Relazione progetto Crab-Inv

Mosè Barbieri, Jonathan Crescentini, Jacopo Foschi

January 2026

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il gruppo si propone di creare un videogioco chiamato "Crab Invaders" ovvero una versione rivisitata del videogioco originale "Space Invaders" [1] del 1978. Il videogioco consiste nel controllare un carro armato (il giocatore) per respingere ondate di entità ostili rappresentate da "granchi alieni" (da qui il nome).
Il videogioco è visualizzato in formato 2D, il carro armato si muove a terra (parte bassa dello schermo), mentre le entità ostili discendono dallo spazio (parte alta dello schermo). L'obiettivo del gioco è quello di respingere i nemici tramite l'arma del veicolo controllato, eliminandoli prima che possano raggiungere la posizione del carro armato ed evitandone eventuali proiettili da essi sparati.
La struttura del gioco è basata sul genere "rogue-lite", ossia ogni tentativo concluso (con successo o meno) permetterà al giocatore di utilizzare le ricompense ottenute per migliorare il proprio veicolo in modo da essere meglio preparato per il tentativo successivo.
Le partite seguono una struttura di livelli predefinita che si susseguono al compimento di quello precedente. La condizione di vittoria per poter passare al livello successivo (ed eventualmente "vincere il gioco") è quella di eliminare tutte le entità ostili prima che raggiungano la Terra, pena la perdita di una vita. La condizione di sconfitta che terminerà il tentativo è la fine delle "vite" disponibili al giocatore. È prevista la possibilità di localizzazione, in modo da permettere di cambiare la lingua dei testi mostrati.

[1] https://it.wikipedia.org/wiki/Space_Invaders

Requisiti funzionali

- Ci dovrà essere un Menu, con la possibilità di :
  - iniziare una nuova partita
  - consultare lo storico dei tentativi
  - usare lo shop
  - cambiare le impostazioni:
    - la lingua: (per il momento italiano o inglese)
    - il volume
  - uscire dal gioco
- Il carro armato controllato dal giocatore deve prevedere il movimento sull'asse laterale (sinistra, destra) e la possibilità di "fare fuoco"
- I nemici dovranno essere divisi ad ondate all'interno dei livelli, e dovranno essere capaci di muoversi verso il basso e sparare contro il giocatore.
- Al termine di ciascuna partita il nuovo stato del profilo del giocatore dovrà essere aggiornato e salvato in modo da garantire l'integrità dei dati
- Sarà possibile utilizzare le risorse acquisite durante i tentativi per potenziarsi solo ed esclusivamente tra un tentativo e l'altro, mai durante un tentativo in corso

Requisiti non funzionali

- Il gioco non dovrà avere grandi requisiti di risorse di sistema, dovrà essere facilmente eseguibile anche su macchine con prestazioni hardware ridotte

1.2 Modello del Dominio

Il gioco sarà diviso in 3 livelli ben distinti: la meta-progressione, che tiene traccia dei progressi fatti dal giocatore, la run management, che si occuperà di gestire la partita attuale (per run si intende appunto la partita), ed infine il core gameplay, costituito dalle entità in gioco.
La partita dovrà essere composta da vari livelli, inizialmente predeterminati e, se il monte ore lo permette, successivamente generati proceduralmente in modo randomico. Ogni livello sarà quindi composto da ondate che il giocatore dovrà respingere.
Il carro armato dovrà fronteggiare i "granchi alieni", sparandogli col suo cannone e schivando i colpi alieni a lui rivolti. I nemici caduti rilasceranno una valuta che verrà poi utilizzata per acquistare potenziamenti per il carro armato. Col susseguirsi delle ondate si dovrebbe incontrare una difficoltà tale da perdere se non si hanno sufficienti potenziamenti. (skill issue)
Proseguendo sempre di più nei livelli si potrà, infine, finire il gioco, scacciando definitivamente gli invasori.
Gli elementi costitutivi del dominio nella sintesi dei 3 livelli sono visibili nella figura 1.1

