Elements Duo

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

Distefano Stefano
stefano.distefano3@studio.unibo.it

Ferri Alice
alice.ferri8@studio.unibo.it

Kadiu Christian
christian.kadiu@studio.unibo.it

Meloni Leonardo
leonardo.meloni@studio.unibo.it

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il software consiste in una versione rivisitata del gioco *Fireboy and Watergirl*. Il gioco prevede due personaggi giocabili, ispirati agli elementi fuoco e acqua. Questi due personaggi possono essere controllati da due giocatori diversi in modalità cooperativa locale, oppure dallo stesso giocatore che gestisce in parallelo entrambi i personaggi.
L'obiettivo di ciascun livello è raggiungere l'uscita, con entrambi i personaggi, superando un percorso composto di ostacoli e nemici. Il completamento del livello è possibile solo attraverso la cooperazione tra i due personaggi.
Il livello può presentare zone pericolose specifiche per ciascun personaggio. L'idea è che il giocatore associato al fuoco sia resistente al proprio elemento, ma vulnerabile ad altri, come l'acqua o le zone verdi; allo stesso modo, il personaggio associato all'acqua sarà resistente all'acqua ma vulnerabile al fuoco ed alle zone verdi. In questo modo, la mappa è progettata per costringere i giocatori a usare le capacità di entrambi. Il gioco comprende tre livelli che introducono gradualmente ostacoli più complessi e sfide che richiedono maggiore coordinazione tra i due personaggi.

Requisiti funzionali

- Il giocatore deve poter controllare i personaggi principali in modo indipendente.
- I personaggi devono essere in grado di saltare e muoversi all'interno della mappa.
- Il gioco deve gestire correttamente le collisioni tra i personaggi e l'ambiente di gioco.

- Devono essere presenti nemici e ostacoli ambientali. Il contatto con questi elementi deve comportare il riavvio del personaggio dalla posizione iniziale del livello.

- Devono essere ben definiti un inizio e una fine del gioco.

Requisiti non funzionali

- Deve essere presente un menù di gioco, dal quale sia possibile avviare una nuova partita o uscire.

1.2 Modello del Dominio

Il gioco è composto da tre livelli, ciascuno popolato da diverse entità. I giocatori, Fireboy e Watergirl, vengono posizionati in un punto specifico della mappa e il loro obiettivo è raggiungere, collaborando, le rispettive uscite.

Durante il percorso incontreranno diversi avversari, inclusi nemici di pattuglia e "bersaglieri" capaci di sparare, che potranno essere eliminati saltandoci sopra. Per avanzare, i giocatori dovranno usare con astuzia gli elementi della mappa: troveranno casse da spostare per raggiungere punti altrimenti inaccessibili, oppure bottoni e leve che attivano piattaforme mobili.

A ostacolarli ci saranno anche pozze di diverso tipo: acqua (percorribile solo da Watergirl), fuoco (percorribile solo da Fireboy) e acido (letale per entrambi). I giocatori hanno infine la possibilità di collezionare le gemme che trovano lungo il tragitto; queste, insieme all'eliminazione di tutti i nemici, serviranno a completare una missione che attesta il completamento del livello al 100%.

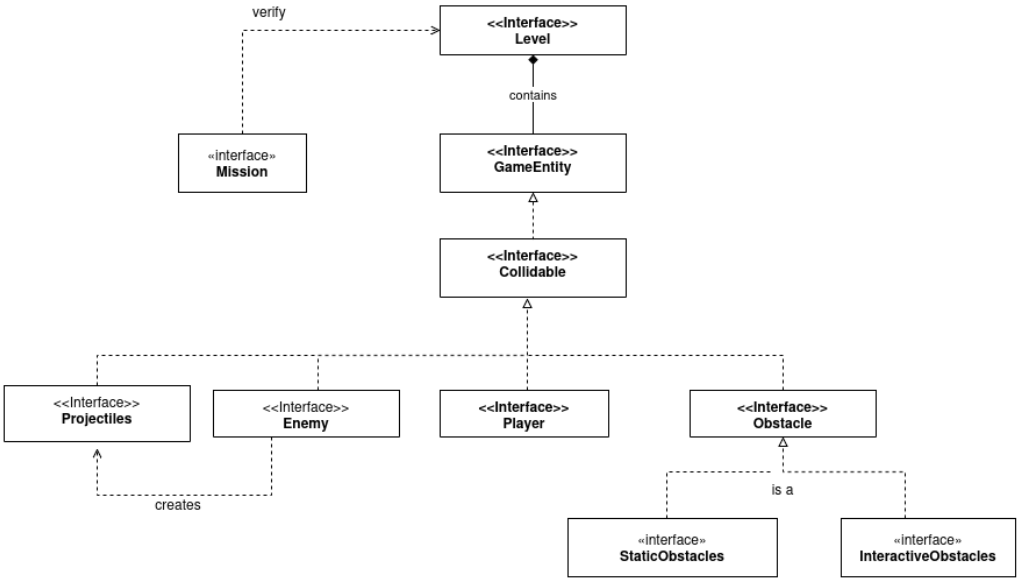

Figura 1.1: descrizione dell'entità di gioco.

Capitolo 2

Design

2.1 Architettura

L'applicazione *Elements Duo* adotta il pattern architetturale MVC.
Questo consente di separare in modo chiaro la logica di gioco, la gestione degli input e la rappresentazione grafica. All'interno del `Model` sono contenuti tutti gli elementi fondamentali del gioco, come i personaggi, le piattaforme, le gemme, le porte e gli ostacoli, oltre alle entità nemiche e alle regole che ne determinano il comportamento e le interazioni. Il modello gestisce la fisica di base, come movimento, gravità e collisioni, ma anche il controllo dei livelli e della progressione di gioco.
Nel `Controller` viene coordinata l'esecuzione del gioco, gestendo il ciclo di aggiornamento e gli input dell'utente. Il `MainController` si occupa di sincronizzare la logica del gioco, mentre la classe `GameLoop` rappresenta il nucleo principale dell'esecuzione, mantenendo costante il flusso di aggiornamento tra modello e vista. Il controller svolge dunque un ruolo di intermediario tra il modello e la vista, aggiornando periodicamente lo stato logico del gioco e notificando alla vista i cambiamenti per la rappresentazione grafica.
La `View`, infine, è dedicata esclusivamente alla presentazione visiva degli elementi di gioco. Essa riceve i dati aggiornati dal modello tramite il controller e li rappresenta graficamente sullo schermo, senza modificare direttamente lo stato del gioco. In questo modo, la vista rimane completamente indipendente dalle logiche di calcolo e di input.
Questa organizzazione garantisce un'architettura ordinata e facilmente estendibile, favorendo la manutenzione e l'integrazione di nuove funzionalità.

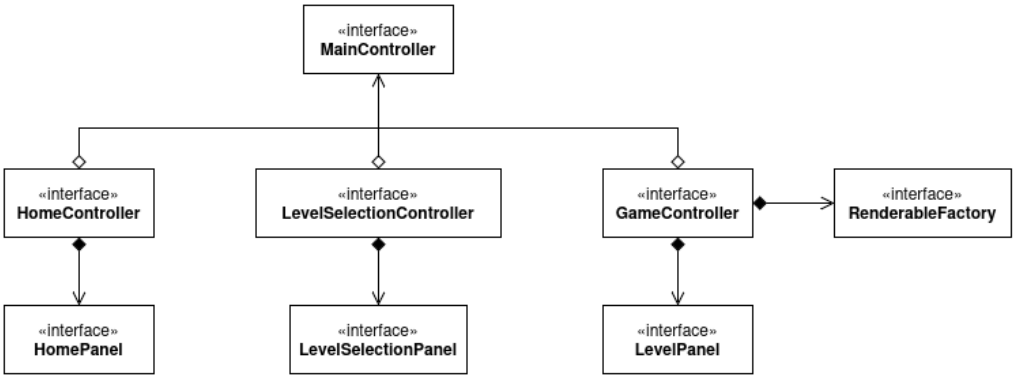

Figura 2.1: Architettura dei controller.

2.2 Design dettagliato

2.2.1 Distefano Stefano

Gestione dei Nemici

Problema Il gioco richiede di implementare diversi tipi di nemici. Il problema sta nel centralizzare le logiche di base che tutti condividono separandola però dai comportamenti unici dei diversi tipi di nemico.

Soluzione I nemici sono oggetti che, una volta creati in un punto della mappa, iniziano a muoversi orizzontalmente. Per gestire le differenze tra i vari tipi, la classe `Enemy` è stata astratta in `AbstractEnemy`, permettendo così di condividere la logica di movimento e di stato in comune. Seguendo il **Template Method Pattern**, il metodo `update(deltaTime)` contiene un metodo "gancio" astratto, `updateAttack(deltaTime)`, che si occupa di gestire l'eventuale logica di attacco specifica di ogni nemico. Per semplificare la creazione è stato usato il **Factory Method Pattern**, delegando alla `EnemiesFactory` l'istanziazione degli oggetti. Infine, per isolare la logica di movimento, questa è stata affidata a `EnemiesMoveManager`, che viene iniettata in ogni nemico.

Pro

- Nessuna duplicazione. La logica di fisica e movimento è centralizzata nella classe astratta.

- Semplice estensione di un nuovo nemico grazie all'astrazione senza cambiare tutta la classe `enemy`.
- L'architettura rispetta il **Single Responsibility Principle (SRP)**, separando le resposabilità in diverse classi.

Contro

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

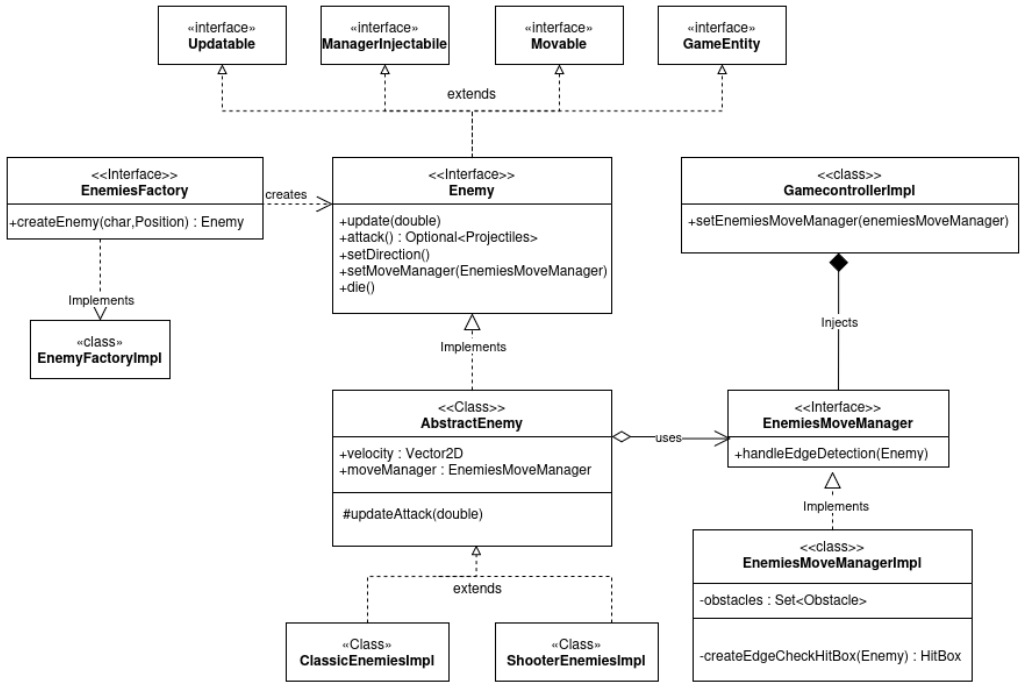

Figura 2.2: Astrazione di enemy e relazione con il moveManager e la Factory.

