Michelito El Esqueleto Explosivo

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

Mattia Daviducci
mattia.daviducci@studio.unibo.it

Alessandro Gardini
alessandro.gardini7@studio.unibo.it

Sohaib Ouakani
sohaib.ouakani@studio.unibo.it

Lorenzo Rossi
lorenzo.rossi50@studio.unibo.it

15 febbraio 2025

Indice

Capitolo 1

Analisi

1.1 Descrizione del Software

Il software mira a riprodurre il gioco arcade Bomberman. In questa versione, il protagonista è **Michelito**, che deve muoversi in un labirinto bidimensionale con muri invalicabili, nemici e casse. Michelito può distruggere casse e nemici posizionando bombe che esploderanno dopo pochi secondi, eliminando tutto nel loro raggio d'azione, inoltre le casse distrutte potranno contenere dei potenziamenti in grado di rendere Michelito più forte. La partita consiste in una serie di stanze, con l'obiettivo di trovare una porta nascosta per accedere alla stanza successiva. Michelito ha cinque vite per completare la partita; una volta esaurite, la partita terminerà con un *game over*.

Requisiti funzionali

- All'avvio, l'applicazione mostrerà un menu per iniziare la partita.
- Durante il gioco:
  - Michelito esplorerà un labirinto per trovare la porta per la stanza successiva, distruggendo casse e eliminando nemici lungo il percorso.
  - Il labirinto sarà costituito da una griglia con muri indistruttibili, casse bloccanti e nemici.
  - Le bombe piazzate da Michelito esploderanno, avranno un raggio d'azione limitato e un tempo di detonazione.
  - Michelito potrà attraversare la porta per accedere alla stanza successiva.

  - Il sistema di vite consente a Michelito di ricominciare una stanza dopo ogni vita persa. Esaurite tutte le vite, la partita termina.

- Dopo aver completato tutte le stanze, il giocatore vince.

Requisiti non funzionali

- L'interfaccia deve offrire un'esperienza di gioco intuitiva.

1.2 Modello del Dominio

Il gioco è composto da vari livelli. Ogni livello è composto da un labirinto. Per ogni labirinto il personaggio apparirà in un determinato punto. Il labirinto sarà formato da mura, casse, nemici e una porta. I nemici appariranno in punti specifici e potranno muoversi. Mura e casse non permetteranno il passaggio di Michelito e dei nemici. Michelito potrà piazzare sotto di sé bombe che all'esplosione genereranno fiamme in grado di sconfiggere i nemici e distruggere le casse. Quest'ultime potranno rilasciare power up che se raccolti da Michelito gli forniranno bonus differenti. Ogni labirinto avrà una porta di uscita nascosta sotto una cassa che permetterà l'accesso al livello seguente, solo se tutti i nemici saranno stati eliminati. Il giocatore avrà a disposizione delle vite. Ogni volta che Michelito perderà una vita, il livello verrà resettato e il personaggio tornerà al punto di partenza del livello. Una volta finite tutte le vite il giocatore avrà perso la partita e dovrà ricominciare dal primo livello. Se si riuscirà a completare tutti i livelli il giocatore avrà vinto il gioco.

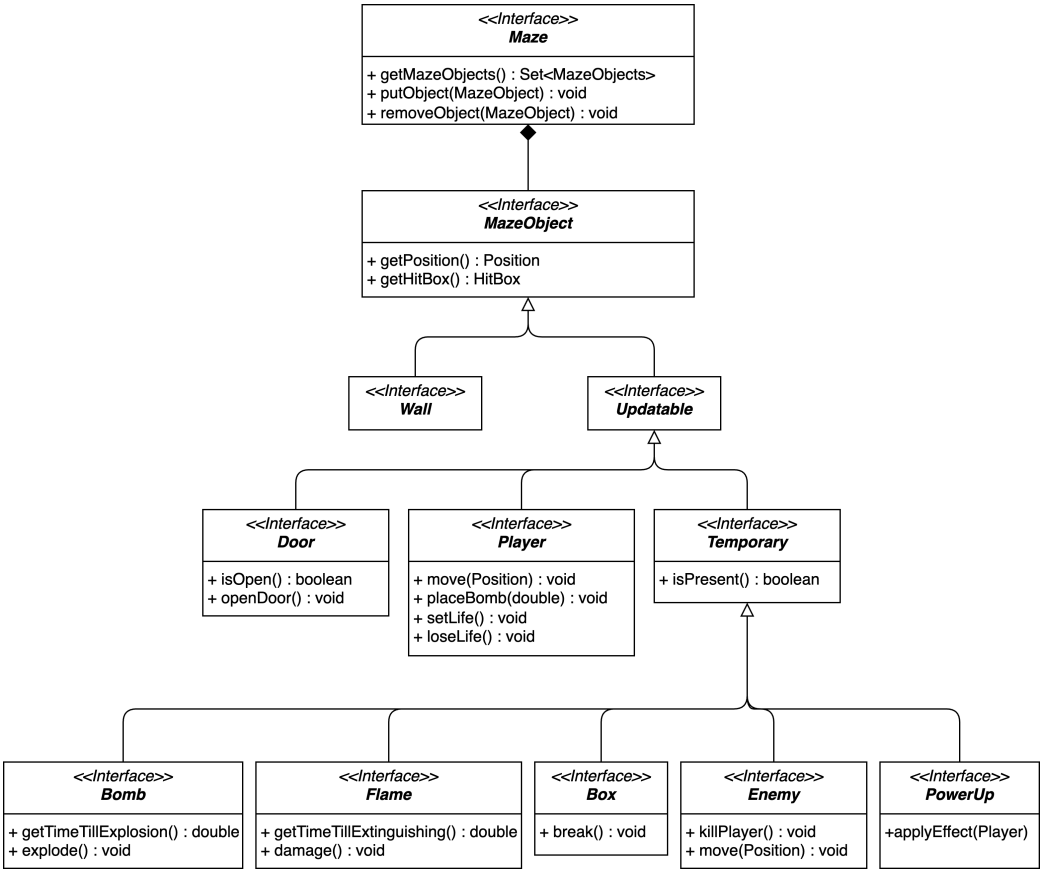

Figura 1.1: Entità della partita

Capitolo 2

Design

2.1 Architettura

L'architettura di Michelito segue il pattern MVC. Abbiamo scelto inoltre di utilizzare diversi controller attraverso l'omonima interfaccia. Questo permette di delegare funzioni diverse a controller specifici. La classe `MainControllerImpl` è il controller principale che gestisce la navigazione dal menù e al gioco vero e proprio. Questa classe istanzia `HomeController` e `GameController`, il primo responsabile del menù principale e il secondo del gioco in sé. Questo Design ci permette di creare piccoli moduli autonomi MVC in grado di gestire i singoli stati garantendo incapsulamento.

