solo un'operazione tecnica, ma anche di logica di gioco: bisogna decidere quale zombie creare in base alla difficoltà della partita e a delle probabilità specifiche. Questa logica non doveva essere mescolata con quella della creazione.

Soluzione: È stata creata la classe ZombieSpawnUtil per gestire esclusivamente la logica di generazione.

- Questa classe è responsabile di determinare il tipo di zombie da creare in base alla Difficulty del gioco.
- Il metodo getRandomZombieType(Difficulty) contiene le tabelle di probabilità per ogni livello (Easy, Normal, Hard) e restituisce il tipo di zombie scelto come String.
- Il metodo principale generateRandomZombie() prima determina una posizione di spawn casuale, poi invoca getRandomZombieType() per decidere quale zombie generare, e infine delega la creazione effettiva alla ZombieFactory, passandole il tipo di stringa ottenuto.

Pro:

- **Single Responsibility Principle**: ZombieSpawnUtil ha la sola responsabilità di decidere quale zombie creare, mentre ZombieFactory ha la sola responsabilità di come crearlo. La separazione è netta.
- **Manutenibilità della Logica di Gioco**: Se si vogliono modificare le probabilità di spawn, basta intervenire su ZombieSpawnUtil, senza toccare le altre classi.

Contro:

- **Valori Hard-Coded:** Le probabilità di spawn sono "hard-coded" nella classe. Per una maggiore flessibilità, potrebbero essere caricate da un file di configurazione esterno.

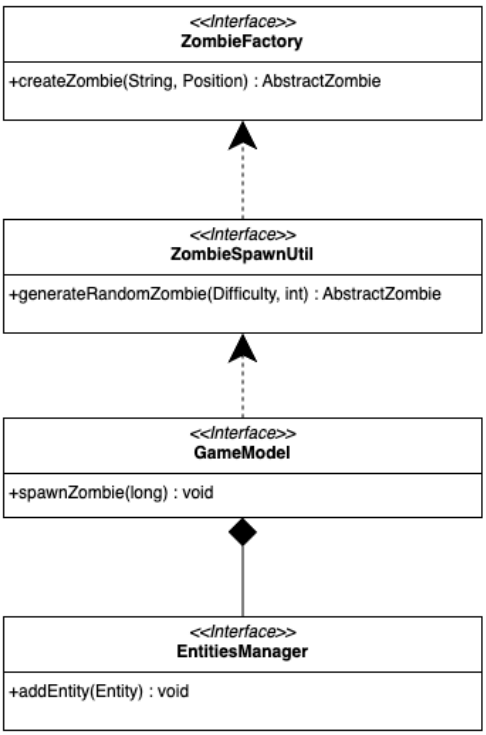


Figura 2.7: UML del modello di generazione Zombie

2.2.4 Maddalena Prandini

Rappresentazione delle Entità

Problema: all'interno del gioco è necerassario presentare una vasta gamma di oggetti dinamici: piante, zombie, proiettili, soli ecc. Questi oggetti condividono diverse caratteristiche comuni come:

- Posizione su una griglia
- Dimensioni e forma (date dalla hit box)
- Gestione dello stato di vita
- Comportamenti nel tempo (update)

Il problema principale è modellare una classe che consenta di centralizzare aspetti comuni, consentire l'estensione da parte di entità eterogenee e consentire ad un gestore esterno (GameModel) di trattare le entità in maniera uniforme, senza conoscerne i dettagli

Soluzione:

- Utilizzare un'interfaccia **Entity** che specifica i metodi fondamentali per qualunque oggetto del gioco (getPosition, getHitbox ecc)
- Una classe **Abstract Entity** che implementa l'interfaccia e fornisce una base comune per entità come proiettili, zombie e piante
- Le entità sono gestite da classi esterne (in particolare dall'EntitiesManager), ma il loro comportamento interno è incapsulato in ogni istanza, secondo il principio di responsabilità singola (SRP).

Pro:

- **estendibilità**: aggiungere nuove entità richiede solo l'implementazione dell'interfaccia
- **uniformità**: tutte le entità sono trattate nello stesso modo dal gestore delle entità
- **risutilizzo**: codice per gestione posizione, vita e hitbox è centralizzato

Contro:

- **Crescita della AbstractEntity**: col tempo, la classe base rischia di accumulare troppa logica condivisa (o semi-condivisa), violando il principio di coesione e rendendo difficile il mantenimento.