Gestione dei Proiettili

Problema Alcuni nemici sparano ciclicamente dei proiettili nella mappa. Il problema è gestire dinamicamente queste entità all'interno del livello, controllare creazione e collisioni di essi.

Soluzione I proiettili sono oggetti che implementano l'interfaccia `Projectiles`. La loro creazione è delegata alle singole entità nemiche, ma è orchestrata dal `GameControllerImpl`. All'interno del metodo `updateEnemies`, il controller itera su tutti i nemici attivi e invoca il metodo `enemy.attack()` su ciascuno. Grazie a questo design, il controller non ha bisogno di sapere il tipo specifico di nemico: se l'invocazione restituisce un proiettile, questo viene aggiunto alla lista di proiettili. La gestione delle collisioni e la loro disattivazione sono demandate strutture esterne: il `InteractionsManager` rileva l'impatto e, tramite l'`EventManager`, pubblica un evento. Questo disaccoppia la logica di collisione dal ciclo di vita del proiettile, che verrà poi rimosso in modo sicuro nel `GameController` in risposta a quell'evento.

Pro

- La logica di attacco permette di aggiungere nuovi tipi di nemici che sparano senza dover fare dei controlli sul tipo di nemico che sta attaccando.

Contro

- Le classi `enemy` sono direttamente legate alla classe concreta `ProjectilesImpl`. Questo rende più complesso aggiungere nuovi tipi di proiettili.

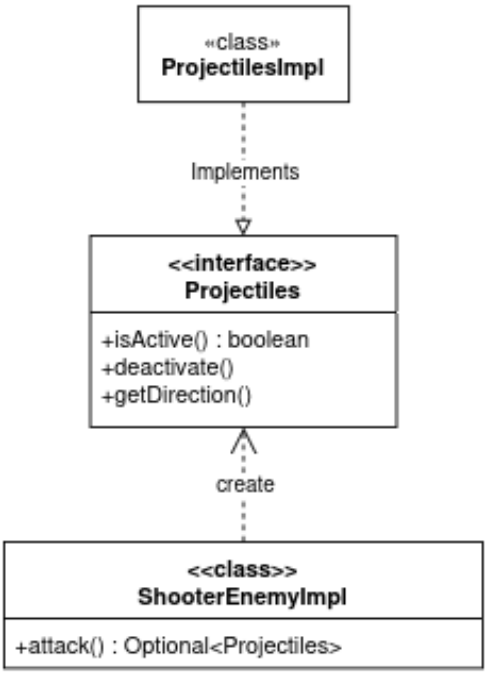

Figura 2.3: Creazione e implementazione dell'entità projectiles.

Gestione della Progressione e Salvataggio Dati

Problema Il gioco richiedeva un sistema di salvataggio per la progressione. Bisognava gestire la logica per cui l'utente potesse scegliere se caricare una partita vecchia o iniziare una partita completamente nuova.

Soluzione Per gestire la struttura del salvataggio è stato applicato il Facade Pattern. Ho creato `ProgressionManagerImpl` affinché agisca come gestore centrale e unico punto di accesso per il resto dell'applicazione. Esso coordina internamente i due componenti specializzati: `ProgressionState`, che agisce solo come "contenitore" dei dati (tempi, missioni), e `SaveManager`, che si occupa unicamente dell'aspetto tecnico di come scrivere e leggere i dati su file JSON.

Il `MainController` gestisce la configurazione iniziale: all'avvio, usa il `SaveManager` per caricare i dati e con quelli inizializza il `ProgressionManager`. Quando avvia un livello, "passa il testimone" iniettando questo `ProgressionManager` nel `GameController`. Il `GameController` si disinteressa del salvataggio fino a quando il livello non finisce. A quel punto, nel metodo `handleGameOver`,

chiama semplicemente `progressionManager.levelCompleted()`, delegando a `ProgressionManagerImpl` il compito di aggiornare i punteggi e salvare su disco.

Pro

- **Rispetta il principio SRP (Single Responsibility Principle):** L'architettura isola le varie responsabilità.

Contro

- **Leggero Aumento della Complessità:** A causa della protezione dei dati, la modifica dello stato non è più diretta.

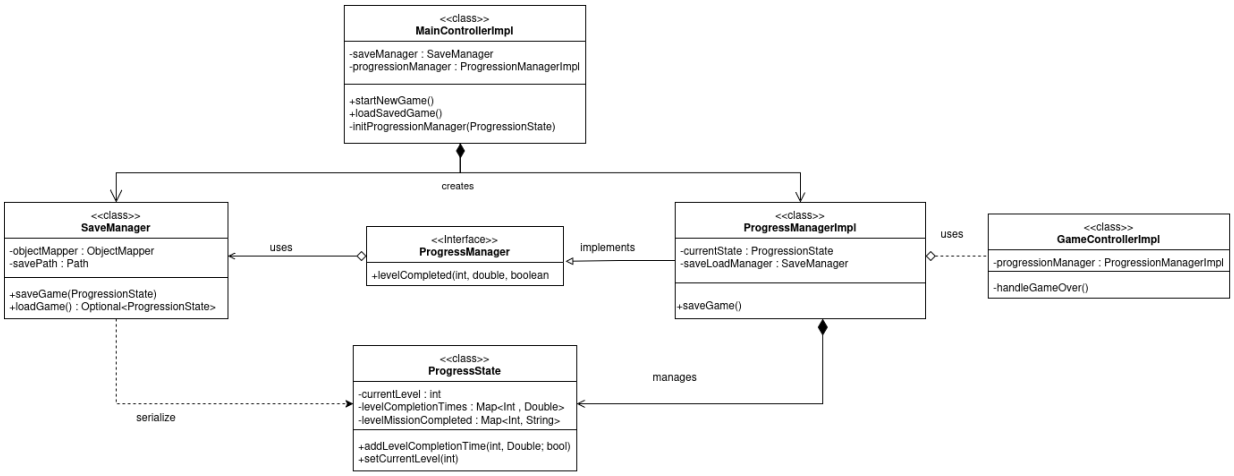

Figura 2.4: relazioni tra saveManager e progressionState con il progressionManager.

2.2.2 Ferri Alice

Gestione dei personaggi

Problema Nel gioco devono essere rappresentati due personaggi distinti. Ciascuno di diverso tipo e ricevono input differenti, ma accomunati da una stessa logica di base relativa a movimento, salto, gravità e collisioni. È quindi necessario sviluppare un sistema che consenta la condivisione di queste funzionalità mantenendo le caratteristiche di ciascun personaggio.

Soluzione La soluzione adottata per gestire i comportamenti dei personaggi, è stata definire l'interfaccia `Player`, che estende le interfacce `Movable` e `GameEntity`. Tale interfaccia include i metodi fondamentali per gestire lo stato del personaggio, la posizione e l'aggiornamento basato sugli input.
La classe astratta `AbstractPlayer` fornisce un'implementazione parziale dell'interfaccia `Player` centralizzando la logica condivisa tra i personaggi. Incapsula lo stato comune e fornisce implementazioni per il movimento, l'aggiornamento dello stato e interazioni con l'ambiente. Dal punto di vista architetturale, `AbstractPlayer` adotta il **Pattern Template Method**, definendo la struttura generale del comportamento del personaggio e demandando alle sottoclassi concrete la personalizzazione delle parti variabili. In questo modo, le classi derivate `Fireboy` e `Watergirl` possono specificare alcune logiche personalizzate, senza alterare il flusso principale definito dalla classe astratta.
Inoltre la classe `AbstractPlayer` utilizza handler diversi per isolare responsabilità specifiche. Tutti gli handler estendono l'interfaccia comune `PlayerHandler`, garantendo un contratto uniforme e fornendo ciascuno una corrispondente implementazione concreta
Questa struttura riflette l'applicazione del **Pattern Strategy**, permettendo di sostituire o estendere i comportamenti senza modificare la logica interna di `AbstractPlayer`. Inoltre, l'approccio adottato segue il principio di **Composition over Inheritance**, privilegiando la composizione di comportamenti tramite oggetti collaboratori, rendendo il sistema più flessibile e modulare.
L'architettura segue il **Pattern Factory**, attraverso `PlyaerFactory`, centralizzando la creazione dei personaggi e la configurazione dei loro handler.

Pro

- **Aderenza a SRP (Single Responsibility Principle):** L'architettura isola perfettamente le varie responsabilità
- **Centralizzazione del comportamento comune e aderenza a DRY (Don't Repeat YourSelf)** La logica e lo stato condivisi tra i personaggi sono implementati in un unica classe astratta, riducendo duplicazioni e il rischio di introdurre errori.
- **Facilmente estensibile** Nuove classi di personaggi o meccaniche di gioco possono essere aggiunte estendendo la classe comune.
- **Testabilità** Grazie alla separazione in handler, ogni componente può essere testato in isolamento.

Contro

- **Costruzione più complessa** Richiede più componenti da inizializzare e collegare.