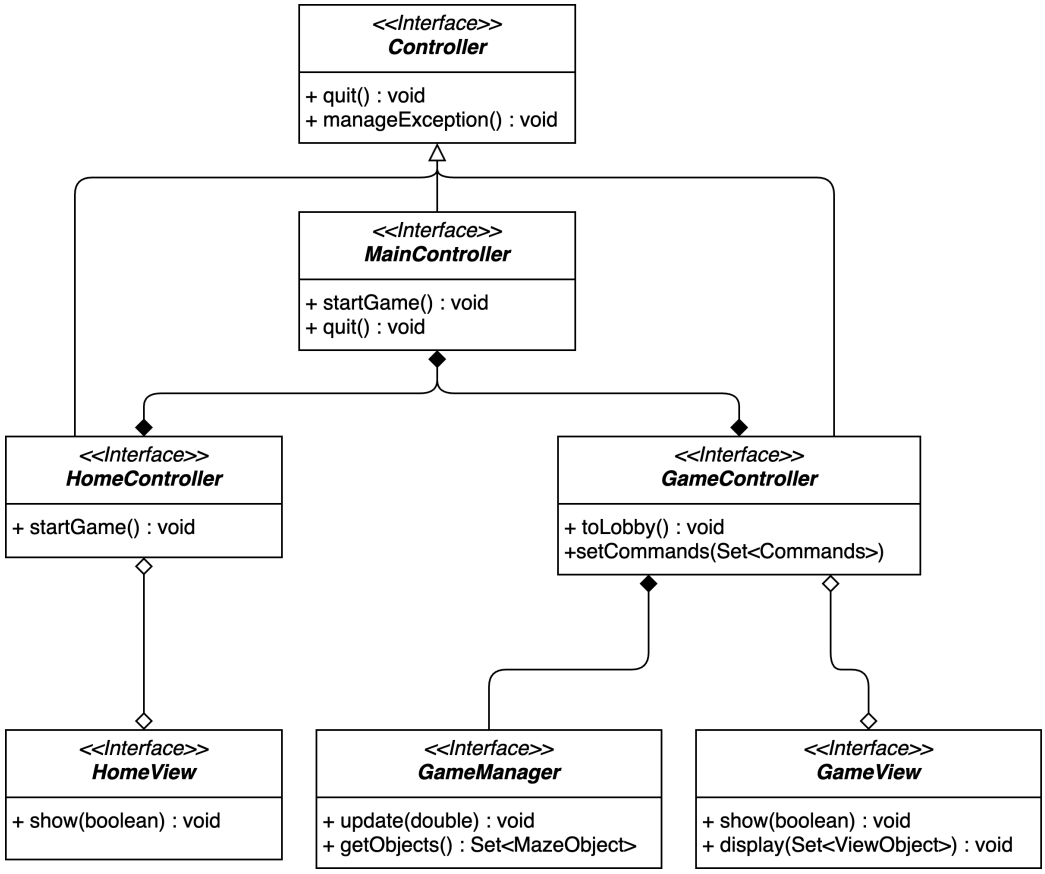

Figura 2.1: Architettura dei controlli

2.2 Design dettagliato

2.2.1 Mattia Daviducci

Gestione delle Bombe

Problema: Nel gioco, le bombe sono un elemento chiave del gameplay e ne esistono di diverse tipologie. Il problema è gestire la loro creazione e comportamento.

Soluzione: Le bombe sono oggetti che, una volta posizionati sulla mappa, hanno un tempo di attivazione, al termine del quale esplodono generando fiamme in base al loro tipo e alle loro proprietà. Devono inoltre interagire con l'ambiente circostante, rimuovendo ostacoli e attivando effetti di gioco.

Per affrontare il problema della loro gestione, ho utilizzato due pattern di progettazione:

- Template Method: Utilizzato nella classe astratta `AbstractBomb` per definire la struttura base di una bomba, delegando il comportamento specifico alle sottoclassi anonime definite in `BombFactory`. Questo pattern consente di centralizzare la logica comune delle bombe, come la gestione del timer e l'esplosione, mentre lascia alle sottoclassi la responsabilità di definire le proprietà specifiche (raggio dell'esplosione e capacità di generare fiamme che attraversano oggetti). Il metodo `generateFlame()` è il Template Method della classe `AbstractBomb`.
- Factory Method: Implementato nell'interfaccia `BombFactory` per creare istanze di bombe in modo centralizzato. È presente una enum `BombType` che definisce le diverse tipologie di bombe disponibili nel gioco. Ogni tipo di bomba viene creato attraverso `BombType` e la posizione.

Pro:

- **Riuso del codice**: la logica comune delle bombe è incapsulata in `AbstractBomb`, evitando duplicazioni.
- **Facilità di estensione**: nuove bombe possono essere aggiunte estendendo la factory e aggiornando `BombType` senza modificare il codice esistente.
- **Separazione delle responsabilità**: la creazione delle bombe è demandata alla `BombFactory`, mantenendo il codice più organizzato.

Contro:

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

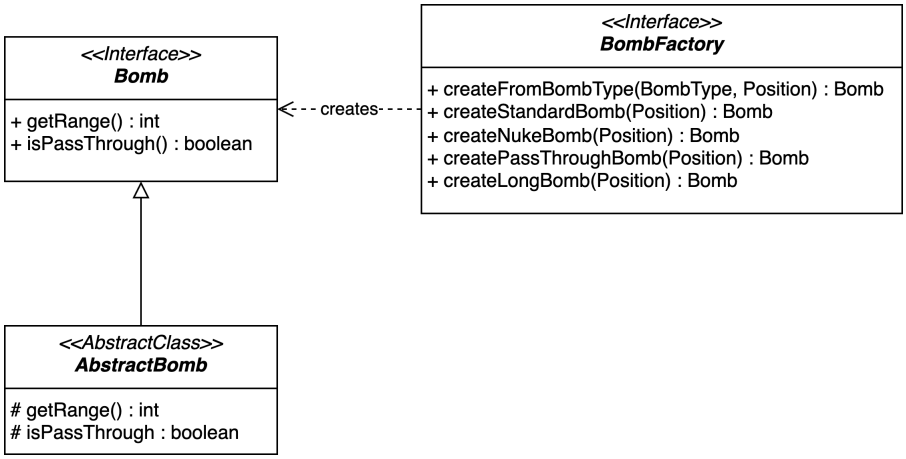

Figura 2.2: Rappresentazione UML dell'organizzazione di Bomb

Gestione delle Fiamme

Problema: Nel gioco, le fiamme si generano quando una bomba esplode e devono propagarsi nel labirinto, interagendo con l'ambiente e i personaggi. Il problema è separare logica di creazione e di propagazione.

Soluzione: Per gestire le fiamme, il codice è stato strutturato separando le interfacce della creazione e della propagazione dalle implementazioni, garantendo una chiara divisione dei compiti. La propagazione è gestita da `FlamePropagation`, che definisce la logica di espansione nel labirinto. `FlamePropagationImpl` calcola il percorso delle fiamme, garantendo che si diffondano correttamente e interagiscano con gli ostacoli, rimuovendo eventuali `Box` colpite. `FlamePropagationImpl` si affida a `FlameFactory` per la creazione delle `Flame` nelle posizioni calcolate. `FlameImpl` gestisce il ciclo di vita delle fiamme e le loro collisioni con le entità di gioco (`Player` e `Enemy`). L'implementazione `FlameFactoryImpl` si occupa di generare nuove fiamme senza esporre i dettagli della loro costruzione.

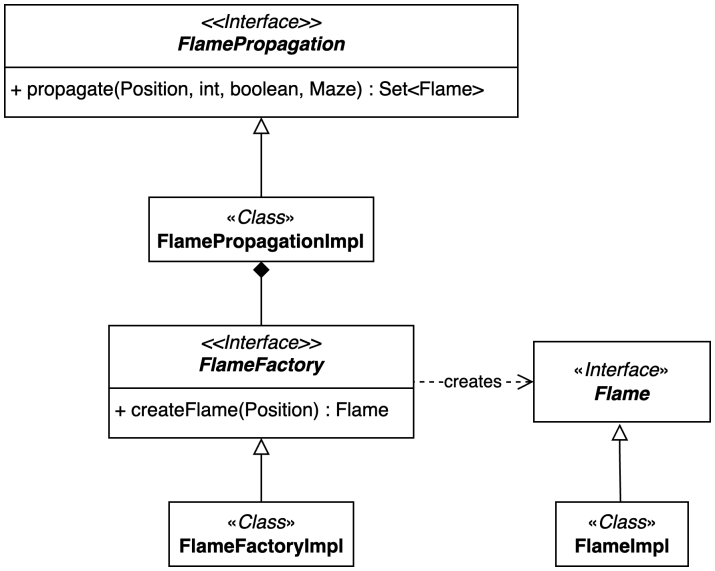

Figura 2.3: Rappresentazione UML dell'organizzazione di Flame

Gestione della Home

Problema: La Home rappresenta il menu iniziale del gioco, permettendo al giocatore di avviare una partita o chiudere l'applicazione. Il problema principale è gestire in modo chiaro la separazione tra logica di controllo e interfaccia grafica.