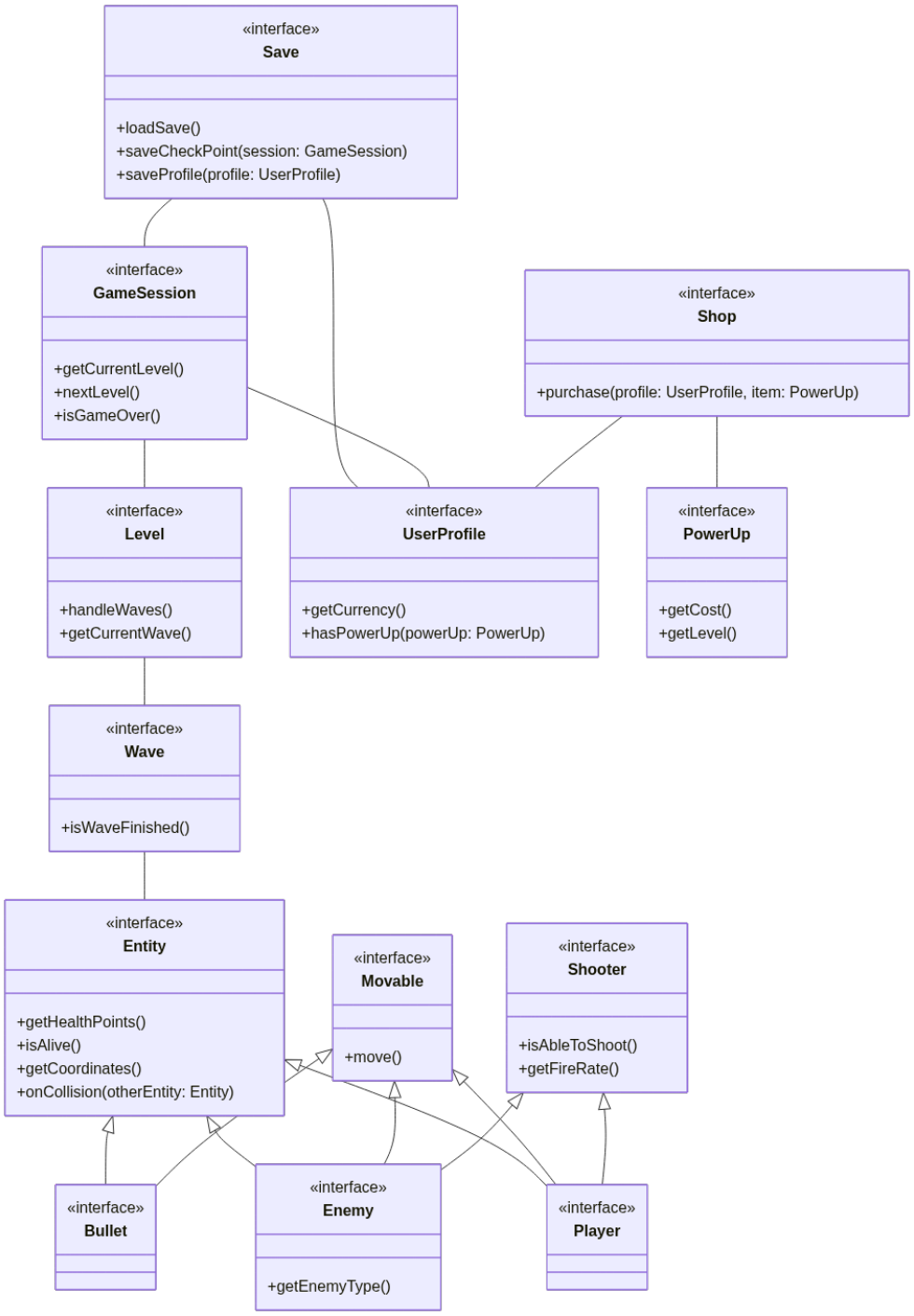

Figura 1.1: Schema UML dell'analisi del dominio, con rappresentate tutti e 3 i livelli del gioco. La meta-progression: rappresentata da Save, UserProfile, Shop e PowerUp. La Run Management: descritta da GameSession, Level e Entity. Ed infinte il livello di gameplay. Rappresentato da entity e tutto ciò che ne deriva

Capitolo 2

Design

2.1 Architettura del Sistema

L'architettura del sistema è stata progettata adottando il pattern **Model-View-Controller (MVC)**, con l'obiettivo di separare in modo chiaro le responsabilità tra gestione dello stato di gioco, rappresentazione grafica e coordinamento del flusso di esecuzione.

La suddivisione consente di ridurre l'accoppiamento tra componenti, migliorare la manutenibilità e favorire l'estendibilità del sistema.

Struttura Architetturale

Il sistema è organizzato nei seguenti tre macro-componenti:

- **Model**: rappresenta il dominio applicativo e contiene la logica di gioco e lo stato corrente.
- **View**: si occupa esclusivamente della rappresentazione grafica e dell'interazione con l'utente.
- **Controller**: coordina l'esecuzione del ciclo di gioco e media le interazioni tra Model e View.

Gli entry point principali dell'applicazione riflettono tale suddivisione:

- **GameEngine** rappresenta l'entry point del Model.
- **SceneManager** rappresenta l'entry point della View.
- **MetaGameController** rappresenta l'entry point del Controller.

Model: GameEngine

Il GameEngine costituisce il punto di accesso principale al dominio e incapsula lo stato complessivo della partita.

All'interno del Model sono gestite:

- la logica di progressione dei livelli;
- la gestione delle wave di nemici;
- l'istanziazione e il ciclo di vita degli enemy;
- l'aggiornamento della currency e dello stato del giocatore.

Il Model non dipende dalla View né dal Controller, garantendo indipendenza della logica di dominio dalla rappresentazione grafica.

View: SceneManager

Esso è responsabile della gestione delle scene e della loro visualizzazione.

Le sue responsabilità includono:

- caricamento e gestione delle schermate di gioco;
- aggiornamento della rappresentazione grafica in base allo stato del Model;
- gestione delle transizioni tra scene.

La View non contiene logica di dominio, ma si limita a riflettere lo stato fornito dal Model tramite il Controller.

Controller: MetaGameController

Il MetaGameController implementa il ciclo principale del gioco (game loop) e coordina l'interazione tra Model e View.

Le sue responsabilità comprendono:

- gestione del ciclo di aggiornamento per tutte le entità ed il mondo;
- acquisizione degli input utente;
- invocazione degli aggiornamenti sul Model;
- sincronizzazione dello stato del Model con la View.

Il Controller non contiene logica di dominio, ma orchestra il flusso di esecuzione, mantenendo separati i livelli architetturali.