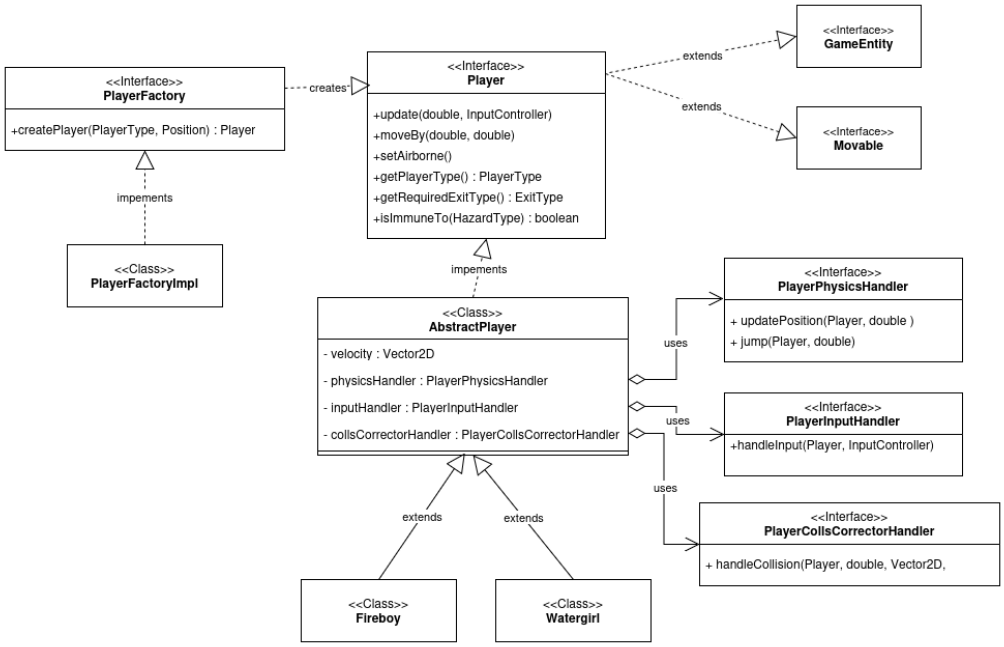

Figura 2.5: Gestione dei player

Gestione degli input da tastiera

Problema Ogni personaggio del gioco deve poter ricevere e gestire in modo indipendente i propri comandi da tastiera, permettendo movimenti simultanei e reazioni differenti in base al tipo di giocatore.
Inoltre, i personaggi devono reagire correttamente a input diversi senza interferenze tra di loro.

Soluzione La soluzione adottata prevede una gestione centralizzata degli eventi tramite la classe `InputControllerImpl`, che implementa sia l'interfaccia `InputController` sia `KeyEventDispatcher`. Tale approccio segue il **Pattern Observer**, in quanto l'oggetto controllore si pone come osservatore globale degli eventi da tastiera, intercettandoli e propagandoli sotto forma

di informazioni strutturate al resto del sistema. In questo modo la logica di input è completamente separata dal motore di gioco, favorendo modularità e indipendenza tra i componenti
Ogni personaggio è associato a una combinazione di tasti specifica, definita dalla classe interna immutabile `DirectionScheme`, che mappa i comandi di movimento e salto. Questa astrazione adotta il **Pattern Command**, poiché traduce le pressioni dei tasti in azioni semantiche indipendenti dall'effettiva sorgente dell'input
Al momento dell'aggiornamento del gioco, il metodo `getInputState()` restituisce uno snapshot immutabile `InputState` dello stato dei tasti per ciascun giocatore, indicando quali azioni siano attive. L'engine di gioco utilizza questo snapshot per aggiornare posizione, velocità e stati dei personaggi garantendo indipendenza tra i personaggi e controller. Tale struttura rappresenta un'istanza del **Immutable Object Pattern**, ispirato ai principi della programmazione funzionale, garantendo coerenza dei dati e assenza di effetti collaterali.

Pro

- **Aderenza a SRP** La cattura e gestione dell'input è isolata e indipendente dalla logica di movimento e fisica, rendendo il sistema più modulare e semplice da mantenere.
- **Centralizzazione e corenza** Tutti gli eventi da tastiera sono processati da un unico punto, evitando conflitti e garantendo che i comandi siano interpretati in modo uniforme.
- **Supporto per input simultanei** Ogni giocatore ha il proprio schema di tasti, consentendo movimenti indipendenti e simultanei.
- **Estensibilità, principio OCP (Open/ClosedPrinciple)** È possibile aggiungere nuovi personaggi, schemi di controllo semplicmente estendendo la mappa dei controlli.
- **Immutabilità e sicurezza dello stato** L'uso della classe `InputState` come snapshot immutabile riflette un approccio funzionale, garantendo coerenza e prevenendo effetti collaterali.

Contro

- **Maggiore complessità strutturale** La gestione tramite dispatcher e snapshot degli stati richiede più componenti e un'architettura più articolata.
- **Difficoltà di debug in tempo reale** Poiché gli eventi vengono processati centralmente e lo stato è immutabile, può risultare meno immediato osservare i cambiamenti durante l'esecuzione.

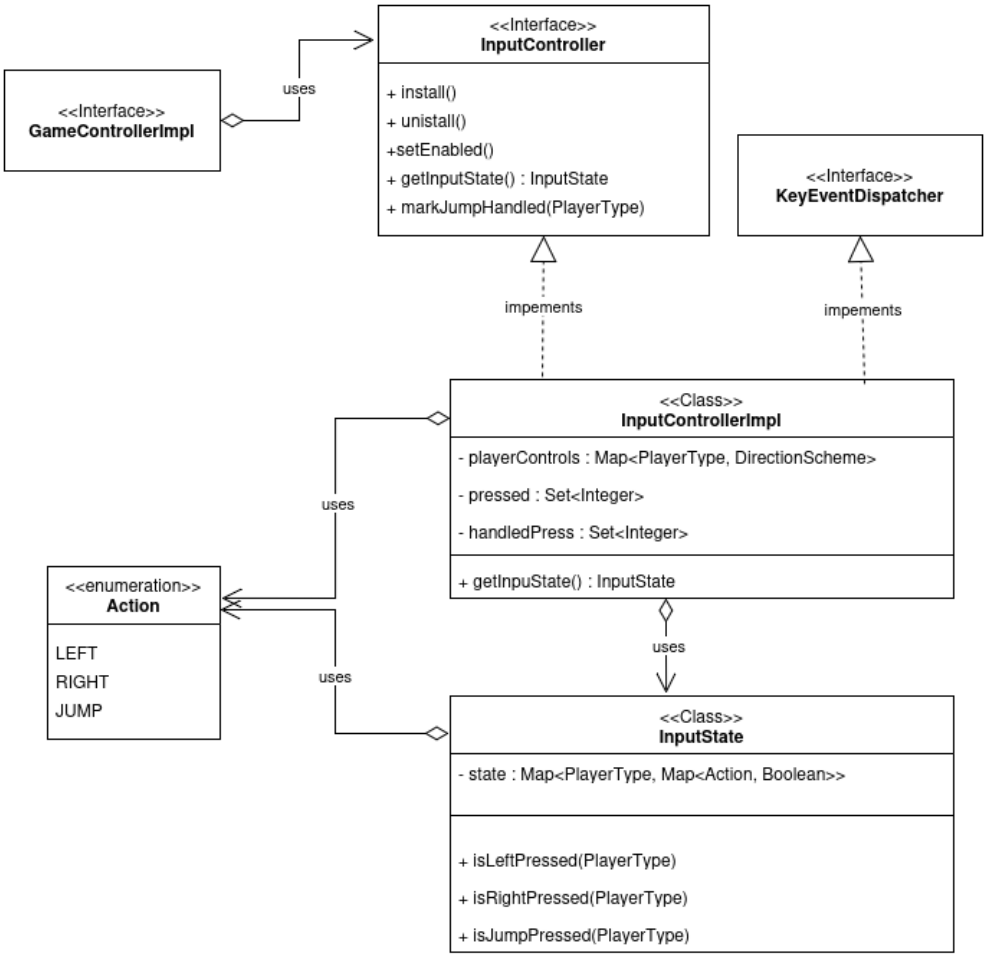

Figura 2.6: Gestione degli input da tastiera

2.2.3 Kadiu Christian

Problema Nel gioco sono presenti varie entità e tipologie di ostacoli diversi. L'interazione del `Player` con essi costituisce un elemento chiave del gameplay.

È necessario gestirne la creazione e il comportamento.

Soluzione Si è fatta una distinzione tra entità prettamente statiche, che compongono la mappa(come `Wall`, `Floor`, `Hazard`), e ostacoli interattivi, che presentano meccanismi dinamici e richiedono una logica di gestione più elaborata(come `Lever`, `Button` e `PushBox`).

Per la creazione delle entità è stato adottato il `Factory Method Pattern`: tramite le interfacce `ObstacleFactory` e `InteractiveObstacleFactory` la creazione viene centralizzata e disaccoppiata dal resto del codice.

Per la gestione delle logiche di attivazione (es. piattaforme mobili che reagiscono a leve o pulsanti), è stato utilizzato l'`Observer Pattern`. Le classi `Lever` e `Button` agiscono come `Subject` implementando l'interfaccia `TriggerSource`, che espone i metodi di registrazione dei listener, mentre `PlatformImpl` funge da osservatore e implementa `TriggerListener`. Il comportamento specifico di leve e pulsanti è definito tramite le interfacce `Toggler` e `Pressable`.

La classe `PushBox` rappresenta invece un elemento reattivo: implementa `Pushable` e `Movable`, consentendo al motore di collisione di gestire la sua spinta e correzione della posizione.

Pro

- Forte disaccoppiamento e separazione delle responsabilità. Il sistema risulta aperto all'estensione.
- Segregazione delle interfacce (ISP): l'uso di interfacce specifiche (`Toggler`, `Pressable`, `Pushable`) fa sì che ogni classe dipenda solo dai metodi necessari.