Soluzione: L'architettura del codice segue il modello VC (View-Controller), garantendo una gestione strutturata della Home e delle sue transizioni di stato. La classe `HomeControllerImpl` implementa le interfacce `HomeController` e `ViewControllerListener`, consentendo un controllo efficace della `HomeView`. L'interfaccia `HomeController`, che estende `Controller`, aggiunge due metodi per la visualizzazione e la rimozione della `HomeView`. `ViewControllerListener`, invece, gestisce il passaggio dallo stato di Home allo stato di Game, assicurando transizioni coerenti e controllate.

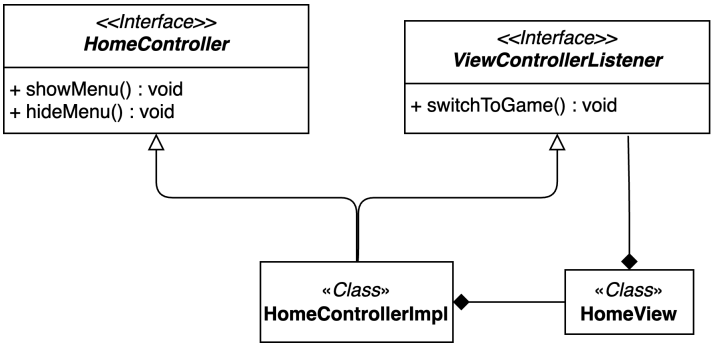

Figura 2.4: Rappresentazione UML dell'organizzazione della Home

2.2.2 Alessandro Gardini

Gestione Controller

Problema: Gestione dei controller subordinati e chiamate al controller principale. Gestione della chiamata di eccezioni da parte dei controller specifici e richiesta di chiusura dell'applicazione.

Soluzione: I controller sono gestiti attraverso una catena di responsabilità (Chain of Responsibility). Questa catena è ottenuta grazie alla creazione dell'interfaccia `Controller` che generalizza le operazioni comuni (`quit` e `handleException`) di ogni controller. Alla creazione di ogni controller figlio, gli verrà passata una copia del padre. Se al figlio viene richiesta la gestione di un'eccezione, questa verrà inoltrata al padre, che ne assumerà la responsabilità.

La soluzione proposta prevede di definire anche un meccanismo per gestire il flusso di controllo tra i diversi stati dell'applicazione (*Home*, *Game*). Il meccanismo è ottenuto tramite la creazione delle interfacce `GameParentController` e `HomeParentController`. Queste interfacce vengono passate rispettivamente ai controller secondari, `GameController` e `HomeController`, i quali potranno richiedere al `MainController` solo cambi di stato coerenti a quello attuale. Un'ulteriore interfaccia `MainController` gestisce solo la relazione con il Game Launcher (`Michelito`).

Pro: La soluzione propone una chiara separazione delle responsabilità (*SRP*) tra controller specifici e principali, evita la duplicazione del codice (*DRY*) attraverso un'interfaccia comune, e garantisce maggiore possibilità di estensione grazie alla *chain of responsibility*, permettendo l'aggiunta di nuovi controller senza modifiche al codice esistente.

Contro: La soluzione introduce una maggiore complessità a causa della *chain of responsibility* e delle interfacce multiple.

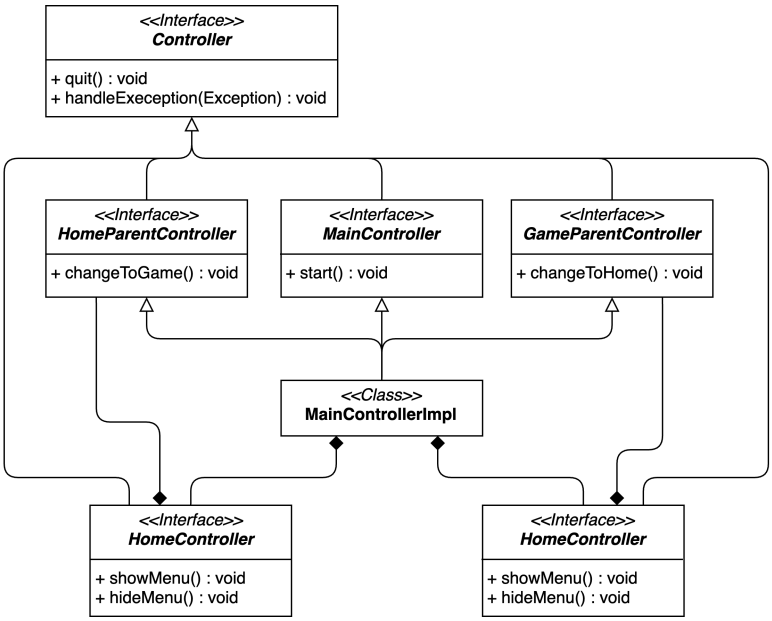

Figura 2.5: MainControllerImpl rappresenta il controller principale, mentre HomeController e GameController rappresentano i secondari

Game Manager

Problema: Interazione con il modello di gioco tramite interfaccia minimale, esponendo solamente le funzionalità minime.

Soluzione: L'interazione con il modello viene mediata attraverso l'entità `GameManager`. Quest'ultima ha il compito di mantenere lo stato globale della partita, di aggiornare il livello corrente e fornire all'esterno il suo stato. Ha anche il compito di gestire il cambiamento dei livelli, modellandone la successione, e il *game over*, sia nel caso in cui il `player` abbia vinto, sia nel caso in cui abbia perso l'ultima vita a sua disposizione.

Il livello è modellato dalla classe `MazeImpl` che implementa l'interfaccia `Level`. Questa espone solamente le funzionalità di aggiornamento degli oggetti, il passaggio di un comando al `player` e metodi per consultarne lo stato.

Pro: Il `GameManager` separa la gestione dello stato globale del gioco dalla logica dei livelli (*SRP*), migliorando la modularità, mentre l'interfaccia mi-

nimale e la struttura semplice rendono il sistema facile da comprendere e mantenere (*KISS*).

Contro: Il `GameManager` diventa un punto centrale di dipendenza, il che potrebbe rendere il sistema meno flessibile in caso di modifiche future.

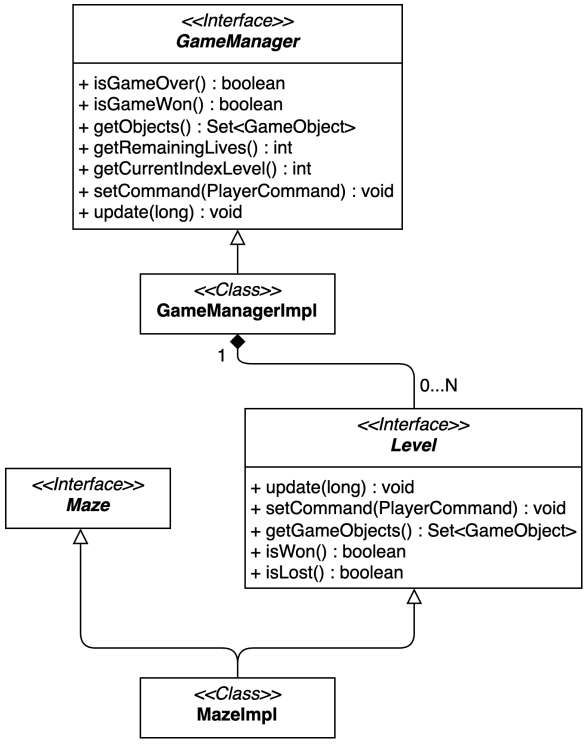

Figura 2.6: Relazione di GameManager e Level

Categorizzazione oggetti

Problema: All'interno del `Maze` dobbiamo gestire il fatto che ci siano entità che possano essere rimosse, entità statiche immutabili ed entità che cambiano di stato, o una combinazione di queste caratteristiche.