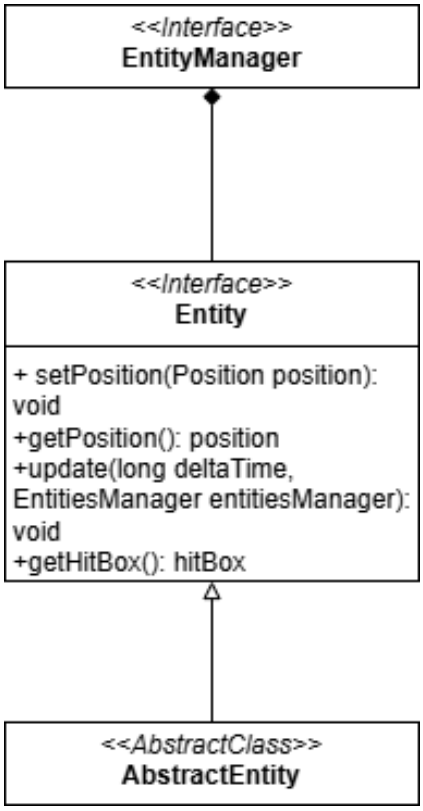


Figura 2.8: Rappresentazione delle entità nel gioco

Creazione delle Piante

Problema: Nel gioco, bisogna creare piante con caratteristiche comuni, azioni e caratteristiche specifiche (es.sparare proiettili, generare soli, eliminare gli zombie). I problemi principali relativi all'implementazione per la loro creazione, sono:

- Consentire l'estensione per introdurre nuove tipologie di piante in futuro
- Centralizzare la logica comune, rispettando il principio DRY

Soluzione: Le piante sono delle entità, con una hitbox, una vita e un'azione unica che si attiva durante il gioco. Per gestire la loro creazione e comportamento in modo flessibile e riutilizzabile sono stati utilizzati i seguenti pattern:

- **Template Method:** Utilizzato nella classe astratta **AbstractPlant** per definire la struttura comune delle piante, delegando i comportamenti specifici alle sottoclassi anonime definite nella **PlantFactory**. Il metodo *getLife()* è il template Method perché definisce la logica comune per tutte le piante (la loro vita attuale nel gioco è definita dalla differenza della vita iniziale specifica della pianta e il danno subito). Il template Method chiama *getMaxLife()*, un metodo astratto implementato nelle classi che impletano la classe astratta.
- **Factory Pattern:** Implementato nella classe **PlantFactory**, centralizza la creazione delle istanze di piante. Per ogni tipo di pianta (PlantType), viene creata un'istanza di AbstractPlant tramite classi anonime specificando il comportamento del metodo *update()* e il valore di *getMaxLife()*.

Pro:

- **Riuso del codice**: il riutilizzo di codice è centralizzato nella classe AbstractPlant.
- **Struttura coerente**: avendo definito la struttura generale nella classe base, tutte le sottoclassi seguono lo stesso schema garantendo coerenza nel comportamento.
- **Separazione delle responsabilità**: le parti invarianti dell'algoritmo sono gestite nella classe base, mentre le parti variabili sono delegate alle sottoclassi, migliorando la manuntebilità.
- **Facilita' di estensione**: è semplice aggiungere nuove piante implementando solo i metodi specifici, senza alterare la struttura esistente.

Contro:

- **Aumento della complessità**: l'uso combinato di più pattern introduce un livello di astrazione che può risultare meno immediato.