Contro

- aumento della complessità. L'uso combinato di pattern introduce un livello di astrazione che può complicare il sistema

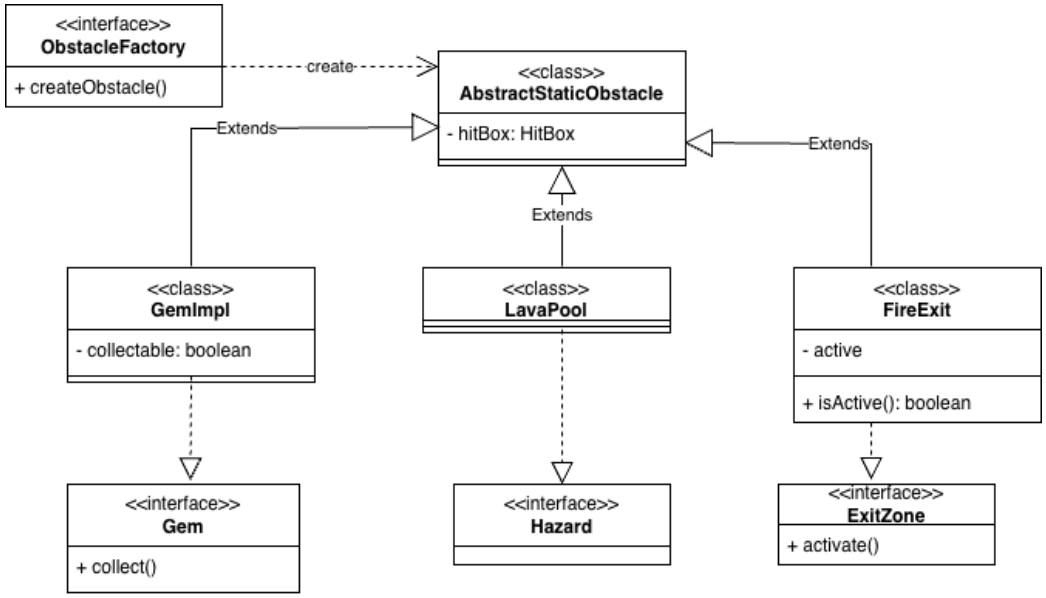

Figura 2.8: entità statiche

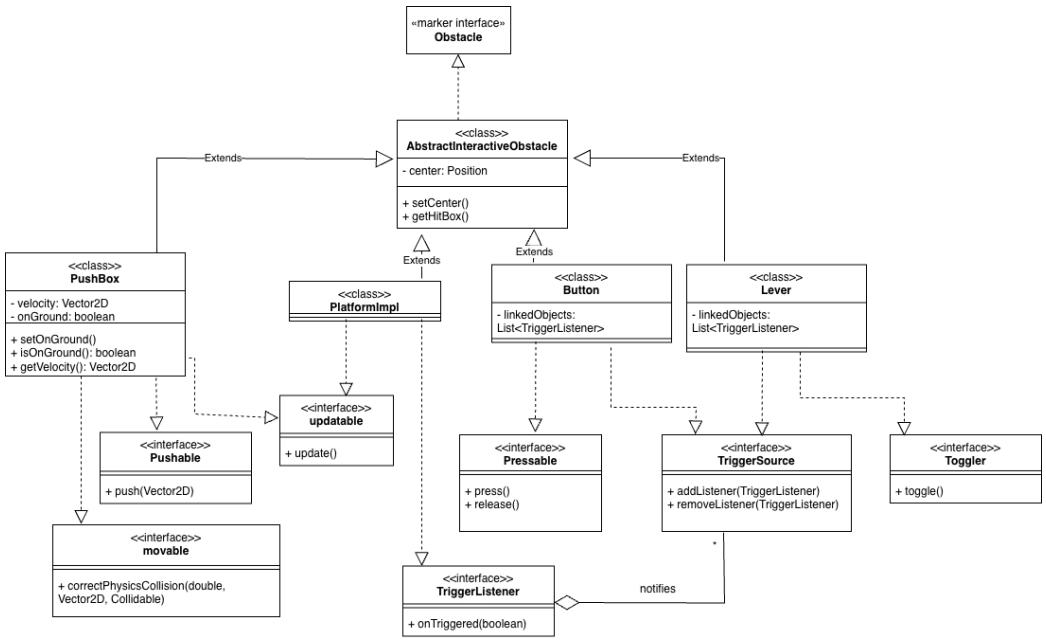

Figura 2.7: entità interattive

PROBLEMA La gestione delle interazioni fisiche e logiche tra le entità del gioco è il cuore del gameplay. La sfida progettuale era quella di realizzare un sistema di gestione efficiente, manutenibile e stabile.

SOLUZIONE È stata realizzata un'architettura a pipeline in tre fasi (rilevamento, gestione,risoluzione), orchestrata dalla classe InteractionsManager che funge da `Facade` dell'intero sistema. Ogni fase è delegata a componenti specializzati, garantendo la separazione delle responsabilità (SRP)

fase di rilevazione: L'`InteractionManager` delega il rilevamento delle collisioni al `CollisionChecker`, che nella sua implementazione fa uso di una struttura dati spaziale ottimizzata, un QuadTree, che effettua un partizionamento ricorsivo dello spazio di gioco. Questo permette, durante il controllo delle collisioni, una notevole riduzione della complessità computazionale, confrontando ogni entità solo con quelle spazialmente vicine. Ogni entità implementa l'interfaccia `Collidable`, il cui contratto rappresenta la fisicità dell'oggetto nel mondo, ed espone un `CollisionLayer` con una collision mask associata. Questa mask definisce quali altri layer innescano una reazione fisica con tale entità. In fase di gestione queste informazioni vengono utilizzate per determinare quali interazioni richiedono correzioni fisiche.

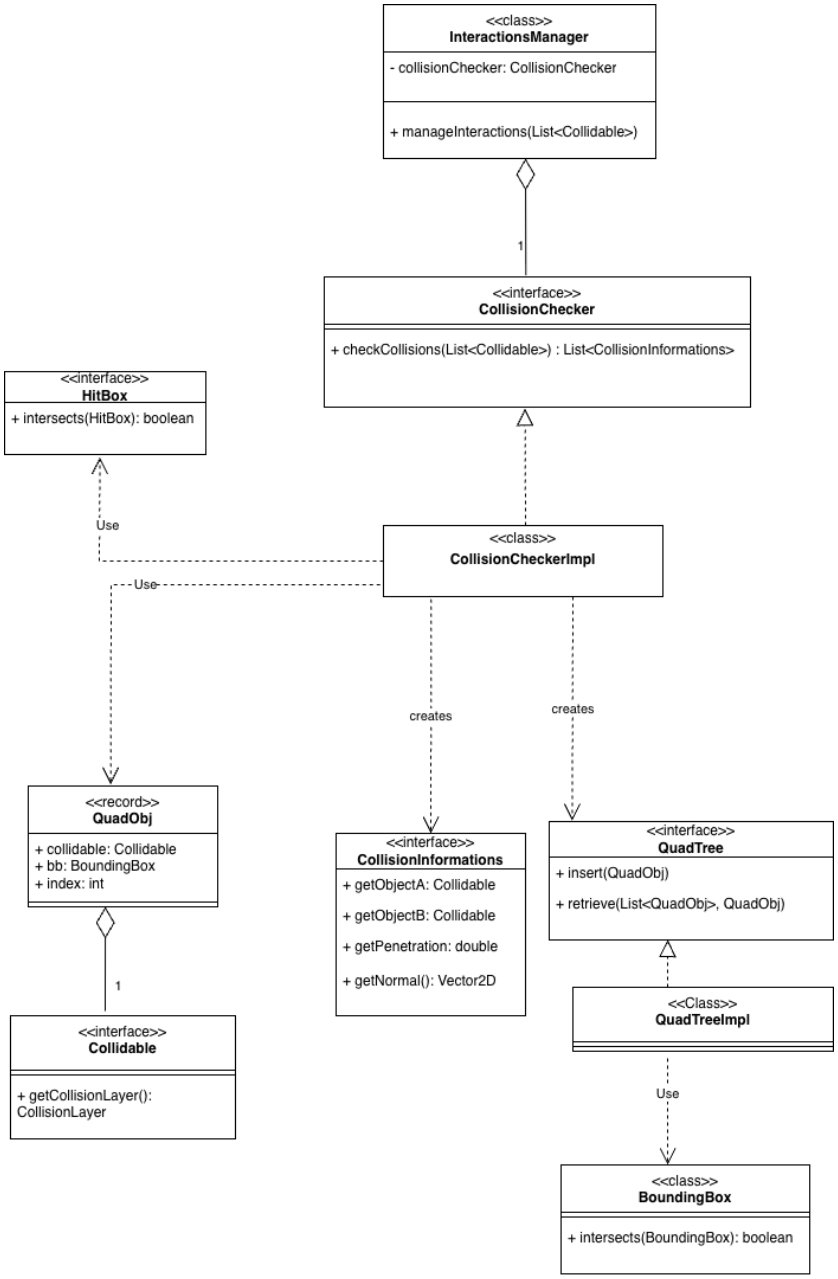

Figura 2.9: fase di rilevamento

fase di gestione: L'`InteractionsManager` non implementa alcuna logica di business, ma delega tutto alla classe `InteractionHandlersRegister` che applica lo **Strategy Pattern**. Questa contiene una lista di strategie, definite dall'interfaccia `InteractionHandler`. ogni strategia incapsula la gestione di una diversa interazione (player–enemy, gemme, hazard, ecc.). Quando arriva una coppia di `Collidable`, il registro usa il metodo `canHandle()` per selezionare gli handler compatibili e permettere loro di intervenire. Per ridurre codice boilerplate e centralizzare il type-checking, gli handler concreti estendono `AbstractInteractionHandler<A extends Collidable, B extends Collidable>`, che funge da **Template Method**: il suo metodo `handle()` effettua il controllo dei tipi e chiama il metodo astratto `handleInteraction()` lasciando alle sottoclassi solo la logica specifica. L'uso dei generics `<A, B>` garantisce che il casting sia sicuro a compile-time.

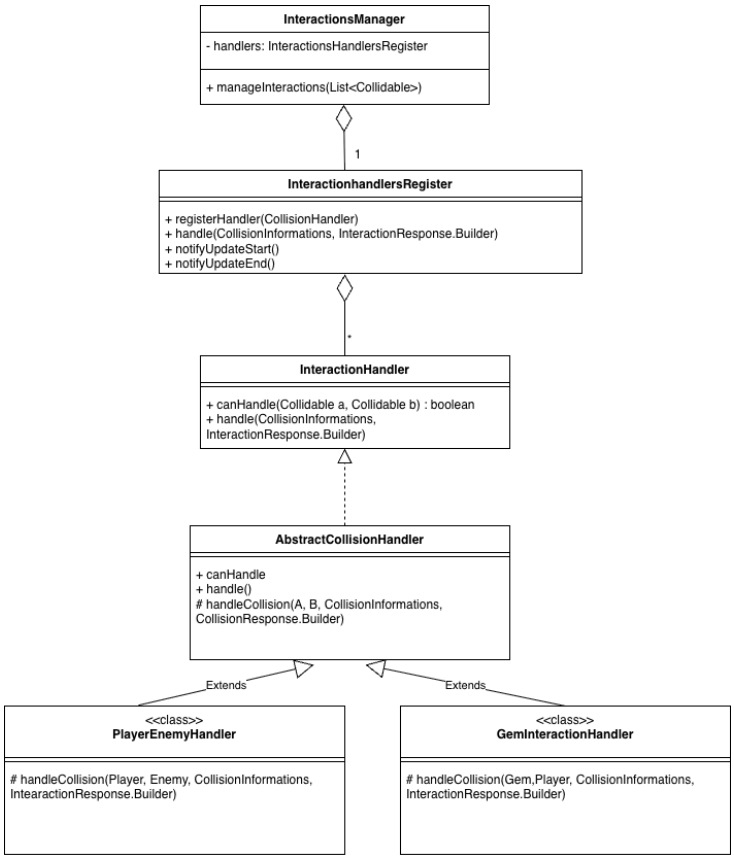

Figura 2.10: fase di gestione

fase di risoluzione: In questa fase è stato impiegato **Command Pattern**: Gli handler non eseguono immediatamente la loro risoluzione al mondo di

gioco, la incapsulano invece in un oggetto `InteractionCommand`, con unico metodo `execute()`. Grazie all'uso di **Builder Pattern**, viene costruito progressivamente un oggetto `CollisionResponse`, il cui compito è quello di raccogliere tutti i comandi generati dagli handler e mantenerli separati in: comandi fisici (come correzioni di posizione) e logici (modifiche allo stato dei componenti del gioco). A fine frame vengono eseguiti tutti i comandi raccolti, a partire da quelli fisici per evitare stati inconsistenti.