Soluzione: Creazione di un'interfaccia comune `MazeObject`, che rappresenta un generico oggetto all'interno del `Maze`. Tutti gli oggetti appartenenti al `Maze` hanno una `Position`, che rappresenta la sua posizione nello spazio, una `HitBox`, che rappresenta lo spazio occupato, e un `Type`, che rappresenta la sua categoria.

In base a specifiche caratteristiche che possono avere le diverse entità vengono create le interfacce `Temporary` e `Updatable`. Gli oggetti `Temporary`

possono essere tolti dal `Maze` perché temporanei, ad esempio un nemico che può essere sconfitto. Gli oggetti `Updatable` possono subire un cambio di stato sotto forma di aggiornamento. Ogni oggetto `Updatable` ha un metodo `update` che riceve in ingresso due informazioni: il delta di tempo passato dall'ultima chiamata e un riferimento al `Maze`, con il quale potrà modificare lo stato della partita.

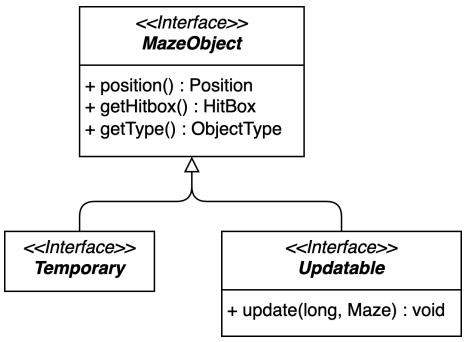

Figura 2.7: Rapporti tra MazeObject, Temporary e Updatable

Maze

Problema: Gestire tutti gli oggetti MazeObject con eventuali rimozioni e/o aggiornamenti. Mantenere lo stato del player.

Soluzione: `MazeImpl`, oltre a `Level`, implementa anche `Maze`. Questa interfaccia fornisce funzioni per recuperare i vari tipi di oggetti presenti nel livello corrente e settare lo stato "vittoria" e "sconfitta".

Durante l'aggiornamento di `MazeImpl`, ogni oggetto `Updatable` viene aggiornato tramite il metodo `update`. La divisione di `MazeImpl` in due interfacce permette di separare l'accesso all'oggetto, lasciando esposti metodi specifici in base alle due situazioni. I metodi dell'interfaccia `Maze` sono accessibili dai vari `MazeObject`, mentre i metodi di `Level` sono chiamati dal `GameManager`.

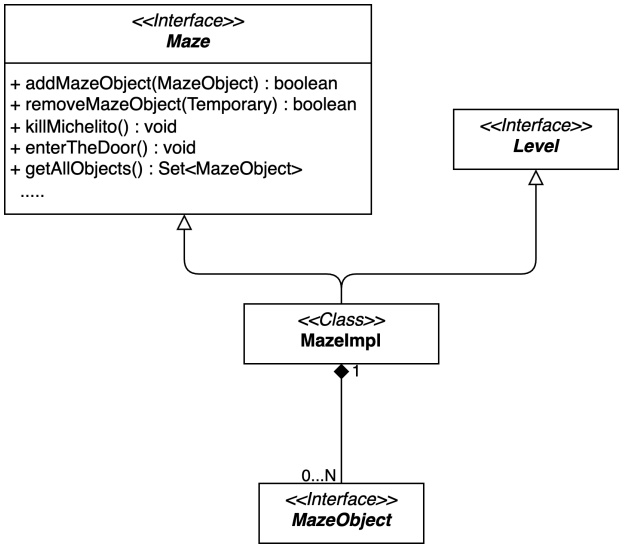

Figura 2.8: MazeImpl composto da MazeObject e la rappresenazione delle sue due interfacce

Adattatore oggetti

Problema: Il `LevelGenerator`, su richiesta del `Maze`, produce `GameObject` che devono essere tradotti in `MazeObject`.

Soluzione: Creazione di un `ObjectsAdapter`. Questa interfaccia definisce un metodo che accetta come parametro l'indice del livello e restituisce il set di `MazeObject` corrispondenti al livello specificato. È stata creata una `ObjectsAdapterImpl` che utilizza la funzione specificata nel suo costruttore come funzione di mapping da `GameObject` a `MazeObject`. Utilizzando il pattern *Proxy* è stata creata una seconda implementazione,
`ObjectsAdapterWithCache`, che salva i `GameObject` di ciascun livello, una volta caricato, in una memoria cache. La creazione dell'adapter avviene tramite una *Static Factory* che permette trasparenza rispetto al tipo di adapter.

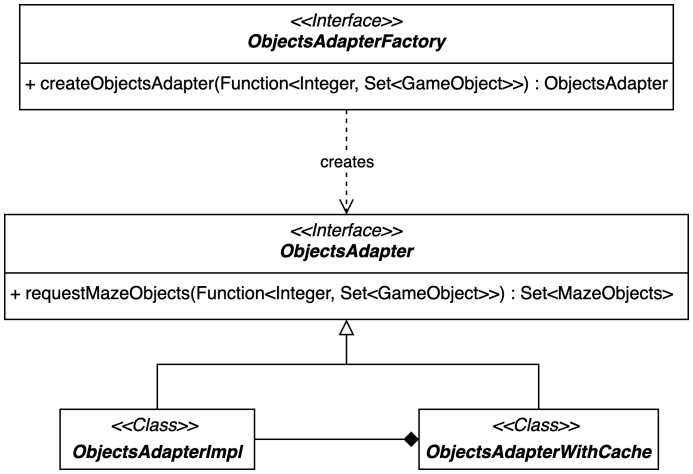

Figura 2.9: UML Pattern Adapter, Proxy e Factory rispetto la soluzione

2.2.3 Sohaib Ouakani

Organizzazione del Player

Problema: Gestire il movimento del player, la sua hitbox e il piazzamento delle bombe.

Soluzione: Separazione delle responsabilità attraverso componenti specifici che vengono composti all'interno di `PlayerImpl`. `MovementComponent` è l'interfaccia che modella il componente responsabile del movimento del player, `HitboxComponent` invece gestisce aspetti legati alle collisioni, infine il `BombManagerComponent` è il componente responsabile del piazzamento delle bombe.

Pro:

- Aderenza ai principi SOLID, in particolare al *Single Responsibility Principle*, separando le responsabilità in classi distinte.
- Approccio ispirato al *Strategy Pattern*, che favorisce la composizione rispetto all'ereditarietà.
- Maggiore modularità, consentendo la sostituzione o estensione di singoli componenti senza impattare l'intera implementazione del player.

Contro:

- L'aggiunta di nuove funzionalità richiede la modifica del player per esporre i metodi necessari all'interazione con nuovi componenti.

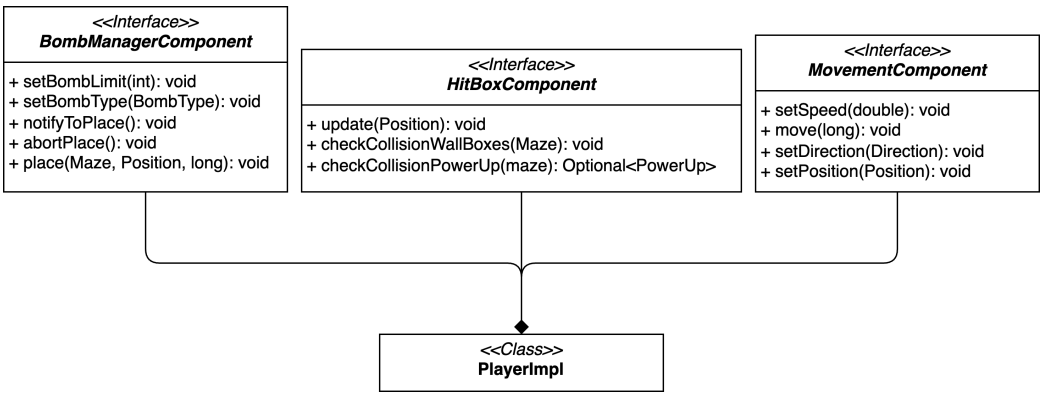

Figura 2.10: Rappresentazione UML dell'oragizzazione del player attraverso la composizione di vari componenti

Gestione Power-Up

Problema: Creare un'architettura scalabile e flessibile per i power-up.