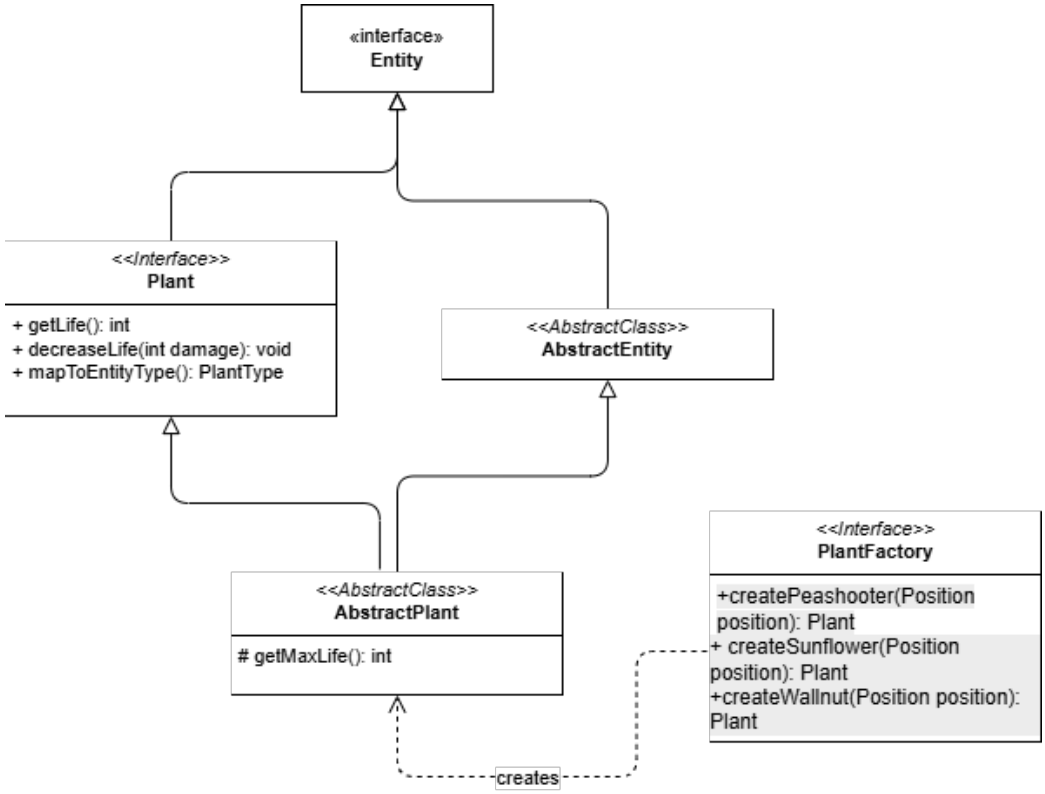


Figura 2.9: Rappresentazione UML per la creazione delle piante

Entry point nel model e gestione della partita

Problema: Nel modello architetturale del gioco è necessario fornire un punto di accesso centralizzato e ben definito per la gestione della logica di gioco. Tale componente deve apparire all'esterno come una entità passiva, ovvero non deve gestire direttamente eventi o cicli di esecuzione, ma offrire un'interfaccia che permetta al controller di interagire con il model in modo chiaro e controllato.

Un'altra criticità consiste nella gestione dello stato della partita, che deve includere logiche di aggiornamento delle entità, spawn di nuovi zombie, gestione della vittoria/sconfitta e piazzamento delle piante, il tutto mantenendo separate responsabilità e permettendo il riutilizzo e testabilità dei singoli componenti.

Soluzione: Per risolvere questo problema è stata introdotta la classe GameModelImpl, che implementa l'interfaccia GameModel. Essa rappresenta l'entry point del model e fornisce al controller metodi come update, pla-

cePlant, getSunCount e isGameOver, mantenendo l'interfaccia del model semplice e ben separata dall'implementazione secondo il principio di singola responsabilità. Al suo interno, GameModelImpl sfrutta un componente chiamato EntitiesManager, responsabile della gestione concreta delle entità di gioco (piante, zombie, proiettili, tagliaerba ecc.). Questo permette di centralizzare e isolare la logica di aggiornamento e interazione tra entità.

In particolare:

- Il metodo update consente di aggiornare lo stato del gioco in base al tempo trascorso, senza che il GameModelImpl gestisca direttamente il ciclo di gioco (che resta a carico del controller).
- Il metodo placePlant incapsula la logica di piazzamento delle piante, delegando la creazione a una PlantFactory, e verifica che vi siano sufficienti risorse prima di aggiungere la pianta.
- La logica di spawn degli zombie è integrata con una gestione del tempo e della difficoltà, completamente astratta rispetto all'esterno. In questo modo, il GameModelImpl risolve efficacemente il problema dell'entry point passivo del model, fornendo un'interfaccia stabile e facilmente estendibile, e delega le responsabilità specifiche a componenti specializzati, rispettando i principi di progettazione ad oggetti come la separazione delle responsabilità e l'incapsulamento

Pro:

- **Separazione dei concetti**: La classe delega la gestione delle entità a EntitiesManager, rispettando il Single Responsibility Principle e mantenendo il GameModelImpl focalizzato sulla logica di gioco complessiva.
- **Incapsulamento**: Espone solo ciò che serve al controller e alla vista, mantenendo private le informazioni interne e proteggendo l'integrità dello stato.
- **Flessibilità nella gestione dello stato**: La centralizzazione dello stato del gioco consente di gestire facilmente transizioni di stato (es. da IN_PROGRESS a WON o LOST) in un unico punto.