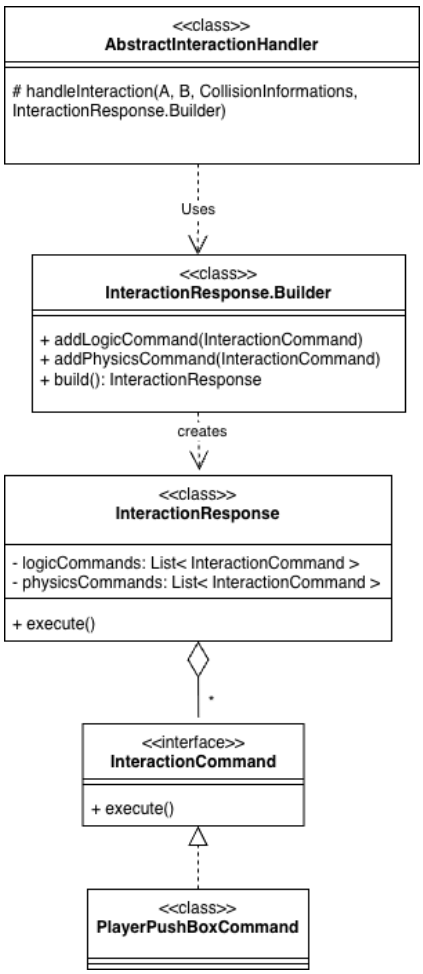

Figura 2.11: risoluzione delle interazioni

Pro

- L'architettura è aperta all'estensione (`OCP`). Grazie allo **Strategy Pattern**, per aggiungere una nuova interazione è sufficiente creare un nuovo `InteractionHandler` e registrarlo, senza modificare l'`InteractionsManager`, che rimane chiuso alle modifiche.
- L'uso del **Command Pattern** disaccoppia la decisione dall'esecuzione. L'esecuzione differita a fine frame e la separazione tra comandi fisici e logici rendono il sistema stabile.
- L'architettura rispetta il `SRP` e `DRY`, ogni classe ha un ruolo definito ed evita ripetizione di codice

Contro

- Complessità architetturale: l'alto livello di astrazione dovuto all'uso combinato di pattern rende il debug più complesso e aumenta la complessità del sistema.

Problema Sistema di comunicazione tra le componenti del sistema. Le componenti devono poter comunicare tra di loro per aggiornare il loro stato o il loro comportamento a seguito di un avvenimento nel gioco.

Soluzione Ho utilizzato l'**Observer Pattern** per implementare un sistema ad eventi centralizzato, pensato per notificare il resto del gioco a seguito di una risoluzione logica da parte del motore centrale affrontato sopra. L'architettura si basa su tre componenti chiave:

- `Event:` Interfaccia marker implementata da classi immutabili che rappresentano messaggi specifici (es. `PlayerDiedEvent`, `GemCollectedEvent`).
- `EventListener :` Interfaccia implementata dagli observer, che si iscrivono per ascoltare un determinato evento (es. `GameStateImpl`). Espone il metodo `onEvent()` che definisce il comportamento del listener quando l'evento si verifica.
- `EventManager` Classe che fa da hub centrale. Permette la registrazione/rimozione dei listener e li notifica quando si verifica un evento

Pro

- **Totale disaccoppiamento:** I publisher notificano eventi senza sapere chi o quanti li ascoltano. I subscriber reagiscono senza conoscere l'origine dell'evento. Entrambi dipendono solo dall'astrazione data da `Event` e `EventManager`.
- **Aderenza a SRP e OCP:** Le classi hanno un ruolo ben definito e il sistema è aperto all'estensione e chiuso alla modifica. Si possono aggiungere nuovi eventi o listener senza modificare codice esistente.

Contro

- **Difficoltà di debug:** L'alto livello di astrazione rende più complesso tracciare il flusso logico degli eventi.
- **Dipendenza centrale:** L'`EventManager` diventa un componente centralizzato per il corretto funzionamento del sistema.

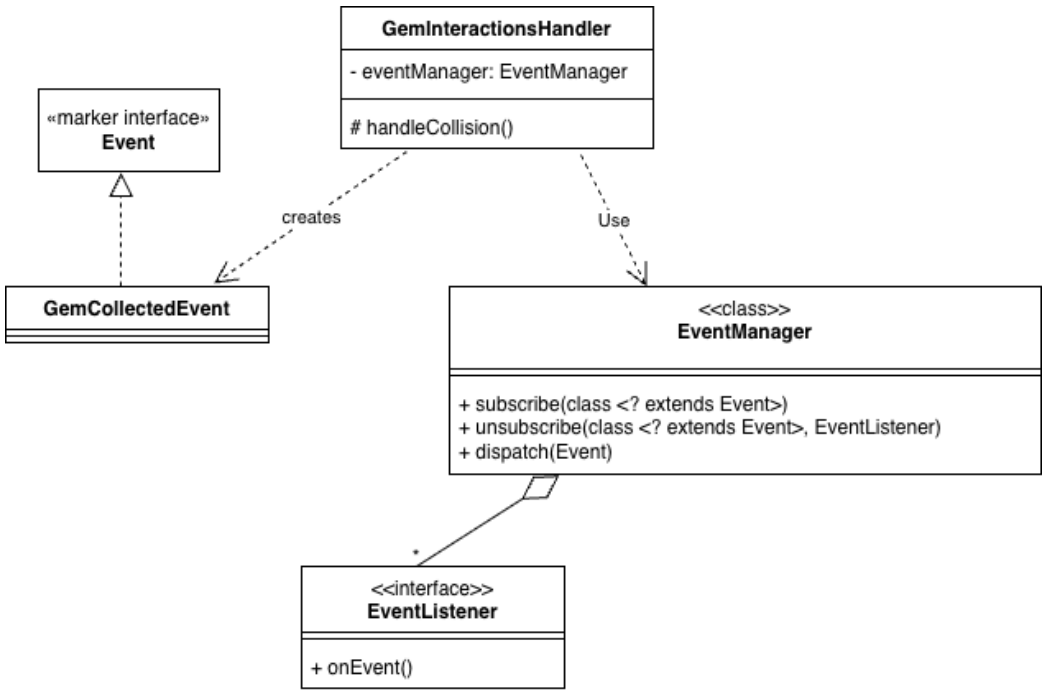

Figura 2.12: gestione degli eventi

2.2.4 Meloni Leonardo

Problema Gestire la creazione di una missione comune a tutti i livelli, formata da diversi obiettivi da raggiungere.

Soluzione La soluzione consiste nella creazione di un'entità astratta chiamata `AbstractObjective` che definisce i campi e metodi comuni utilizzati dai vari obiettivi, con l'implementazione di un **Template Method**, `checkCompletion()`, che imposta la base di completamento di un obiettivo, verificato poi tramite il metodo abstract `checkObjective` che cambierà da obiettivo a obiettivo. Per la gestione della missione ho applicato il **Composite Pattern** che permette di gestire tutti gli obiettivi come se fossero un unica istanza della classe `Mission`. Estende anch'essa `AbstractObjective` e il suo compito, quindi, è verificare che tutti gli obiettivi siano completati nascondendo la gerarchia inferiore. In caso di esito positivo anch'essa risulterà completata.

Pro

- La soluzione è aperta all'estensione `(OCP)`, in quanto la creazione e l'implementazione nella missione di un nuovo obiettivo è immediata.
- Il pattern Composite nasconde la complessità della gerarchia di obiettivi al gestore (`MissionManager`), che tratta l'intera missione come un singolo `Objective`, garantendo il disaccoppiamento.
- Ogni classe ha la propria responsabilità `(SRP)` e l'uso del **Template Method** in `AbstractObjective` evita la ripetizione di codice `(DRY)` per la logica di completamento.

Osservazioni

- In aggiornamenti futuri, si potrebbe implementare l'Observer Pattern per gestire obiettivi real-time.

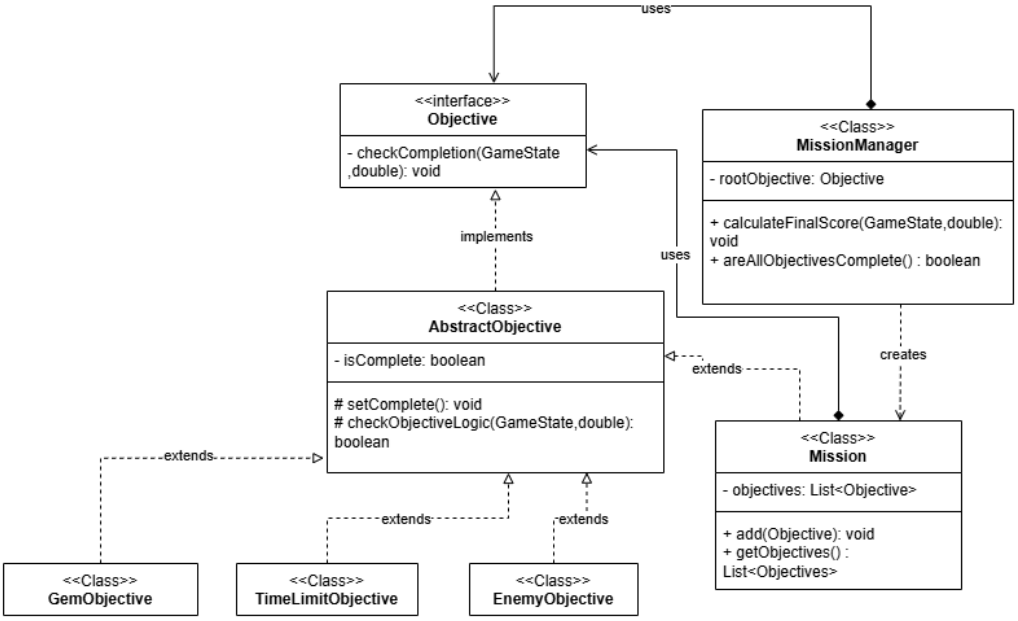

Figura 2.13: Gestione della missione

Problema Durante l'esecuzione di un livello, lo stato logico del gioco (numero di nemici sconfitti, gemme raccolte, ecc.) è in continuo aggiornamento. Nasce quindi la necessità di tracciare queste informazioni e, soprattutto, di determinare lo stato finale (vittoria o sconfitta).