Soluzione: Definizione di un'interfaccia `PowerUp` che implementa `Temporary`. L'interfaccia `PowerUp` definisce un metodo per applicare l'effetto del power-up al player. Il codice ridondante nelle varie implementazioni di `PowerUp` è stato accumulato in una classe astratta `AbstractPowerUp`, rispettando così il principio DRY.

Per semplificare la creazione di power-up, ho deciso di utilizzare il *Factory Method Pattern*, attraverso l'interfaccia `PowerUpFactory`, la quale fornisce un metodo per creare power-up in base a un'enumerazione dei tipi disponibili.

Pro:

- Evita la duplicazione del codice, migliorando la manutenibilità.
- Permette un'implementazione più chiara e modulare dei diversi tipi di power-up.
- Incapsulamento dell'implementazione dei power-up, permettendo la loro creazione esclusivamente tramite la factory.

Contro:

- L'uso di un `enum` per identificare i tipi di power-up può ridurre la flessibilità rispetto a una soluzione basata su classi e registrazione dinamica.

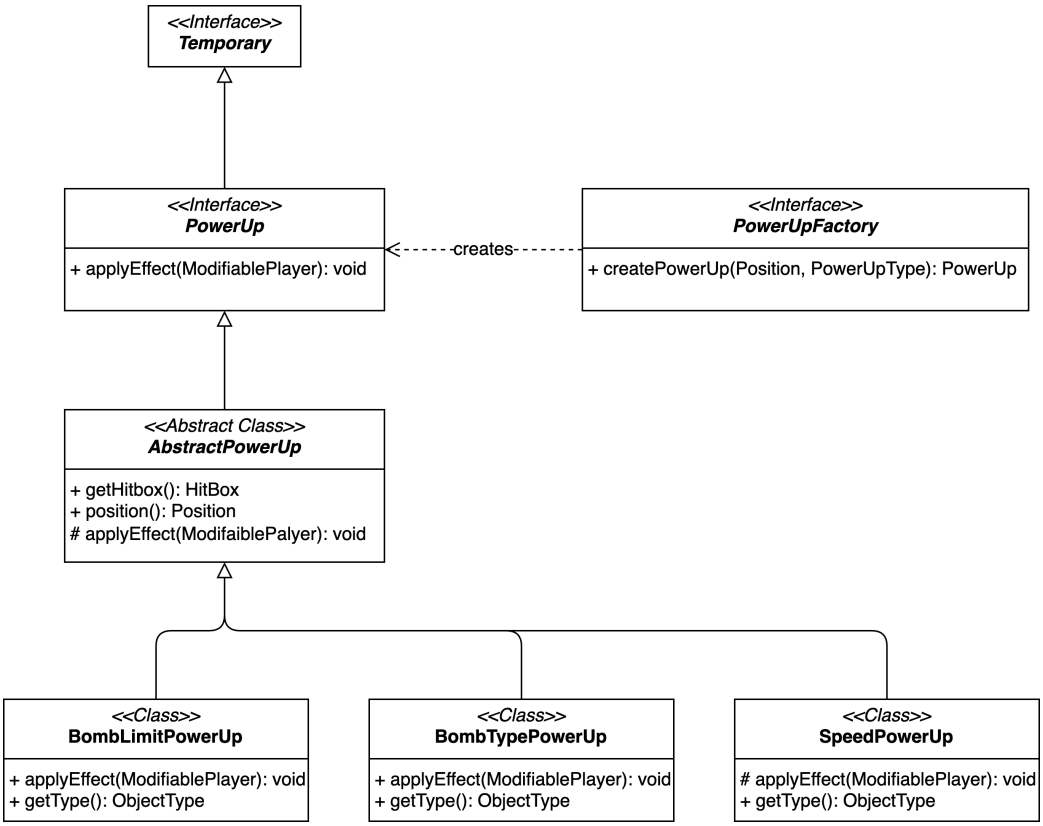

Figura 2.11: Rappresentazione UML della struttura dei PowerUp e del pattern Factory Method

Interazione Power-Up e Player

Problema: È necessario che i power-up possano modificare le caratteristiche del player senza avere accesso diretto ai metodi che aggiornano il suo stato e senza poter impartire comandi diretti al player.

Soluzione: Creazione dell'interfaccia `ModifiablePlayer`, implementata da `PlayerImpl`. Questa interfaccia consente ai power-up di interagire con il player limitandosi ai metodi di modifica delle caratteristiche, senza alterare direttamente il suo stato.

Pro:

- Migliora l'incapsulamento, permettendo a ciascuna classe di interagire con `PlayerImpl` solo secondo le proprie necessità.

Contro:

- L'aggiunta di `ModifiablePlayer` introduce un livello di complessità, poiché è necessario dichiarare esplicitamente tutti i metodi di modifica all'interno di questa interfaccia.

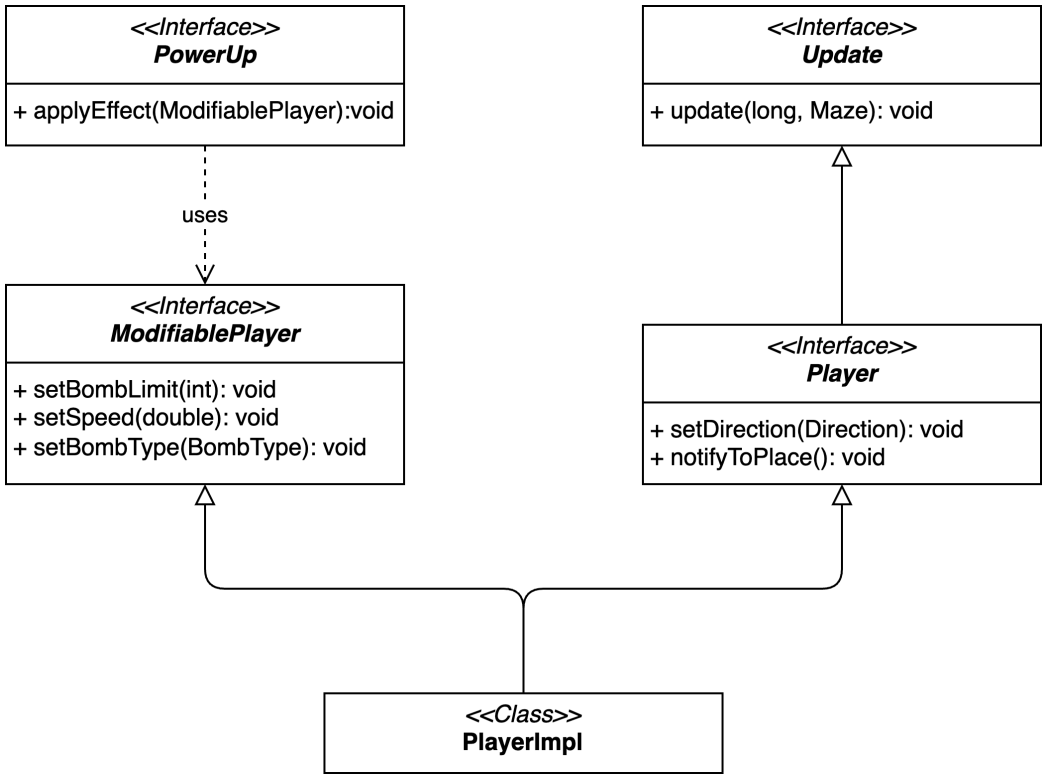

Figura 2.12: Rappresentazione UML delle interfacce implementate da PlayerImpl e relazione tra PowerUp e ModfiablePlayer

Gestionde del Game Loop

Problema: Realizzazione di un Game Loop.