Contro:

- **Complessità accentrata**: Anche se GameModelImpl delega molte responsabilità, centralizza comunque logiche rilevanti (piazzamento piante, aggiornamento stato, spawn nemici), e rischia di diventare un og-

getto “monolitico” difficile da mantenere se la complessità del gioco aumenta.

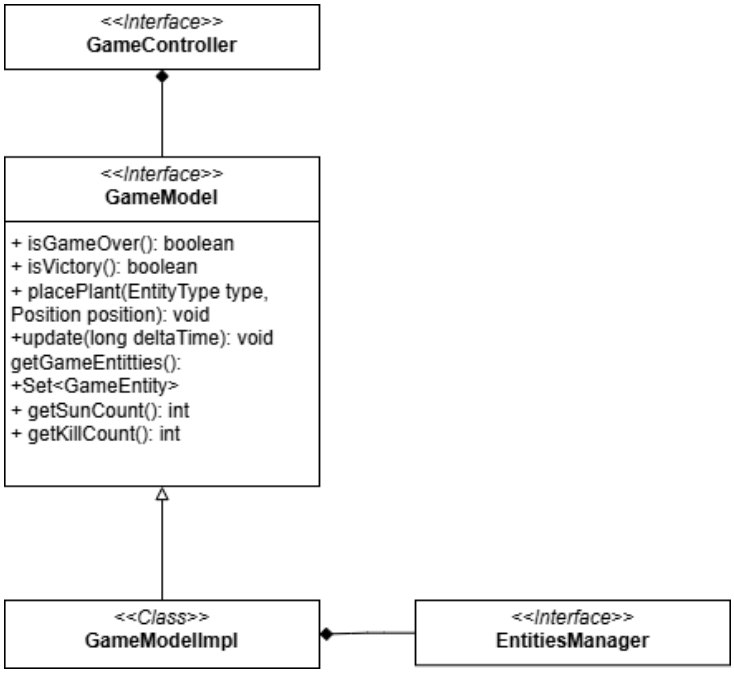


Figura 2.10: Rappresentazione UML dell’organizzazione del Model

Gestione del flusso tra le tre componenti del pattern MVC

Problema: Uno dei principali problemi affrontati nello sviluppo del gioco è stato la gestione del flusso di informazioni tra le componenti View e Model, che operano in maniera asincrona e con responsabilità ben distinte. E’ necessario garantire:

- Coerenza tra input utente e stato del gioco
- Disaccoppiamento tra i componenti

In particolare, il problema riguarda la gestire di due flussi distinti:

- Un flusso bidirezionale **View ←Controller→ Model**, che riguarda gli input dell’utente: ad esempio, la selezione di una pianta e il successivo clic sulla griglia per posizionarla.
- Un flusso unidirezionale **Model → View**, in cui il modello aggiorna lo stato interno del gioco (movimenti, spawn, collisioni, ecc.), e la view deve riflettere fedelmente questo stato al giocatore

Soluzione: Per affrontare questa complessa gestione del flusso informativo tra View e Model è stato introdotto un componente centrale, il GameControllerImpl, che rappresenta il fulcro del coordinamento logico dell'applicazione. Il controller agisce come mediatore attivo, disaccoppiando le due componenti principali (View e Model) e orchestrando le interazioni in modo modulare, asincrono e controllato. A livello di design, il GameControllerImpl ha le seguenti responsabilità:

- Per il flusso **View ←Controller→ Model** (input utente): il controller implementa l'interfaccia ViewListener, ricevendo dalla View gli input utente già interpretati (come la selezione di una pianta o il clic su una cella). Questi input non modificano direttamente il Model, ma vengono incapsulati in oggetti evento e inseriti in una coda di eventi. Questo approccio consente una gestione ordinata e asincrona degli input e la possibilità di estendere facilmente il sistema con nuovi tipi ti evento
- Per il flusso **Model → View** (aggiornamento gioco), durante ogni iterazione del game loop, il controller elabora gli eventi specifici nella coda, applicandoli al modello di gioco tramite metodi specifici. Successivamente chiama il metodo update sul GameModel, facendo avanzare lo stato del gioco (movimenti, spawn, collisioni...). Infine interroga il modello per ottenere tutti gli elementi visibili tramite interfacce standardizzate (es. GameEntity per vedere le entità presenti nel gioco) e li trasmette alla view