Soluzione La soluzione consiste nella realizzazione della classe `GameState` che agisce come un "registratore" di stato. La comunicazione avviene tramite il **Pattern Observer**

La classe `GameState` implementa l'interfaccia `EventListener` (l'*Observer*) e, nel suo costruttore, si sottoscrive agli eventi di suo interesse tramite un `EventManager`. Quando un evento (es. `PlayerDiedEvent`) si verifica, `GameStateImpl` lo riceve e aggiorna il proprio stato interno.

Per garantire la robustezza, la classe utilizza controlli di sicurezza per impedire aggiornamenti di stato dopo che `gameOver` è diventato `true`, risolvendo così le race condition.

Pro

- La soluzione aderisce perfettamente al principio `SRP`. La classe `GameState` ha una sola responsabilità: "tracciare lo stato". Non sa nulla di colli-

sioni, input o grafica. La sua unica ragione per cambiare è se cambiano le regole di vittoria/sconfitta.

- La logica interna della classe è chiara, robusta e facile da comprendere (**aderenza a KISS**) .

Contro

- Non ci sono svantaggi significativi, in quanto l'Observer Pattern è la soluzione ideale per questo problema, garantendo reattività (lo stato si aggiorna immediatamente all'evento) senza introdurre accoppiamento stretto.

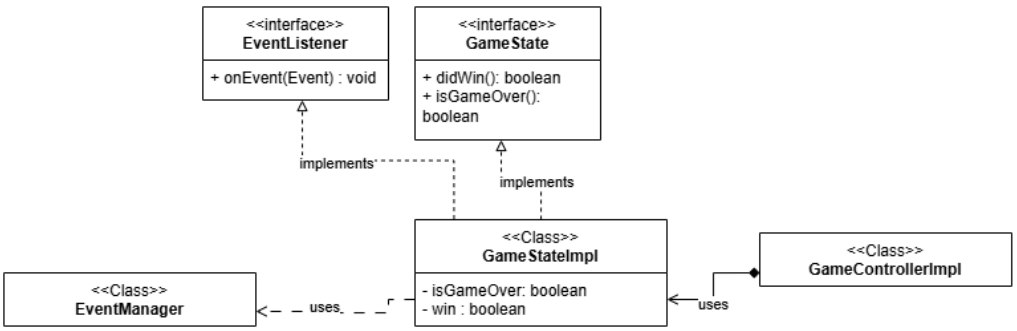

Figura 2.14: Gestione del GameState

Problema Gestione delle varie entità di gioco, caricamento della mappa da file e validazione di quest'ultima.

Soluzione La soluzione scompone il problema in componenti diversi, ognuno con una singola responsabilità, che vengono gestiti dal `GameController`:

`MapLoader`: Si occupa solo dell'I/O. Legge il file e, per ogni char, delega la creazione alla factory e restituisce un `Set` .

`LevelImpl`: Il `Set` viene passato al costruttore di `Level`. Questa classe agisce come un contenitore sicuro, ovvero da Wrapper:

Esegue una copia difensiva del `Set` nel costruttore per impedire modifiche esterne tramite il riferimento originale. Tutti i suoi metodi restituiscono viste non modificabili per proteggere la sua collezione interna. Sono state inoltre relizzate due interfacce LevelData e LevelUpdate per evitare la modifica al set a classi non adeguate.

`MapValidatorImpl`: Infine, l'oggetto `Level` (ora incapsulato e sicuro) viene passato a un validatore separato. Il `MapValidator` esegue la logica

di business (se esiste un solo FireBoy, se esite un percorso idoneo di uscita etc...), lanciando eccezioni in caso di fallimento.

Pro

- Il design isola perfettamente le responsabilità `SRP`: `MapLoader` (I/O), `Level` (Gestione della collezione) e `MapValidator` (Logica e regole). Ogni classe ha una sola ragione per cambiare.
- `Level` , eseguendo una copia difensiva e restituendo viste non modificabili, impedisce modifiche accidentali alla lista delle entità durante il game loop.
- Le classi sono disaccoppiate e testabili unitariamente (es. si può testare `MapValidator` con un `Level` creato ad hoc).

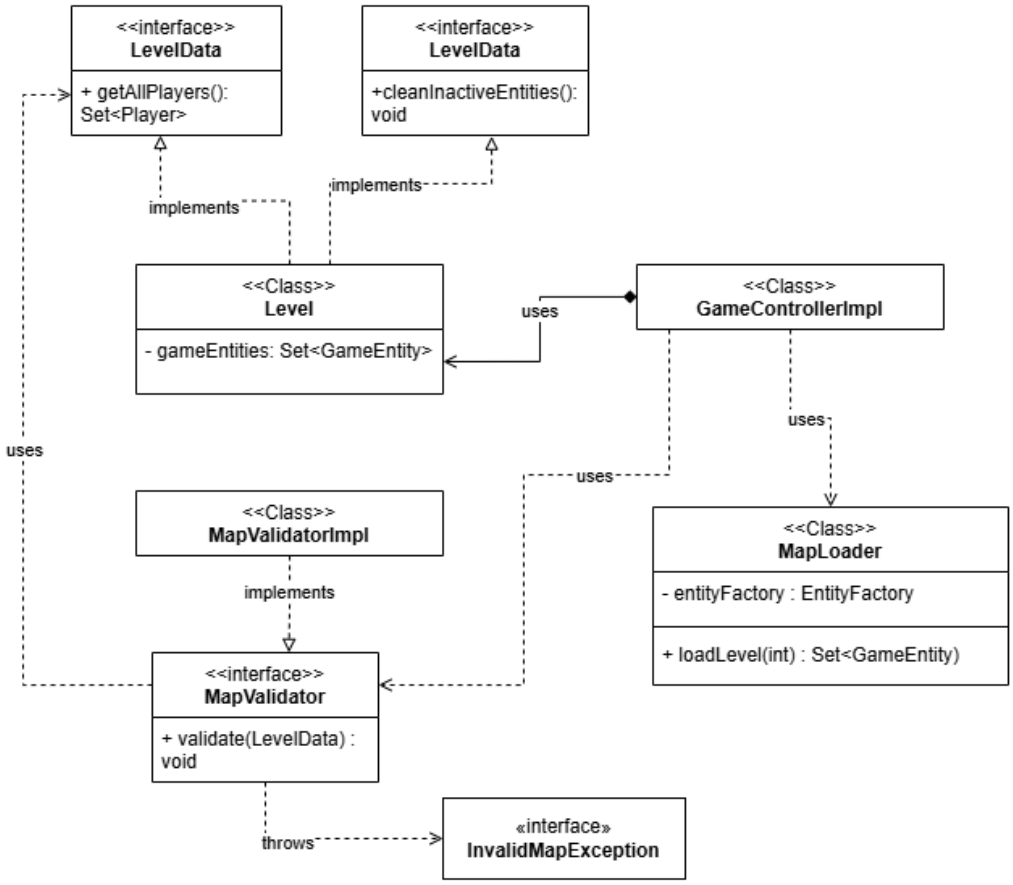

Figura 2.15: Gestione del Level,MapValidator e MapLoader

Problema Il nostro gioco è formato da più stati (Menù Iniziale, Selezione Livelli, Gioco). Sorge dunque il problema di gestire la navigazione e la collaborazione tra questi stati.

Soluzione La soluzione adottata è l'implementazione di un Controller Principale (`MainControllerImpl`), che agisce da `Coordinator`. Questo controller è l'unico componente a conoscere l'intera struttura di navigazione ed è il "cardine" dell'applicazione.

Quando si naviga,il Main Controller segue il ciclo di vita dell'applicazione attivando o disattivando i vari controller secondari.

Per evitare che i sotto-controller dipendano dall'intera classe del MainContoller, la soluzione applica il Principio di Segregazione delle Interfacce (ISP). Vengono create interfacce di navigazione specifiche come HomeNavigation, GameNavigation . Il Main Controller le implementa tutte, ma quando si passa come parametro ad un sotto-controller (es. `HomeController`), passa sé stesso "filtrato" solo con l'interfaccia di cui quel controller ha bisogno (es. HomeNavigation).

Pro

- I sotto-controller dipendono solo da astrazioni (HomeNavigation, GameNavigation), non dall'implementazione concreta del Main Controller. L'uso di interfacce dedicate limita l'accesso ai soli metodi necessari, nascondendo il resto dell'applicazione.
- Le responsabilità sono separate (SRP). `Home Controller` gestisce la logica del menu; il Game Controller gestisce la logica del gioco; `Il Main Controller` ha la sola responsabilità di gestire la navigazione.
- Il sistema è facilmente estendibile (OCP). Per aggiungere una nuova schermata (es. una `GuidePanel`), è bastato creare il pannello, aggiungere un metodo all'interfaccia `HomeNavigation` e implementarlo in `Main Controller`. Non è stato necessario modificare gli altri controller.

Contro

- La necessità di avere un centro di navigazione in MainController che rischia in applicazioni di grandi dimensioni di aumentare la complessità del sistema. Ma nel nostro caso di studio questo problema non persiste.

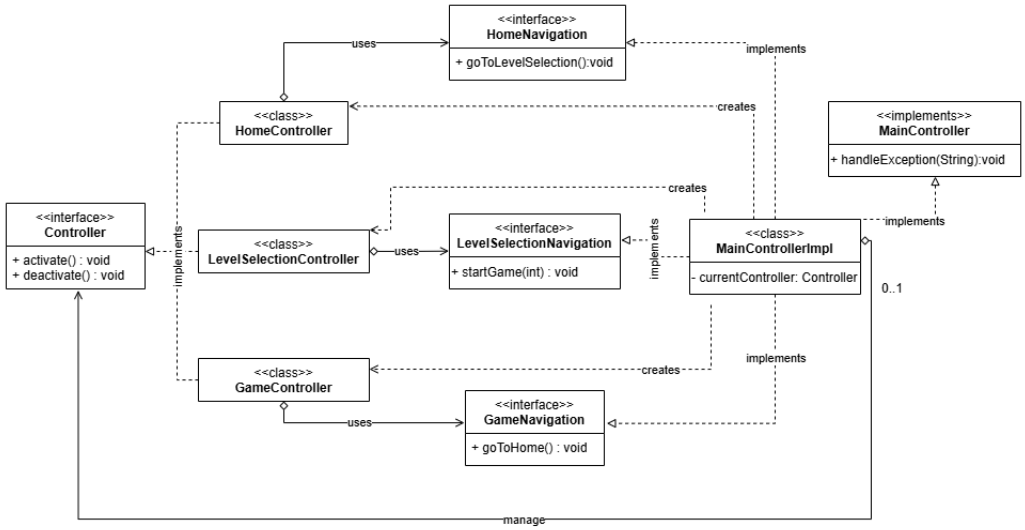

Figura 2.16: Gestione dei controller

Problema Il gioco è composto da varie entità eterogenee (es. `Player`, `Enemy`...). Occore quindi gestire la loro gestione da parte della classe `Level` nel modo adeguato e la loro creazione, che avviene con parametri differenti.