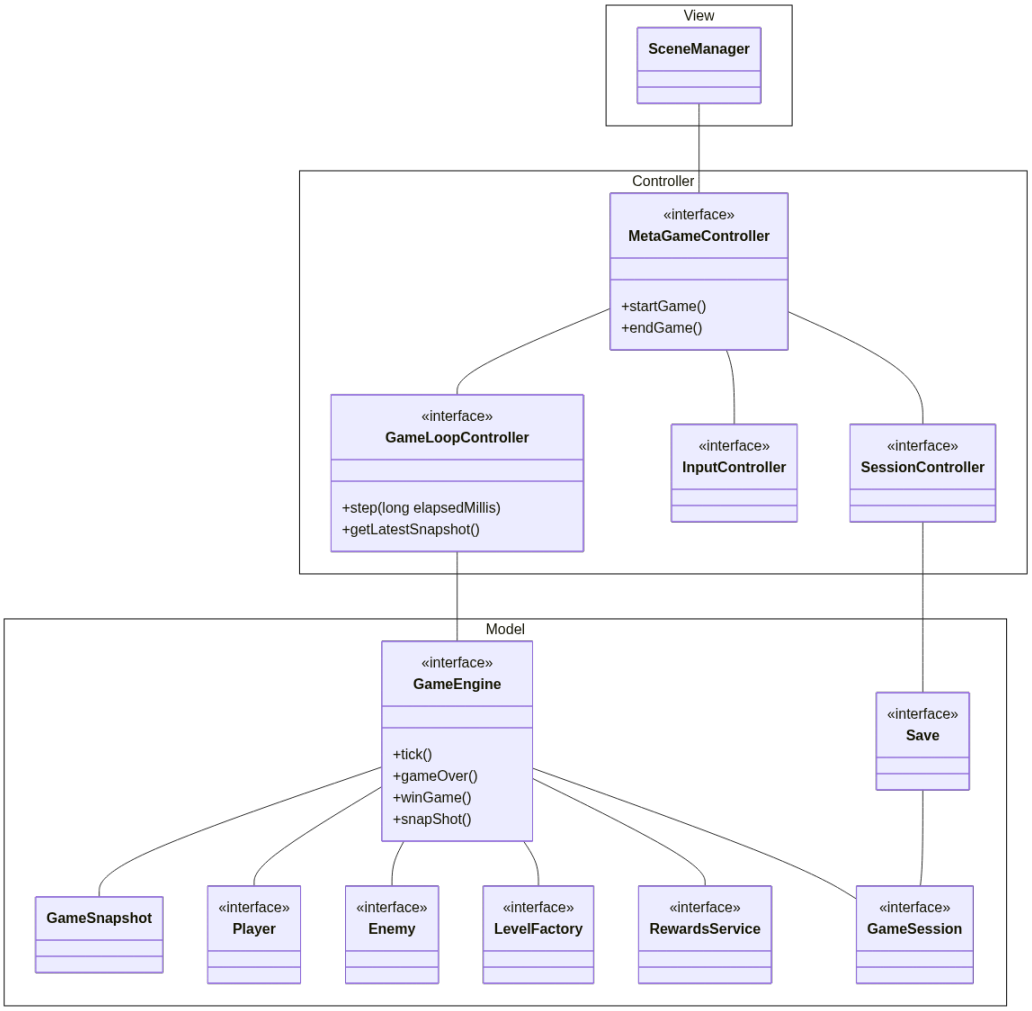

Figura 2.1: Qui lo schema UML dell'architettura. SceneManager è l'entry-point della view, che leggerà il contenuto delle altre view dichiarate mostrandolo a schermo. MetaGameController si occupa di orchestrare tutti i cambiamenti nel gioco. GameEngine gestisce i cambiamenti di stato del model

2.2 Design dettagliato

2.2.1 Design Foschi

Sincronizzazione dello stato della localizzazione

Problema Per garantire che tutte le sezioni dell'interfaccia grafica condividessero uno stato coerente della lingua, era necessario centralizzare l'accesso alle stringhe localizzate.

Soluzione provata Ho implementato il pattern singleton, perché in questo contesto è corretto avere una sola istanza di Localization per client. Questo approccio mi ha permesso di creare un'unica istanza di Localization, che funge da servizio globale per recuperare le stringhe dai ResourceBundle. In questo modo, lo stato della lingua rimaneva coerente in tutta l'applicazione.

Problemi generati L'uso di questo pattern, ha reso complicato il testing unitario, e ha reso problematica la gestione del codice qualora si volesse aggiungere il multithreading. Inoltre questo pattern imponeva l'inizializzazione con un ResourceBundle di default, in piena contraddizione con la pratica di scegliere la lingua alla prima apertura del gioco.

Soluzione La classe è stata rielaborata con l'approccio single instance by design con dependency injection. Questo permette di mantenere uno stato consistente per tutta la GUI, ma l'istanza è accessibile solo dai componenti che la ricevono esplicitamente, invece di essere disponibile globalmente. Questo rende anche molto più semplice il testing automatico, perché permette di avere situazioni di partenza diverse senza dover modificare manualmente lo stato dell'istanza.

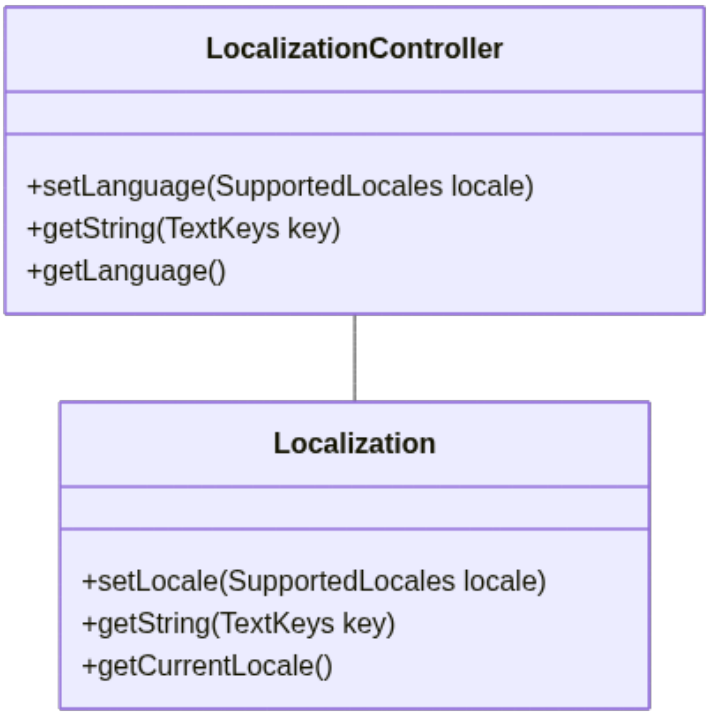

Figura 2.2: Qui lo schema UML del design della localizzazione

Gestione del suono

Problema Gestire il suono in maniera uniforme per tutta l'esecuzione del gioco senza dipendere completamente da JavaFX

Soluzione Si è fatto uso come nella sezione precedente di un singleton, anch'esso poi corretto da un approccio single instance by design con dependency injection. Per lo sviluppo è stato utilizzato il pattern facade creando quindi l'interfaccia SoundService, per fare in modo che il controller non sapesse nulla dell'esistenza di JavaFX e che potesse quindi eseguire qualsiasi model con qualsiasi libreria che implementasse l'interfaccia di facciata.

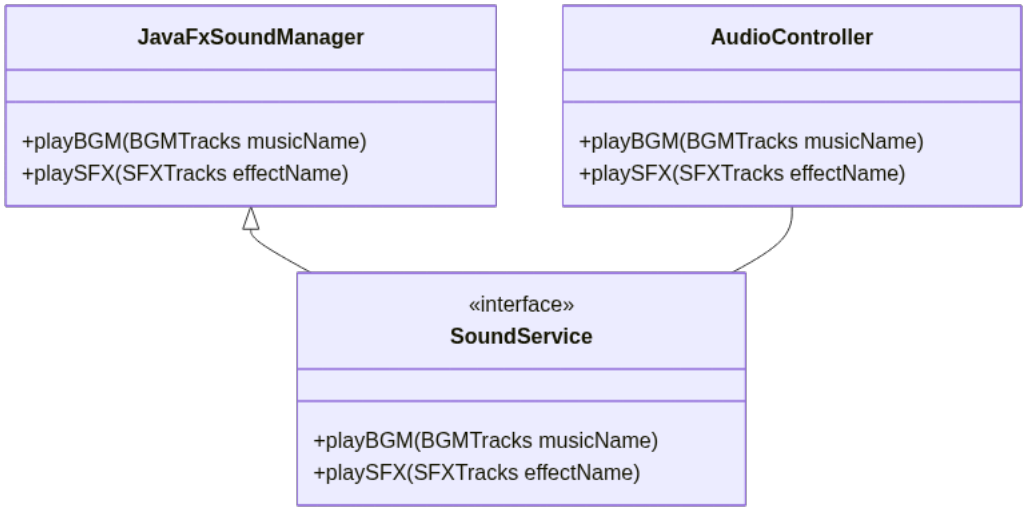

Figura 2.3: Schema UML che mostra la progettazione fatta con solo i metodi principali

Strutturazione delle entità

Problema Le entità del gioco hanno tutte un comportamento base pressoché identico e vi sarebbe perciò una grande ripetizione di codice.