Pro:

- nel flusso **per il flusso View ←Controller→ Model**
  - **Sincronizzazione implicita**: gli input utente vengono serializzati nella coda
  - **Ordine garantito**: è rispettata la sequenza selezione pianta → piazzamento
  - **Disaccoppiamento**: View e Model non si vedono mai direttamente, tutto passa per eventi intermedi
- nel flusso **per il flusso Model → View**
  - **Frequenza controllata**: la view viene aggiornata solo nel tick del game loop, evitando update eccessivi
  - **Coerenza**: i dati mostrati riflettono sempre lo stato "ufficiale" del modello dopo il tick

Contro:

- **Stato intermedio fragile**: il controller mantiene uno selectedPlant-Type (nel caso l'utente non avesse abbastanza soli per comprare la pianta fino a quando li raggiunge) tra due eventi (il click della toolbar e il click della griglia) che può portare a rallentamenti nel piazzamento

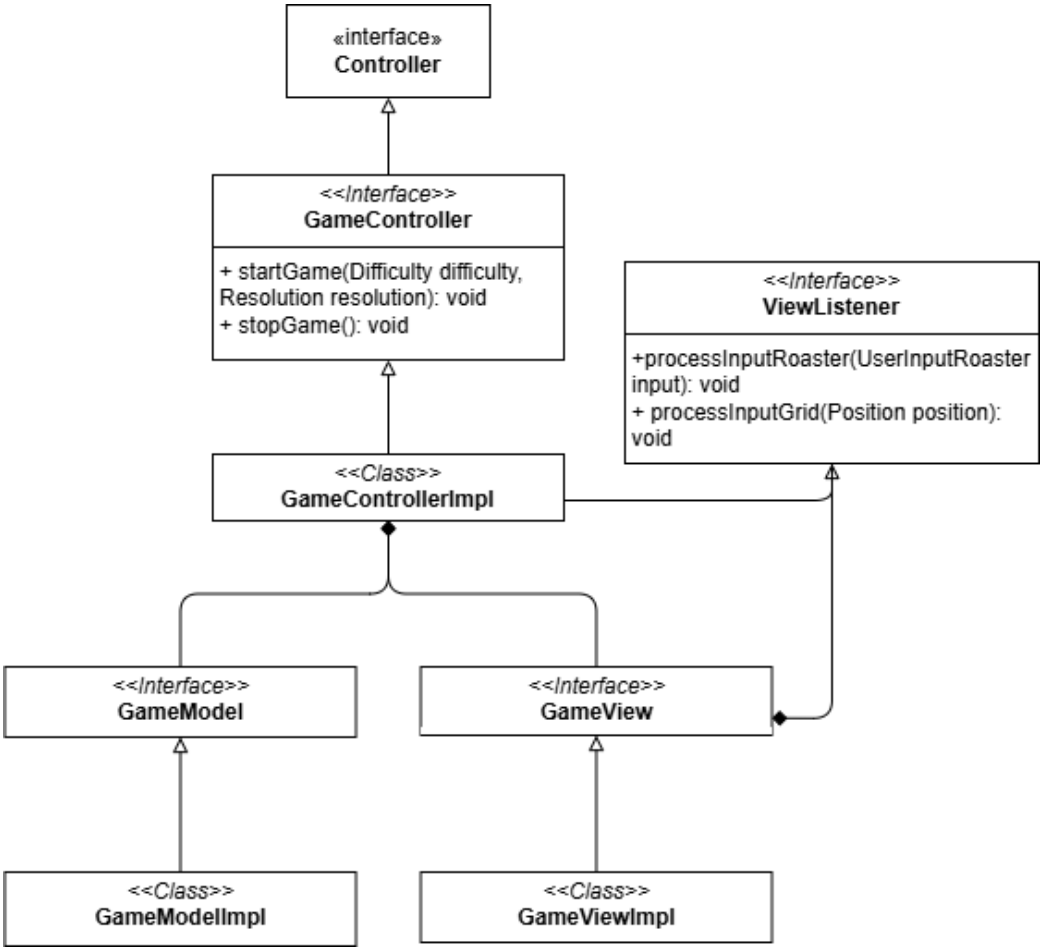


Figura 2.11: Rappresentazione UML del flusso del Controller

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Elementi del modello e dei controller sono stati testati utilizzando JUnit. Seguono le classi testate:

- **PlantFactoryTest**: E' stata testata se la creazione della pianta avviene in maniera corretto, restituendo il tipo di pianta scelto e le sue caratteristiche associate
- **UpdateTest**: E' stata testata la capacità delle piante di aggiornare correttamente la propria azione
- **CollisionsTest**: E' stato testato il corretto funzionamento del CollisionManager e quindi delle varie collisioni tra le entità.
- **EntitiesManagerTest**: E' stato testato il corretto funzionamento dell'entitiesmanager nel raccogliere e restituire le entità e i counter.
- **LawnMowerTest**: E' stato testata la corretta creazione dei tosaerba e la loro collisione con gli zombie.
- **ZombieTest**: E' stato testato il corretto funzionamento degli zombie e quindi la loro capacita di subire danni e la corretta diminuzione dei punti vita, si è inoltre verificato che ogni zombie venga creato con i giusti attributi di danno e velocità.

3.2 Note di sviluppo

3.2.1 D'Amario Leonardo

Utilizzo di Lamba expression e Stream : ad esempio dentro i metodi privati di CollisionManagerImpl : permalink.

3.2.2 Fabbri Luca

Utilizzo di Optional :Un esempio in questo permalink.

Utilizzo di lambda : Un esempio in questo permalink

Utilizzo libreria Math : Un esempio in questo permalink

3.2.3 Onofri Matteo

Utilizzo di Optional :Un esempio in questo permalink.

3.2.4 Prandini Maddalena

Utilizzo di Lamba expression e Stream :Un esempio in questo permalink.

Utilizzo della libreria Math : Un esempio in questo permalink

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 D'Amario Leonardo

Questo progetto ha rappresentato per me una delle esperienze più formative dal punto di vista del lavoro di gruppo. All'inizio non è stato semplice: comunicare in modo efficace, coordinarsi e trovare un metodo di lavoro condiviso richiedeva uno sforzo costante. Tuttavia, con il tempo, il team è riuscito a trovare un equilibrio, trasformandosi in un vero e proprio gruppo affiatato e collaborativo. Un aspetto che mi ha colpito è stata l'importanza di scrivere codice chiaro, manutenibile e facilmente modificabile: in più occasioni ci siamo trovati a dover intervenire su porzioni già sviluppate, e avere una base solida ha reso queste operazioni molto meno complesse. Il lavoro iniziale di progettazione architetturale è stato particolarmente difficile: prendere decisioni strutturali senza avere ancora una visione completa del sistema è stato impegnativo. Tuttavia, col senno di poi, quel tempo investito è stato determinante per facilitare lo sviluppo successivo dell'applicazione, permettendoci di lavorare con una direzione chiara e coerente.

4.1.2 Fabbri Luca

Questo progetto per me è stato molto importante per comprendere le difficoltà che si incontrano lavorando in gruppo. Ripensando all'inizio e agli errori dovuti all'inesperienza, sono comunque soddisfatto dello sviluppo progressivo che mi ha permesso di entrare nell'ottica di scrivere un codice leggibile, curato e facilmente estendibile, oltre che di migliorare la mia capacità di relazionarmi con gli altri membri del team. Ho anche compreso il vero motivo dietro l'utilizzo di pattern e principi: se applicati correttamente, rendono il

lavoro individuale e di gruppo molto più semplice, soprattutto nei progetti di larga scala. Arrivato alla fine, quasi vorrei poter ricominciare da capo con l'esperienza acquisita, per vedere quante cose avremmo potuto fare più velocemente e per scoprire ancora meglio quanto margine di miglioramento ci sia.