Soluzione La soluzione è stata divisa in due parti:

Si realizza un'interfaccia comune `GameEntity`, che unifica tutte le entità. Questo permette alla classe `Level` di gestirle in un unico `Set<GameEntity>`, e di fornire metodi di filtraggio (es. `getAllPlayers()`) per l'accesso specializzato. Per la creazione invece si realizza un `EntityAssembler`, utilizzando il `Facade Patter` che astrae la creazione dal `MapLoader`. Al suo interno utilizza una serie di sub-factories e tramite una strategia che associa ogni `char` a una specifica di creazione (es. `interactiveObsFactory::createLever`), realizza l'opportuna entità.

Pro

- La creazione resta indipendente dal `MapLoader`, aderendo al principio SRP.
- L' `EntityAssembler` con la sua strategia interna è aperta all'estensione. Per aggiungere una nuova entità, è sufficiente aggiungere una riga alla sua mappa di creazione, senza modificare la logica esistente soddisfando quindi il princio di OCP.

- La classe `Level` gestisce le entità in un'unica collezione, evitando la duplicazione della logica di gestione (filtri, pulizia) per ogni tipo di entità, aderendo al principio DRY.

Contro

- La classe `EntityAssembler` dipende strettamente dalle sotto-factory, necessario per mantenere il resto dell'applicazione (come il `MapLoader`) semplice e disaccoppiato.

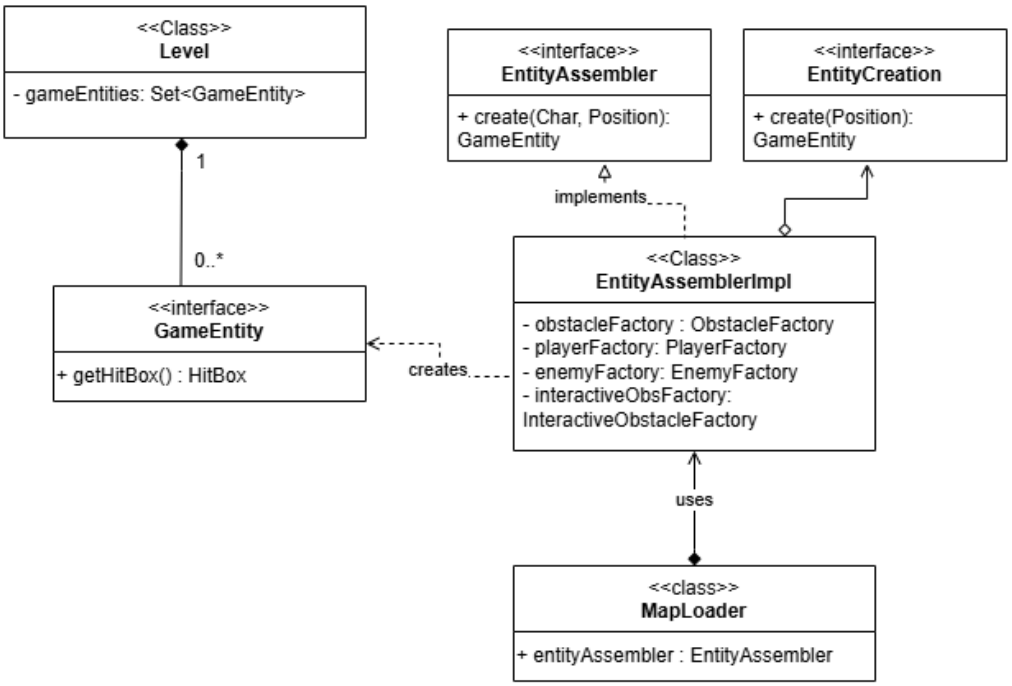

Figura 2.17: Gestione delle entità

Capitolo 3

Sviluppo

3.1 Test Automatizzati

Per il progetto è stato realizzato un insieme di test automatici utilizzando la suite *JUnit*, finalizzati principalmente alla verifica del modello logico dei livelli di gioco. L'obiettivo dei test è stato principalemente quello di assicurare la correttezza del comportamento del gioco.

- `TestFireboy`: testa il corretto funzionamento del player Fireboy e i relativi movimenti e aggiornamenti.
- `TestWatergirl`: testa il corretto funzionamento del player Watergirl e i relativi movimenti e aggiornamenti.
- `TestClassicEnemy` - Testa il Classic enemy, verificando il movimento, la collisione e che il suo attacco sia sempre vuoto.
- `TestProjectilesImpl` - Testa il proiettile, verificando il suo stato iniziale, il movimento e la disattivazione.
- `TestShooterEnemy` - Testa il nemico che spara, concentrandosi sulla logica di attacco e la gestione del cooldown.
- `TestProgressionState` - Testa lo stato di progressione, assicurando che salvi correttamente solo i tempi migliori e lo stato di completamento delle missioni.
- `TestEntityAssembler` - Testa la creazione delle entità con caratteri validi o la risposta in caso di caratteri non validi

- `TestLevel` - Testa i vari filtri utilizzati per richiamare le istanze di entità e il corretto aggiornamento del set in caso di entità inattive.
- `TestMapLoader` - Testa il comportamento in caso di file assente e che vengano create correttamente le entità.
- `TestMapValidator` - Testa il funzionamento dei metodi di controllo delle regole di gioco scelte tramite l'utilizzo di mappe non valide.
- `TestMission` - Testa il comportamento della missione in caso di obiettivi mancati e nel caso di successo
- `TestCollisionCheckerImpl` – Valida l'intero ciclo di rilevamento collisioni: crea collider sovrapposti o separati e verifica che il `CollisionChecker` produca (o meno) le `CollisionInformations` con penetrazione e normale corretti.
- `TestPhysicsHandler` – Controlla la gestione della collisione fisica.
- `TestButtonActivationHandler` – Simula l'intero ciclo di vita di un pulsante, assicurandosi che `press()` venga chiamato una sola volta per contatto e che `release()` venga invocato quando il player si allontana.
- `TestPlatformImpl`: controlla lo stato della piattaforma, e la sua attivazione.
- `TestPushBox`: controlla il comportamento fisico della PushBox

3.2 Note di Sviluppo

3.2.1 Distefano Stefano

- **Utilizzo di Lambda expression e Stream.** Un esempio in questo permalink.
- **Utilizzo di Optional.** Un esempio in questo permalink.

3.2.2 Ferri Alice

- **Utilizzo di Lambda expression.** Un esempio in questo permalink.
- **Utilizzo di Stream.** Un esempio in questo permalink.
- **Utilizzo di Optional.** Un esempio in questo permalink.

3.2.3 Kadiu Christian

- **Utilizzo di Stream.** Usati in svariati punti permalink.
- **Utilizzo di generics.** Usati in svariati punti permalink.
- **Utilizzo di Lambda Expression.** usate in svariati punti. permalink.
- **Utilizzo di Optional.** permalink.

3.2.4 Meloni Leonardo

- **Utilizzo di Reflection.** Un esempio in questo permalink.
- **Utilizzo di Stream.** Un esempio in questo permalink.
- **Utilizzo di Lambda Expression.** Un esempio in questo permalink.

Capitolo 4

Commenti Finali

4.1 Autovalutazione e lavori futuri

4.1.1 Distefano Stefano

Lavorare a questo progetto è stata un'esperienza davvero costruttiva. Personalmente, ho trovato molto stimolante il lavorare in gruppo, mi ha permesso di mettermi alla prova ed approfondire le mie conoscenze in Java. Nonostante qualche inevitabile piccolo problema di coordinamento iniziale, siamo riusciti a risolverli velocemente, definendo meglio i ruoli e confrontandoci con regolarità. Sviluppare questo progetto ha rafforzato la mia capacità di coordinarmi efficacemente con i colleghi e mi ha dato l'opportunità di padroneggiare l'uso di GitHub per il controllo di versione in un contesto di team.

4.1.2 Ferri Alice

La partecipazione a questo progetto ha rappresentato per me un'esperienza particolarmente significativa, trattandosi del primo lavoro di gruppo di questa entità a cui ho preso parte. Le fasi iniziali non sono state semplici, poiché è stato necessario comprendere a fondo la struttura generale del progetto attraverso una buona analisi e adattarsi a una metodologia di sviluppo collaborativa. Inoltre, la distanza fisica ha reso per me la comunicazione inizialmente più complessa, ma grazie all'impegno siamo riusciti a collaborare efficacemente anche a distanza, attraverso chiamate e incontri online.
Mi sono occupata principalmente della gestione del player e dell'input. Questo ruolo mi ha permesso di approfondire le mie conoscenze del linguaggio Java e di applicare in modo concreto concetti appresi durante le lezioni, come la gestione degli eventi, il controllo del flusso logico e l'organizzazione modulare del codice. Durante lo sviluppo ho avuto modo anche di approfondire

l'utilizzo di Git, imparando a gestire versioni del progetto, risolvere conflitti e collaborare in modo efficace sullo stesso repository, competenza che si è rivelata essenziale per il lavoro di gruppo.
Il lavoro di squadra ha avuto un ruolo determinante nel superamento delle difficoltà incontrate. Attraverso la comunicazione e il supporto reciproco, siamo riusciti a risolvere i problemi in modo efficace.
Nel complesso, considero questa esperienza estremamente formativa, mi ha consentito di sviluppare non solo competenze tecniche, ma anche capacità trasversali fondamentali come l'adattabilità, la comunicazione in team e il problem solving. In prospettiva futura, mi piacerebbe contribuire a un eventuale perfezionamento del progetto, ampliandone le funzionalità e consolidando ulteriormente quanto appreso.