Soluzione A risoluzione di ciò è stata progettata una classe astratta per tutte le entità che implementa tutti i metodi che esse devono avere, lasciando poi la possibilità a chi la eredita di specializzarla.

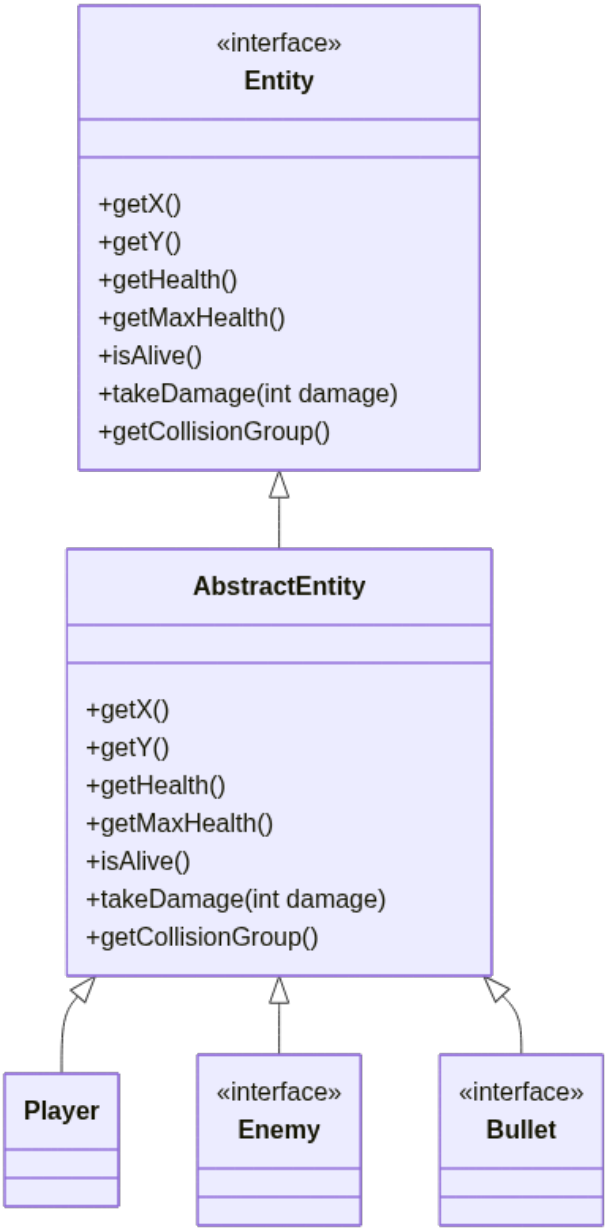

Figura 2.4: Schema UML della progettazione delle entità con alcuni dei metodi che esse implementano come esempio

Standardizzazione dei controller dell'entità

Problema In sede di sviluppo si è rilevato come i controller per le entità abbiano un comportamento pressoché identico eccezion fatta per il metodo

update. Che ha lo scopo di fornire un update dello stato dell'entità ad ogni tick.

Soluzione Strutturare un controller astratto che fornisca implementazione per tutti i metodi in comune a tutte le entità specializzando poi all'occorrenza. La differenza nel metodo update è stata invece gestita creando due interfacce apposite con la propria versione di update. Metterli entrambi in overload come metodi astratti avrebbe portato obbligatoriamente ed erroneamente a dover implementare entrambi i metodi. In questo modo invece non si è costretti a farlo mantenendo quindi il codice pulito nel suo scopo.

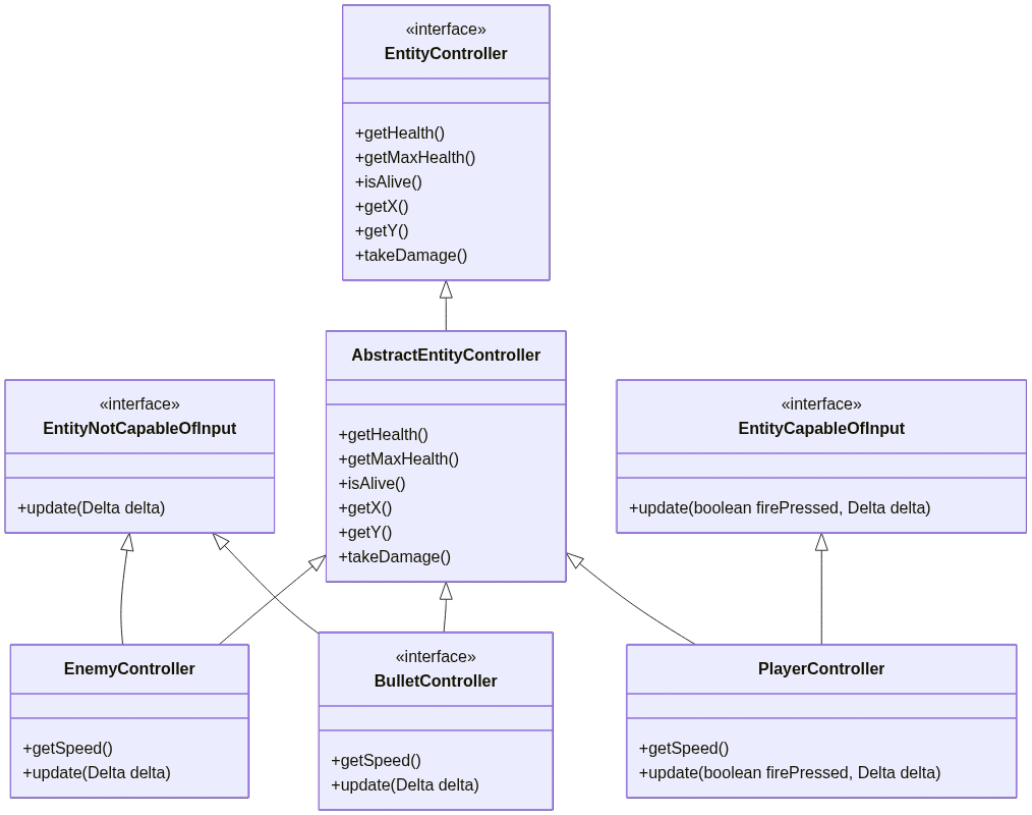

Figura 2.5: Schema UML della progettazione degli EntityController mostrando alcuni metodi come esempio

2.2.2 Design Barbieri

Gestione delle Wave

Problema: L'interfaccia fatta inizialmente non basta per gestire la progressione delle Wave pensata da noi.