4.1.3 Onofri Matteo

Partecipare a questo progetto ha rappresentato la prima vera occasione per capire cosa significa lavorare in un contesto collaborativo. Lavorare in team mi ha insegnato una lezione fondamentale che va oltre la semplice programmazione. Inizialmente pensavo che suddividersi il lavoro sarebbe stata la parte più complessa, ma mi sono presto accorto che la vera sfida era far sì che il pezzo di codice che scrivevo si integrasse bene con quello dei miei colleghi. Ho iniziato a capire a cosa servissero tutte quelle regole di progettazione e pattern che studiamo. Non sono regole astratte, ma strumenti pratici che permettono a persone diverse di collaborare efficacemente su un unico obiettivo. Ripensando all'intero percorso, sono soddisfatto di come sia cambiato il mio approccio, se all'inizio mi concentravo solo sul "far funzionare" il mio codice, ora penso anche a come renderlo comprensibile e utile per gli altri. Questa esperienza mi ha lasciato la competenza più importante: non solo come scrivere codice, ma come costruirlo insieme ad altri.

4.1.4 Prandini maddalena

Cimentarmi in questo progetto è stata per me una prima vera esperienza di sviluppo collaborativo su larga scala, e si è rivelata tanto impegnativa quanto istruttiva. Non avevo mai affrontato un lavoro che richiedesse un livello alto di organizzazione del codice, coordinamento tra componenti e gestione di un repository condiviso. È stata tuttavia un'esperienza formativa che mi ha fatto crescere sia come sviluppatore che come membro di un team. In futuro, mi piacerebbe tornare sul progetto per estenderlo e rifinire ulteriormente alcune idee emerse durante il lavoro.

Appendice A

Guida utente

Breve spiegazione dei comandi e delle dinamiche del gioco:

- Il menu principale (A.1) consente di leggere il tutorial di gioco, selezionare il livello di difficoltà (cliccando ripetutamente il pulsante SelezionaDifficolta'), impostare la risoluzione dello schermo (mediante il menù a tendina) e avviare la partita.

Figura A.1: Menù

- Una volta avviata la partita si presenterà la schermata di gioco (A.2), in cui vengono mostrate la griglia di gioco e i bottoni per scegliere le piante da piazzare sulla griglia.

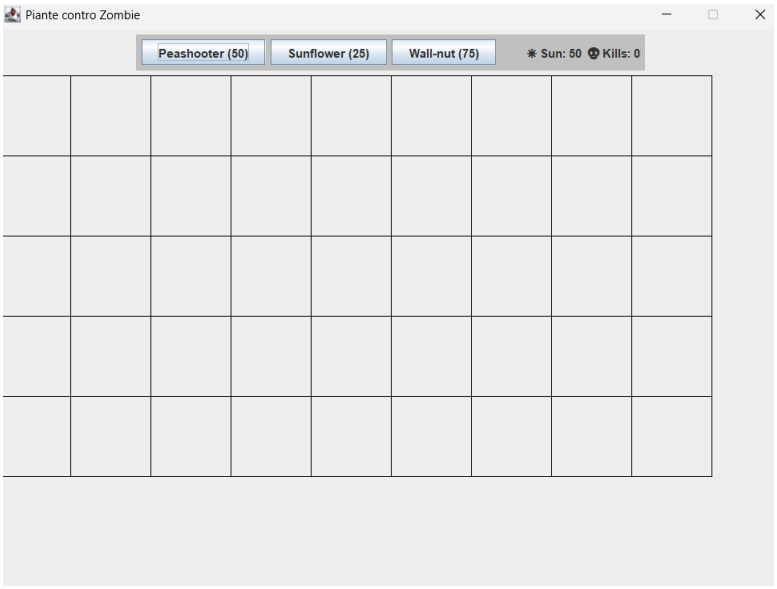


Figura A.2: Interfaccia di gioco

- La prima cosa che deve fare il player è piazzare sulla griglia i Sunflower (A.3), cliccando sul bottone apposito e poi su una casella della griglia, in modo tale da iniziare a far crescere i soli che servono da valuta per poi comprare le piante difensive.