4.1.3 Kadiu Christian

La realizzazione di questo progetto è stata un'ottima occasione per applicare le conoscenze teoriche che ho acquisito fin'ora in un progetto concreto. In particolare, mi sono occupato, tra le altre cose, della gestione centrale delle interazioni logico–fisiche tra le entità del gioco, un compito che ha richiesto un forte coordinamento con il resto del team. Per far procedere il lavoro in modo coerente in una direzione comune, ho dovuto comprendere le esigenze di ciascun membro del gruppo e trovare un punto di incontro tra idee e soluzioni tecniche. Questo non è stato affatto facile, soprattutto nelle fasi iniziali, per via di varie problematiche organizzattive.Lavorare a questo progettomi ha permesso di rafforzare le mie competenze nell'uso di git e di conoscere meglio questo strumento. Nel complesso è stato molto stimolante, ho avuto modo di mettermi alla prova e questo mi ha motivato ad intraprendere nuovi progetti di sviluppo e sperimentare nuove idee.

4.1.4 Meloni Leonardo

Il lavoro di gruppo è ormai fondamentale in un ambiente come quello informatico. Penso, quindi, che questo progetto mi abbia aiutato a sviluppare una mentalità meno individualistica ma più orientata al lavoro di squadra. Non è stato facile, ci sono stati alcuni problemi iniziali di organizzazione ma che siamo comunque riusciti a gestire. Soprattutto la mia parte (Livello,GameState etc. . .), essendo molto a stretto contatto con quella degli altri membri, ha reso necessario il confrontarmi spesso con loro e mi ha permesso di capire come realizzare una comunicazione chiara ed efficace. Inoltre, penso che lavorare a contatto diretto in un progetto del genere permetta di capire al meglio i concetti appresi durante le lezioni, come nell'utilizzo dei pattern

o degli streams. Mi ha permesso anche di migliorare nell'uso di Git. Questa esperienza mi ha dato lo stimolo e delle nuove idee per lanciarmi in progetti futuri da portare avanti, tra cui anche perfezionare il gioco, mi piacerebbe implementare un editor di livelli interno.

Appendice A

Guida utente

A.1 Descrizione Software

Il software sviluppato è un videogioco a due giocatori in cui l'obiettivo principale è quello di raggiungere le rispettive porte di arrivo, evitando ostacoli mortali e nemici presenti nel percorso. Il gioco unisce elementi di piattaforma e cooperazione, richiedendo coordinamento tra i due giocatori per superare le difficoltà e completare i livelli.

A.2 Menu Principale

All'avvio dell'applicazione viene visualizzato il menu principale, composto da diverse opzioni interattive:

- **Inizia a Giocare**: avvia una nuova partita mostrando la schermata di selezione del livello.
- **Carica Salvataggio**: consente di riprendere una partita precedentemente salvata.
- **Guida del gioco**: mostra le istruzioni di gioco e i comandi principali.
- **Esci Dal Gioco**: chiude l'applicazione in modo sicuro.

Il design del menu e dei pulsanti è coerente con lo stile grafico del gioco ed è stato pensato per essere chiaro e minimale.

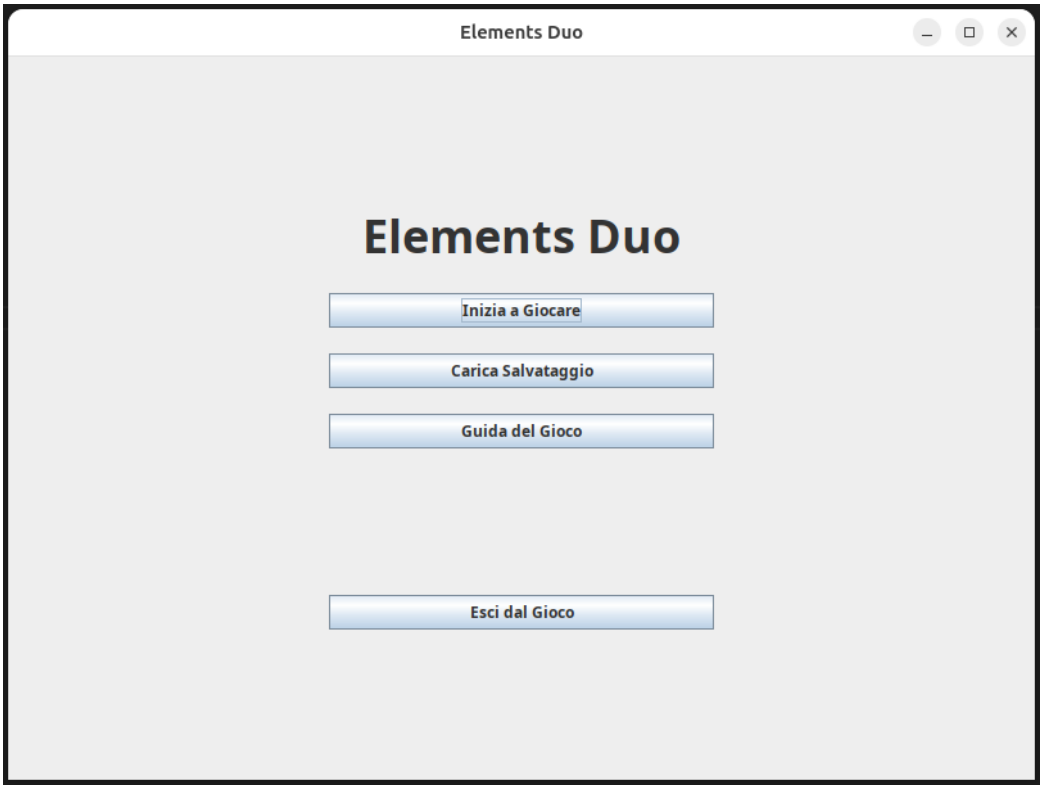

Figura A.1: Schermata Menu Principale

A.3 Modalità e comandi di gioco

Il gioco prevede la presenza di due giocatori, ognuno dei quali controlla un personaggio con abilità equivalenti ma ruoli e modalità di interazione con l'ambiente diverse. Entrambi, per completare il livello, devono raggiungere la propria **porta di destinazione**, che appare dello stesso colore del player. Durante la partita, i giocatori devono evitare le **zone verdi** del percorso, i proiettili dei nemici e i **nemici** stessi o possono eliminarli saltando su di essi. Mentre le **zone azzurre** e **arancioni** risultano letali esclusivamente per il personaggio del colore opposto. Il contatto tra il giocatore e una di queste entità determina il riavvio del livello.
Nel corso del livello saranno presenti **blocchi** da spostare, utili per creare passaggi o raggiungere aree sopraelevate, e **piattaforme mobili**, che potranno essere attivate quando uno dei personaggi si posiziona su un apposito blocco sensibile. In questo modo la piattaforma si metterà in movimento, permettendo all'altro giocatore di salire e proseguire lungo il percorso. Queste meccaniche richiederanno coordinazione e cooperazione tra i due personag-

gi. Inoltre, saranno presenti **gemme** da collezionare, che contribuiranno al superamento della sfida del livello.

I **comandi** sono semplici e intuitivi, pensati per consentire il controllo simultaneo dei due personaggi su una singola tastiera:

- **Watergirl**: utilizza i tasti **freccia sinistra**, **freccia destra** per il movimento orizzontale e **freccia su** per il salto.
- **Fireboy**: utilizza i tasti **A** per muoversi a sinistra, **D** per spostarsi verso destra e **W** per saltare.

Attraverso i due pulsanti in alto sullo schermo è possibile, durante la partita, tornare al *Menu Principale* o accedere alla schermata di *Selezione Livello*.

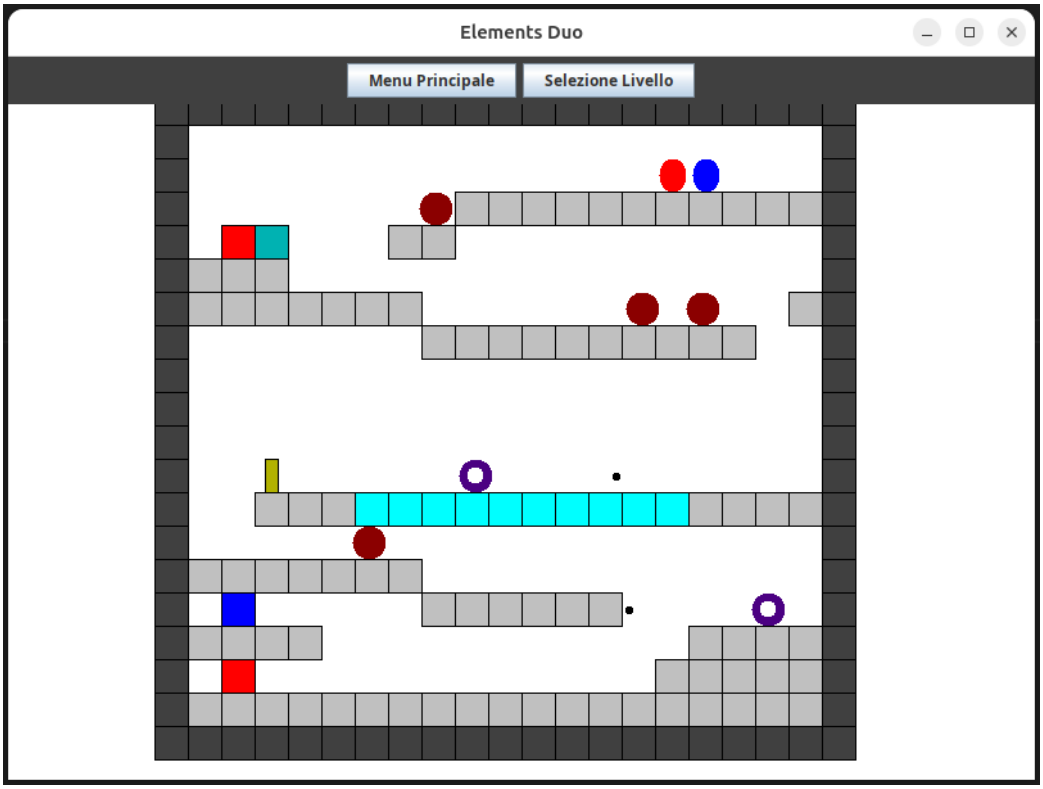

Figura A.2: Schermata Gioco

Appendice B

Esercitazioni di Laboratorio

B.1 christian.kadiu@studio.unibo.it

- Lab 06
- Lab 07
- Lab 08
- Lab 09
- Lab 10
- Lab 11
- Lab 12

B.2 leonardo.meloni@studio.unibo.it

- Lab 06
- Lab 07
- Lab 08
- Lab 09
- Lab 10
- Lab 11
- Lab 12
- Esercizio 12

B.3 Stefano.distefano3@studio.unibo.it

- Lab 06
- Lab 07
- Lab 08
- Lab 09
- Lab 10
- Lab 11
- Lab 12
- Esercizio 12

B.4 alice.ferri8@studio.unibo.it

- Lab 06
- Lab 07
- Lab 08
- Lab 09
- Lab 10
- Lab 11
- Lab 12
- Esercizio 12