Soluzione: Il Game Loop è stato realizzato nella classe `GameController` attraverso la classe innestata `Loop` che estende `Thread`. Qui viene coordinato l'aggiornamento della `GameView` all'aggiornamento del `Model`, gestendo il ritorno alla home in caso di vittoria, sconfitta o chiusura del pannello di gioco.

Pro:

- Mantiene il gioco responsivo anche in caso di rallentamenti da parte del modello.

Contro:

- Lavorare con i `Thread` aumenta di molto la complessità.

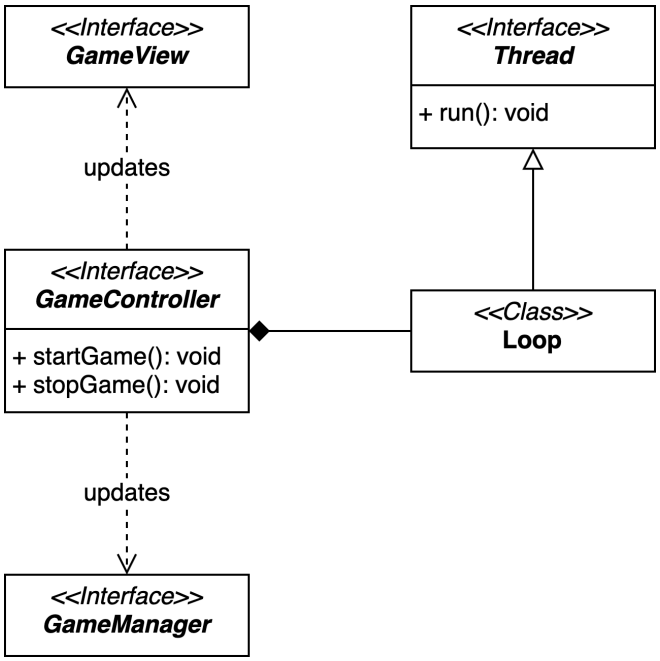

Figura 2.13: Rappresentazione UML dell'interazione tra il GameController, composto dalla classe Loop, GameView e GameManager

2.2.4 Lorenzo Rossi

Gestione delle Collisioni

Problema: Gli oggetti dentro il labirinto devono possedere una fisicità per la gestione delle interazioni con l'ambiente.

Soluzione: La soluzione scelta comprende l'utilizzo di un oggetto `Hitbox`, che, data la posizione centrale, possa essere utilizzato per rappresentare la forma fisica dell'oggetto. Questo permette di determinare eventuali collisioni con altri oggetti o la presenza di punti all'interno dello spazio occupato dall'oggetto.Per permettere una più completa rappresentazione degli oggetti ed eventuale espandibilità futura, si è scelto di implementare una `HitboxFactory`. Questa restituisce tipi diversi di `Hitbox` che seguono un template generale comprendente i metodi comuni, ma che espandono metodi astratti in modo da permettere caratteristiche diverse.

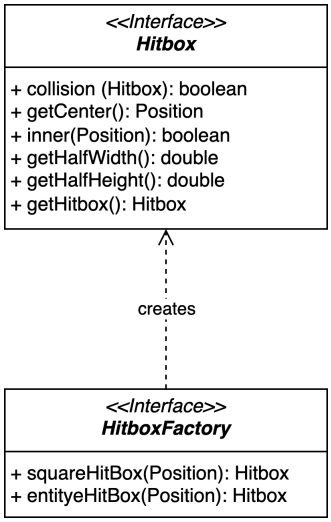

Figura 2.14: Composizione di Hitbox

Struttura base nemici

Problema: Modellazione dei nemici. I nemici si devono muovere autonomamente nel labirinto col passare del tempo senza transitare sopra a muri o casse, uccidendo il giocatore a contatto, continuando a mantenere aggiornati la loro posizione e la loro hitbox.

Soluzione: La soluzione proposta comprende lo scorporamento delle funzioni del `Enemy` in componenti dedicati quali `MoodAI` e `Movement`, affidando al secondo le funzioni di movimento del `Enemy` assicurando solo spostamenti in posizioni accettate dal `Maze` (posizioni non sovrapposte ad altri oggetti), lasciando ad `Enemy` i controlli riguardo alla collisione con il `Player` e l'aggiornamento della propria `Hitbox`. Questa scelta permette :

- Migliore leggibilità di codice rendendo più chiara l'implementazione di `Enemy`

- Maggiore modularità che permette modifiche a singole funzioni di `Enemy` senza impattare l'intero sistema.

Pro: Separazione delle responsabilità secondo il `Single Responsibility Principle` e maggiore leggibilità del codice.

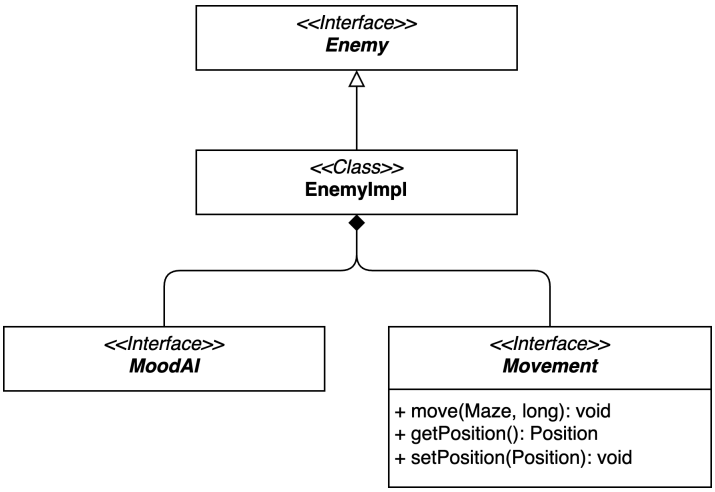

Figura 2.15: Divisione dei componeti di Enemy

Comportamento dei nemici nel tempo

Problema: Rendere le interazioni dei nemici più dinamiche in base ai cambiamenti nel livello.

Soluzione: Per risolvere questo problema, è stato introdotto il componente `MoodAI`, che viene aggiornato direttamente dal nemico e determina il tipo di movimento da adottare in base alle condizioni del labirinto e al tempo di gioco.Il `MoodAI` seleziona i movimenti da una *factory*, che fornisce diverse varianti basate su un *template* comune ma con caratteristiche uniche. Questo approccio consente ai nemici di modificare il proprio comportamento nel corso della partita, lasciando aperta però la possibilità di estensioni. Tutti i nemici condividono la stessa logica riguardante i cambiamenti di *mood*, ma non essendo una AI centralizzata ogni `Enemy` può valutare per sé stesso cambiamenti di stato. Ciò permette una reazione indipendente dagli altri.

Contro: I nemici condividono la stessa logica dell'AI; questo complica la creazione di nemici con AI uniche.