Soluzione: Quindi è necessario introdurre nel dominio un WaveProvider, con l'obbiettivo di fornire Wave a Level. Questo permette di separare la logica di progressione e le singole ondate, rendendo così il modello estendibile. Il WaveProvider è modellato tramite Strategy Pattern: dove il "Context" è il Level, la "Strategy" è il WaveProvider, e la "ConcreteStrategy" saranno le effettive implementazione di WaveProvider.L'introduzione di WaveProvider consente di delegare la logica di generazione delle wave a strategie intercambiabili, garantendo l'aderenza al principio Open/Closed ed evitando modifiche alla classe Level in caso di nuove modalità di progressione.

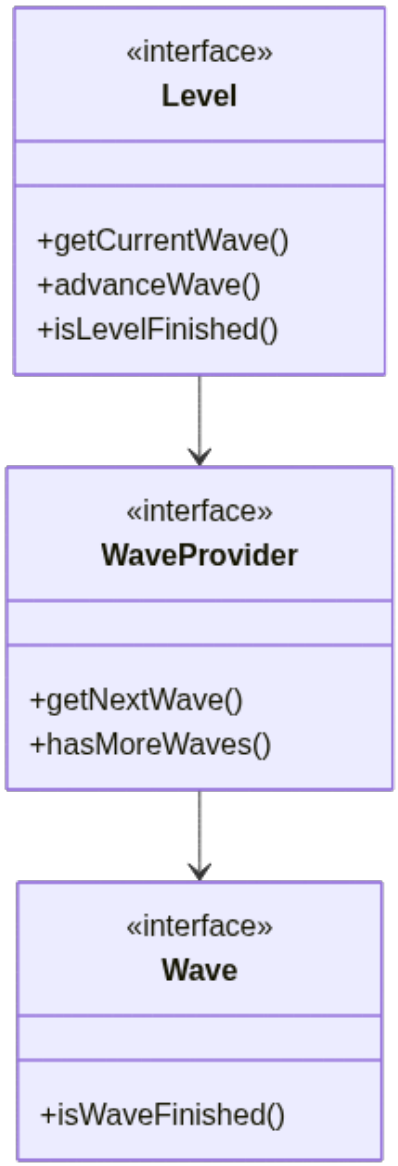

Figura 2.6: Schema UML di parte del dominio, che mostra l'interfaccia di implementazione di Wave provider

La Generazione di nemici

Problema: Nell'interfaccia iniziale esistevavo semplicemente i nemici quindi manca qualcosa che effettivamente li generi secondo il loro tipo.

Soluzione: Quindi è necessaria una EnemyFactory che prenda il tipo di enemy, basandosi sul pattern SimpleFactory. Così il "Product" sarebbe l'Enemy e EnemyFactory la "Factory" con poi le sue implementazioni che saranno il "ConcreteProduct" basandosi poi su un Enum contenente tutti i vari tipi di nemici: EnemyType, che farà da parametro discriminante. Quindi così si evita che una classe sappia troppo dell'altra e che si centralizza la logica di creazione evitando così switch sparsi nel codice.