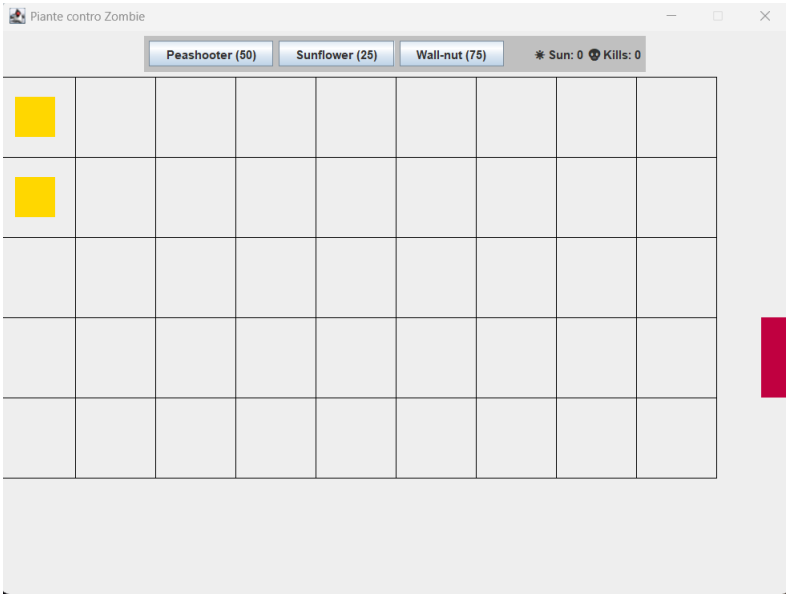


Figura A.3: Disposizione soli

- Una volta schierati i sunFlower, il player avrà a disposizione un rifornimento di soli per potersi difendere dall'avanzata degli zombie schierando i Peashooter e le Wall-nut sulla griglia (A.4).

Figura A.4: Disposizione piante difensive

- Se uno zombie supera la tua linea difensiva e arriva in fondo la prima volta si attiva un tosaerba che elimina tutti gli zombie su quella riga! (A.5).

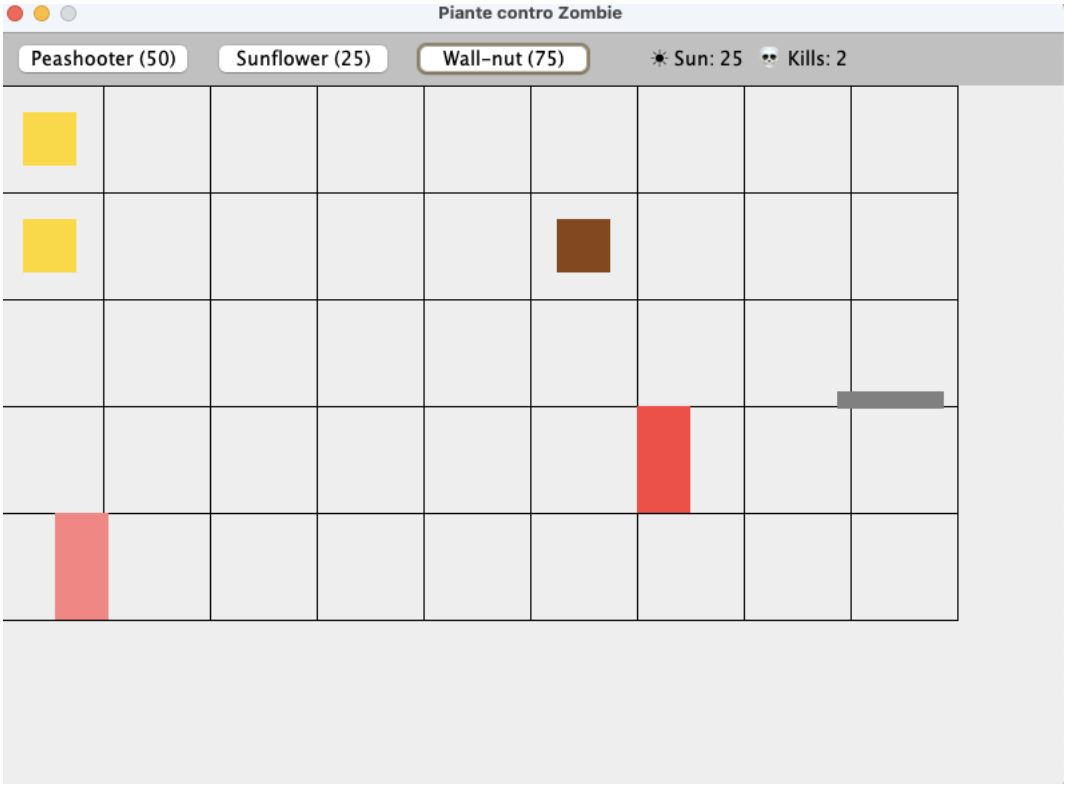


Figura A.5: Attivazione del tosaerba

- Se uno zombie arriva in fondo a una riga senza tosaerba hai perso! (A.6).

Figura A.6: Schermata finale