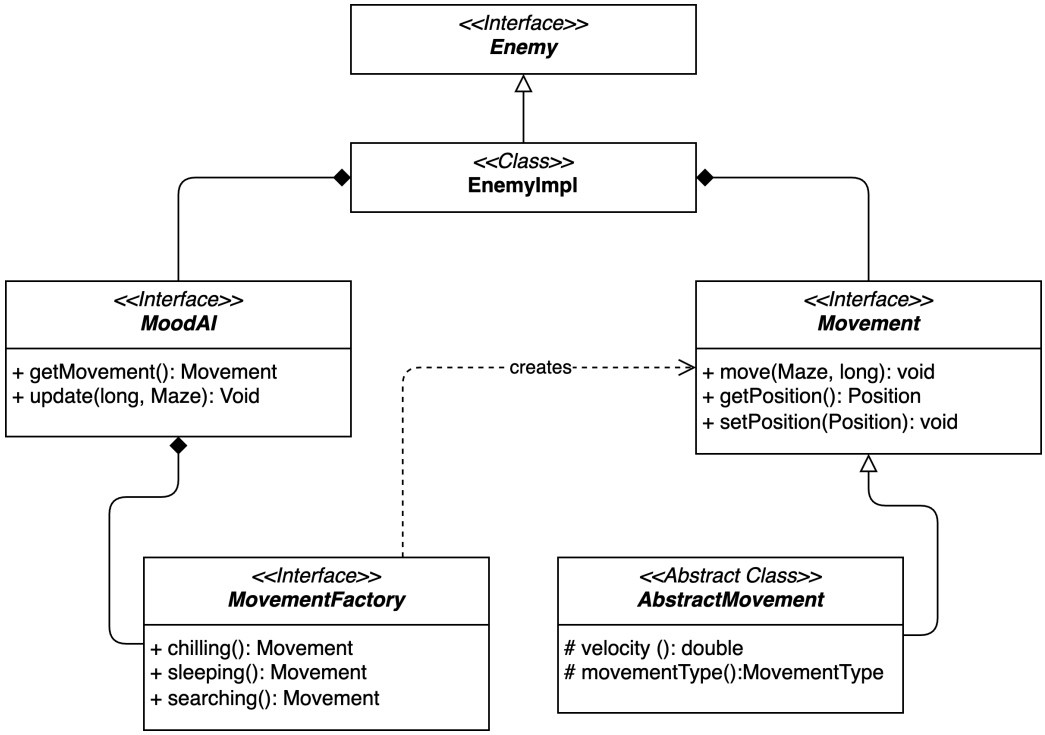

Figura 2.16: Rappresentazione UML dell'oragizzazione delle interfacce legate a MoodAI

Creazione livelli

Problema: Si è scelto di avere livelli sempre uguali nel gioco e che il `Player` debba affrontarli in un determinato ordine. Serve un modo per fornire al gioco questi livelli.

Soluzione: Si è scelto di salvare i livelli su file. Attraverso un `LevelGenerator`, al quale si richiede il livello desiderato, si può leggere il file corrispondente e trasformare i dati ottenuti in una collezione di `Gameobject`, contenente tutti gli oggetti presenti dentro il livello del `Maze`. Questi oggetti vengono poi utilizzati per costruire il livello di gioco. `LevelGenerator` implementa `Function`, permettendo che al modello venga passata una strategia, che usa inconsapevolmente le funzioni di `LevelGenerator`.

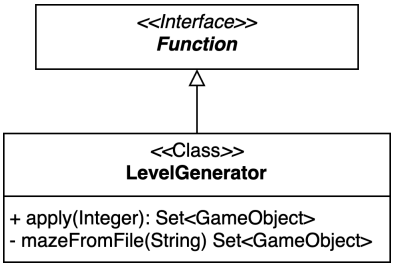

Figura 2.17: Composizione del LevelGenerator

Capitolo 3

Sviluppo

3.1 Testing automatizzato

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

- BlankSpace, Box e Wall sono stati testati nelle funzionalità di ogni MazeObject.
- È stata testata l'abilitá della door di aprirsi una volta sconfitti tutti i nemici.
- Di MazeImpl sono stati testati i metodi di entrambe le interfacce, la rimozione e l'inserimento di oggetti e il throw di eccezzioni.
- Del GameManager è stato testato il corretto stato iniziale.
- Hitbox sono state testate nelle varie possibili combinazioni e direzioni di collisione.
- Enemy è stato testato sia nello spostamento che nel cambiamento di "mood".
- LevelGenerator è stato testato nella sua corretta generazione di livelli di test, nell corretto posizionamento dei wall e la presenza del Player e della Door.
- Di Player si è testato il movimento e la collisione con Wall e Box.
- PowerUp con tutte le sue implementazioni, quindi testate tutte le classi BombLimitPowerUp, SpeedPowerUp e BombTypePowerUp.

- PowerUpFactory si è testata la corretta creazione di PowerUp.
- Di MoveCommandBuilder si è testato la corretta costruzione di comandi in base alle Direction passate.
- Di Bomb è stata testata la sua corretta creazione, la sua abilità di esplodere e di generare la prima fiamma nella sua posizione.
- Di Flame è stata testata la sua corretta creaezione, il suo ciclo di vita, la sua espansione nelle direzioni giuste e nella corretta quantità, la sua abilità di eliminare i nemici, il player e le box in caso di collisione.

3.2 Note di sviluppo

3.2.1 Mattia Daviducci

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

Utilizzo di Optional Un esempio in questo permalink.

Utilizzo libreria Math Un esempio in questo permalink.

3.2.2 Alessandro Gardini

Utilizzo di Lamba expression, Stream, Optional Un esempio in questo permalink.

Utilizzo di reflection e generici Utilizzati in un medoto per evitare repetizione di codice, qui il permalink.

3.2.3 Sohaib Ouakani

Utilizzo di Lambda Expressions, Stream e Optional Utilizzate in vari punti, ecco un esempio: permalink.

Utilizzo reflection Utilizzato nel test di FactoryPowerUp, permalink.

Utilizzo libreria Math Utilizzata in diverse parti di codice, qui un esempio: permalink.

3.2.4 Lorenzo Rossi

Utilizzo di Stream e Lambda expression Usate ripetutamente . Il seguente è un singolo esempio presente in questo permalink.

Utilizzo libreria Math Un esempio in questo permalink.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Mattia Daviducci

Non avevo mai lavorato a un progetto di gruppo di questa portata prima d'ora e devo dire che l'esperienza è stata estremamente formativa. Mi sono trovato molto bene con il team: nonostante l'impegno richiesto fosse notevole, siamo riusciti a organizzarci in maniera efficace, dividendo i compiti in modo equo e collaborando costantemente per raggiungere gli obiettivi prefissati. Sono molto soddisfatto del risultato finale: l'applicazione non solo funziona come sperato, ma rispecchia anche la cura che tutti abbiamo dedicato alla parte di analisi e all'architettura. A livello personale, sento di aver messo in luce alcuni punti di forza, come la mia determinazione nel trovare soluzioni ai problemi che si sono presentati e la buona conoscenza del gioco originale, che mi ha aiutato a orientarmi meglio nelle varie fasi di sviluppo. D'altra parte, ho anche potuto constatare due punti deboli su cui lavorare: la mia scarsa esperienza iniziale nell'utilizzo di Git e le mie abilità di gestione di un lavoro di gruppo, che all'inizio hanno richiesto un certo sforzo di adattamento. Nel complesso, però, mi sento davvero soddisfatto; credo che, in futuro, potrei tornare a mettere mano al progetto, sia per perfezionare alcune parti di codice sia per integrare ulteriori idee proposte dal gruppo ma non ritenute essenziali in questa prima versione. È stata un'esperienza impegnativa ma molto arricchente, che mi ha offerto la possibilità di crescere, sia tecnicamente che a livello di collaborazione e gestione del lavoro in squadra.

4.1.2 Alessandro Gardini

Questo progetto si è rivelato una sfida stimolante e produttiva. Abbiamo avuto l'opportunità di lavorare in gruppo e sviluppare, a partire dai concetti appresi a lezione, un progetto che, pur nella sua semplicità, rappresenta un primo passo concreto verso la realizzazione di applicazioni più complesse e strutturate. Dal punto di vista personale, questa esperienza mi ha permesso di consolidare le mie competenze tecniche e migliorare le capacità di collaborazione. Ogni fase del mio lavoro è stata svolta a stretto contatto con il resto del gruppo, e questo mi ha permesso di capire quanto il lavoro di squadra possa facilitare la comprensione della materia. Spero di poter continuare a lavorare su questo progetto insieme ai membri del gruppo, apportando piccoli miglioramenti alla mia parte.