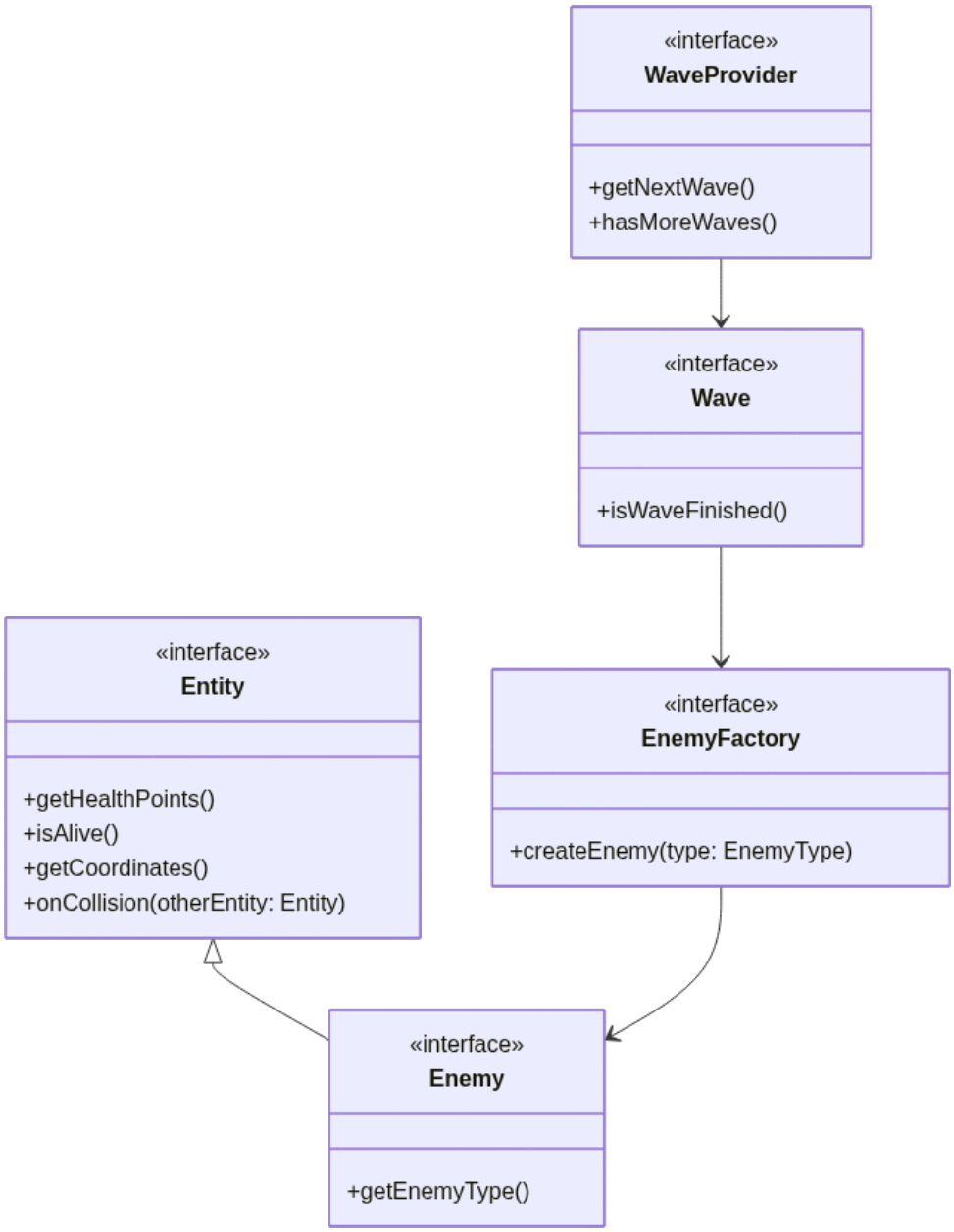

Figura 2.7: Schema UML di parte del dominio che mostra l'interfaccia di EnemyFactory e come interagisce con l'EnemyType che è l'enum con la tipologia di nemici.

La currency

Problema: All'interno del modello di dominio iniziale non è presente qualcosa che applichi l'aumento della moneta spendibile del giocatore e ci dovrebbe essere qualcosa che eviti che l'Enemy modifichi direttamente UserProfile.

Soluzione: Perciò ho ritenuto necessario aggiungere ai singoli la quantità della ricompensa e poi creare un meccanismo di notifica basato su Observer, dove il "Subject" sarà l'Enemy e l'"Observer" il RewardService che alla morte del nemico notifichi la morte allo userProfile.

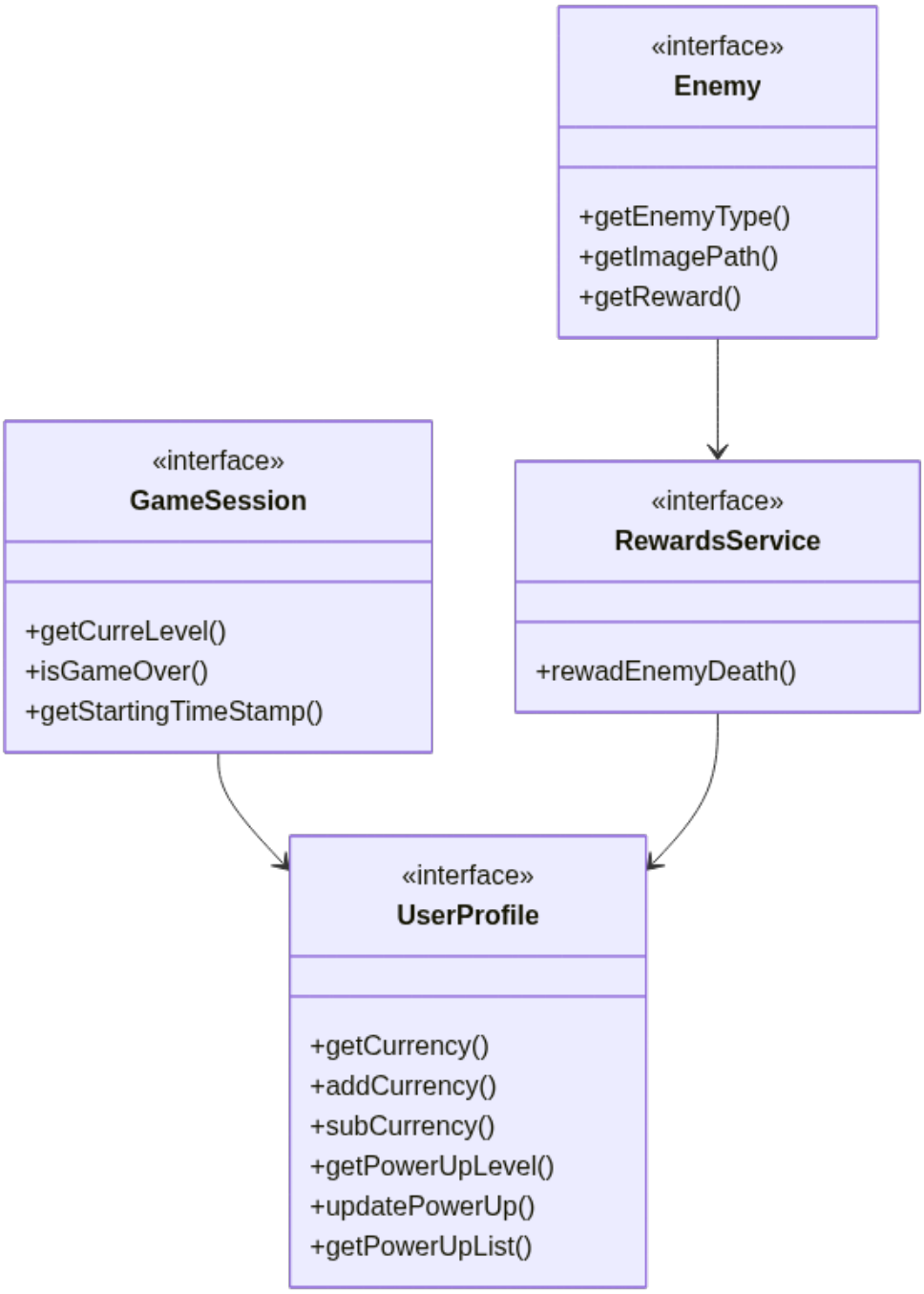

Figura 2.8: Schema UML di parte del dominio che mostra le interfacce modificate e come si collega con il resto della parte del dominio.

2.2.3 Design Crescentini

Gestione dei dati di salvataggio all'interno dell'applicazione

Problema : Necessaria una struttura per fare in modo che ogni parte della classe di salvataggio sia sempre riconducibile al proprio Save ed un modo di salvare lo stato.

Soluzione : Usato pattern Facade per far si tutte le componenti della classe di salvataggio devono essere richiamate attraverso la classe di salvataggio principale per poter essere utilizzate. Usato pattern Memento per la conservazione dello stato.

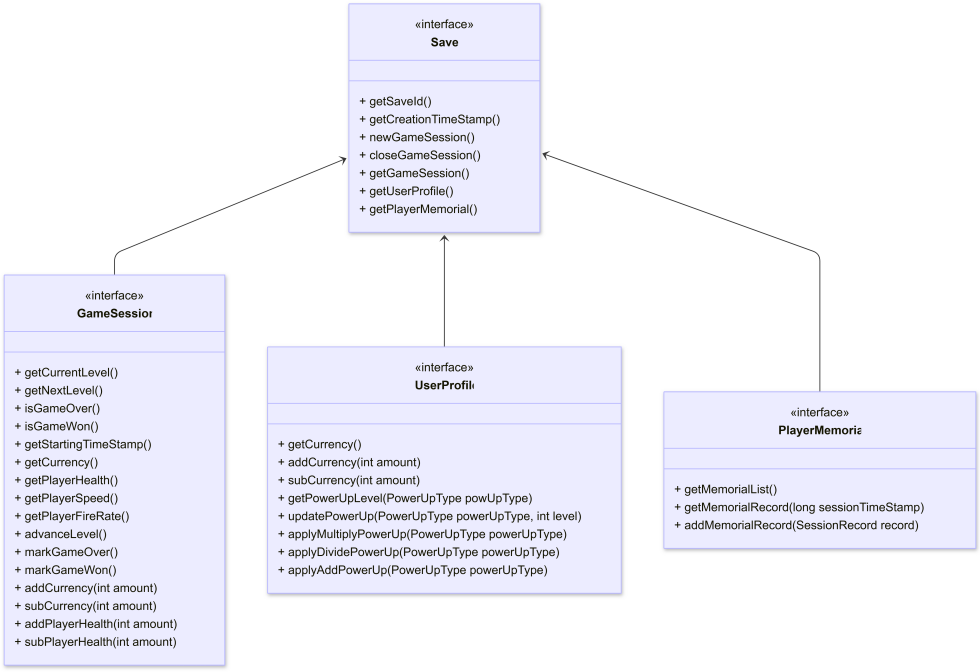

Figura 2.9: Schema UML che mostra la relazione tra Save ed i suoi componenti

Persistenza dei salvataggi fuori dall'applicazione

Problema Necessità di una soluzione per poter permettere l'esistenza dei dati di gioco anche alla chiusura dello stesso, in modo da mantenere progressi riprendibili alla nuova apertura del gioco

Soluzione Utilizzato pattern Chain-of-responsibility per separare il Save, la trasformazione da dati grezzi a dati utilizzabili dalla applicazione (o la

creazione ove fosse necessario), e il salvataggio/caricamento/gestione dei file. Usato pattern Factory in SaveFactory per assicurarsi che la implementazione di SaveRepository noo abbia effetto sul Save.

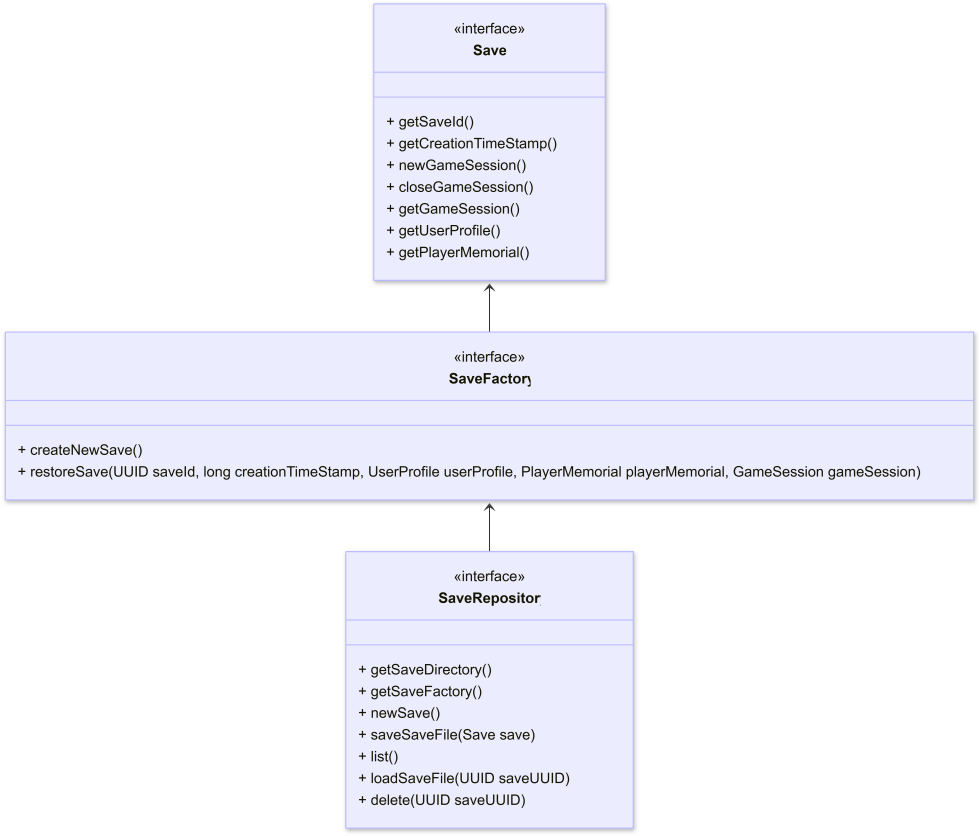

Figura 2.10: Schema UML che mostra la catena dal recupero del file al componente effettivo usato dalla applicazione

Separazione ed indipendenza dalla View

Problema Indipendenza del funzionamento dell'applicazione rispetto alla implementazione di View.

Soluzione Utilizzato pattern Facade attraverso l'interfaccia MetaGameController, quest'ultima è l'unico punto di giunzione tra la parte di View ed il resto dell'applicazione e ne "nasconde" il funzionamento facendo da intermediario.