4.1.3 Sohaib Ouakani

E' stato molto interessante lavorare a questo progetto, per via delle difficoltà superate durante lo sviluppo, permettendomi di affinare le mie capacità di lavorare in gruppo. Inoltre la parte individuale ha messo in evidenza alcune lacune su cui ho potuto poi migliorare, cosa che mi ha fatto molto piacere. Ho trovato una buona intesa con i membri del gruppo, questo ci ha permesso di essere molto efficienti e di ottenere i risultati prefissati nei tempi previsti. Mi piacerebbe portare avanti questo progetto per poter implementare idee scartate per mancanza di tempo, oppure per poter rivisitare e migliorare alcune parti. In conclusione ho apprezzato molto il modo in cui questo esame ci ha messi alla prova, e spero di poter fare esperienze simili durante il mio percorso.

4.1.4 Lorenzo Rossi

Ritengo che questo progetto sia stata una sfida interessante che ti spinge ad utilizzare le conoscenze apprese a lezione e svilupparle in un ambiente più pratico. La mia parte riguardava lo sviluppo dei nemici, delle collisioni e del level generator le quali, soprattutto le ultime due, richiedevano un'organizzazionee più serrata con elementi del gruppo che si occupavano di parti collegate. Le sfide maggiori di questo progetto sono state l'utilizzo corretto di git, la giusta rappresentazione della risoluzione dei problemi nel report e l'utilizzo delle librerie grafiche. Ritengo che porterò avanti il progetto come prova personale per sperimentare varie idee che sono venute al gruppo alla fine del lavoro o l'aggiunta di nuove funzionalità ritenute secondarie durante l'ideazione.

Appendice A

Guida utente

Breve spiegazione dei comandi e delle dinamiche del gioco:

- Il menù (A.1) principale permette di avviare la partita (A.2).

Figura A.1: Menù

- Il movimento di Michelito (rettangolo verde) è ottenuto grazie alle frecce direzionali e piazzare bombe con la barra spaziatrice.
- Per superare ogni livello è necessario sconfiggere tutti i nemici (rettangoli rossi) e trovare la porta (rettangolo viola) nascosta sotto una box (rettangoli azzurro).
- Rompendo le box si possono trovare potenziamenti (cerchi piccoli colorati) di vario tipo. Giallo per un aumento di velocità. Rosa per vari potenziamenti del raggio della bomba. Blu per ottenere l'abilità di piazzare più bombe.

Di seguito una serie di immagini per aiutare la comprensione:

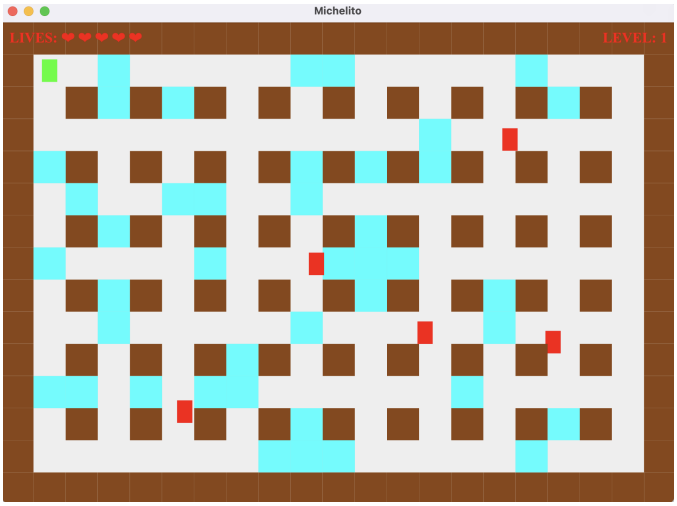

Figura A.2: Inizio, Michelito è il rettangolo verde

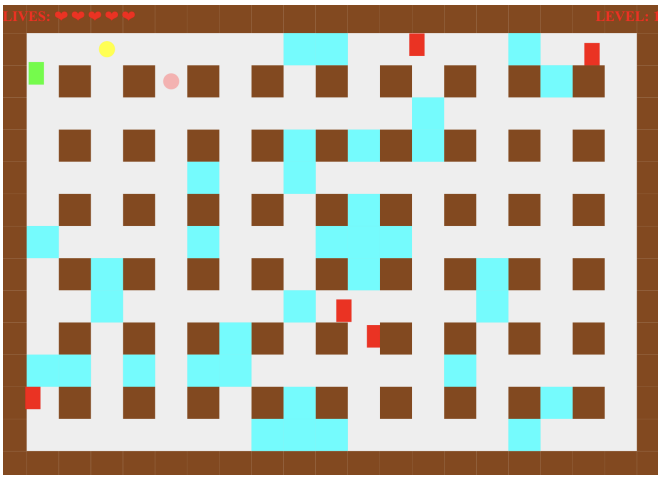

Figura A.3: Presenza dei due potenziamenti (velocità sopra, raggio bomba a destra)

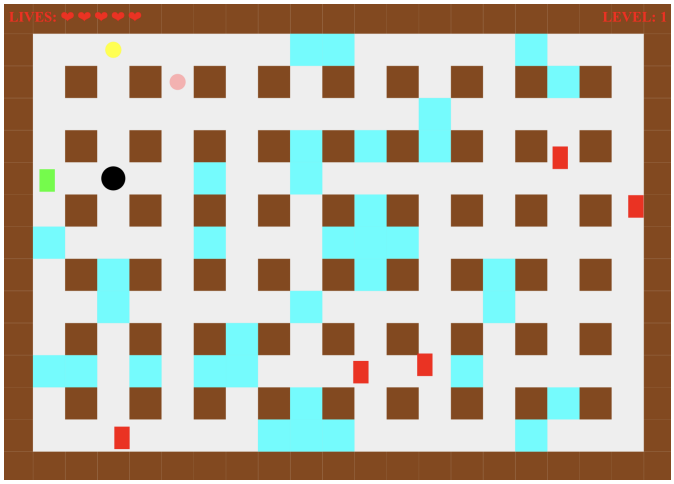

Figura A.4: Michelito ha piazzato una bomba, in questo momento alla sua destra

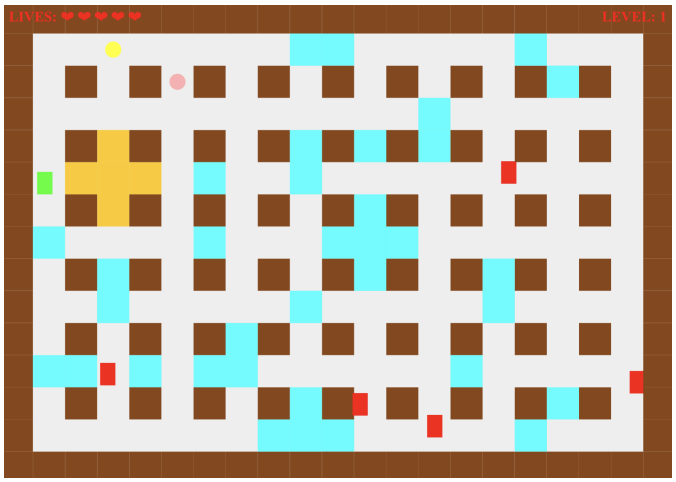

Figura A.5: la bomba precedentemente piazzata è esplosa generando fiamme introno a se

Appendice B

Esercitazioni di laboratorio

B.0.1 sohaib.ouakani@studio.unibo.it

- Lab 09
- Lab 10
- Lab 11