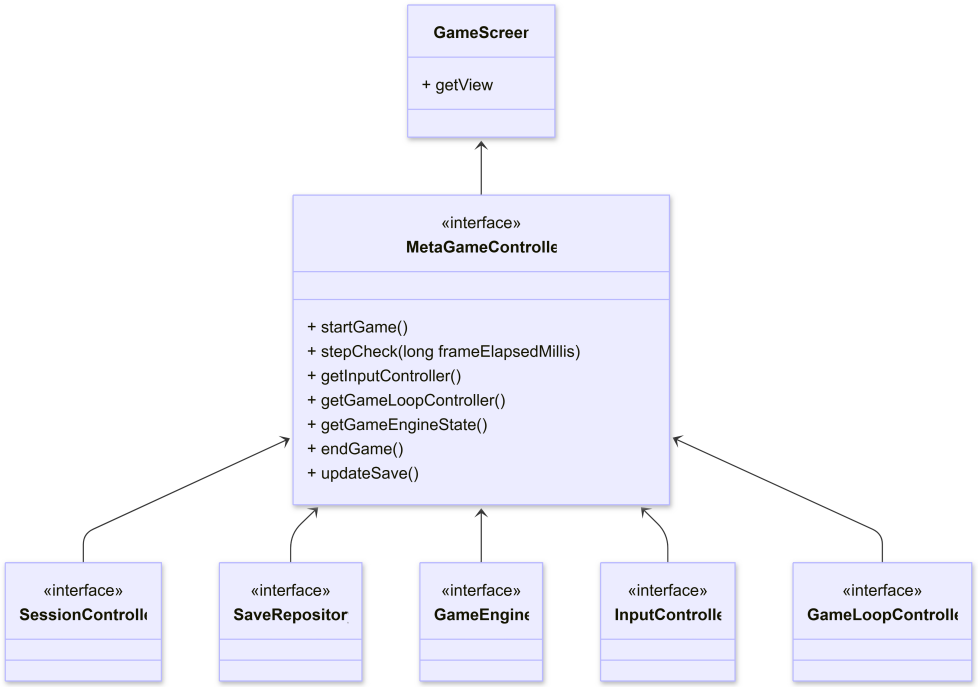

Figura 2.11: Schema UML che mostra l'interfaccia Facade MetaGameController, il suo aggancio con la View in GameScreen e tutte le altre componenti Model e Controller sottostanti ad essa

Infrastruttura di Game Engine e Game Loop

Problema Creare una struttura di Game Engine / Game Loop.

Soluzione Utilizzato il GameLoop pattern per la separazione dei ruoli. Il GameEngine contiene lo stato e la logica del gioco mentre il GameLoop esegue gli aggiornamenti e tiene traccia del tempo.

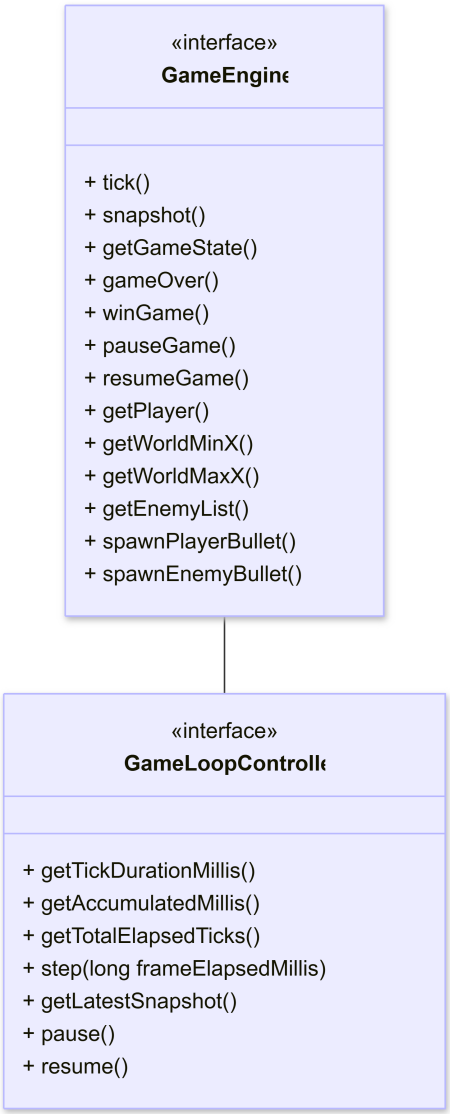

Figura 2.12: Schema UML che mostra la separazione dei ruoli tra GameEngine (Model) e GameLoop (Controller)

Player Input

Problema Gestire il flusso di input e trasformarlo in azioni di gioco.

Soluzione Utilizzato pattern Chain-of-responsibility per separare i vari passaggi dell'input, dalla acquisizione tramite HID all'azione di gioco, in modo da assicurare indipendenza dall'HID utilizzato

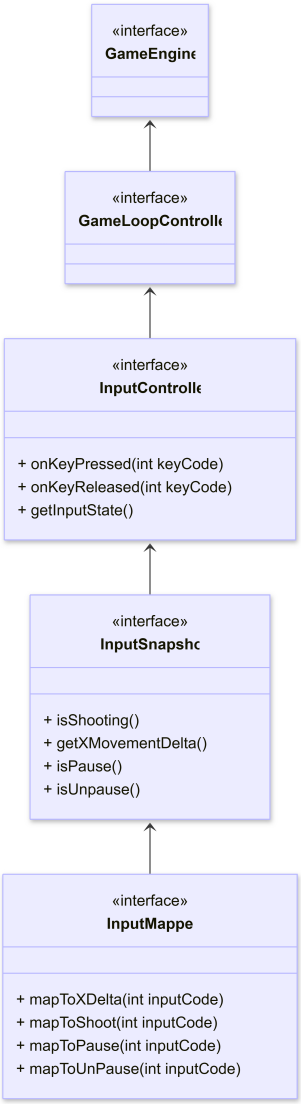

Figura 2.13: Schema UML che mostra la catena di input (Nota: GameEngine e GameLoopController sono visualizzati in formato ridotto poichè non partecipano all processo di input ma sono il punto di arrivo)

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il testing automatico è stato sproporzionato nei vari membri del gruppo, dove alcuni ne hanno fatto meno uso e in maniera meno estensiva. Si è cercato di praticare il TDD ma la grande maggioranza dei test è stata fatta in modalità test-after.
Soggetto dei test sono state le parti di model e di controller. Con test appositi anche per la persistenza dei dati e quindi dei salvataggi. A questo fine si è fatto uso della libreria JUnit e in un'istanza specifica, ovvero per il testing del controller dell'audio, anche della libreria Mockito. Questa libreria è stata utilizzata in quanto alcuni controller fanno utilizzo di model che devono avere uno stato ben definito che si affidano anche ad esempio a JavaFX, in questo modo è stato possibile controllare logicamente in contesto di mocking che il controller funzionasse adeguatamente.
Non è stato fatto testing automatico della view JavaFX perché sarebbe stato necessario fare uso di una libreria che non è mai stata trattata nel corso, il cui apprendimento e successiva integrazione non ci è stato possibile nel monte ore prefissato.

3.2 Note di sviluppo

3.2.1 Note di sviluppo Foschi

Utilizzo della libreria JavaFX

Utilizzata per fare la view del gioco e la gestione dei suoni. Qui un esempio Permalink

Utilizzo della libreria Mockito

Utilizzata per fare mocking nel testing Permalink

Utilizzo della libreria Lombok

Utilizzata per automatizzare la creazione dei builder delle entità Permalink

Utilizzo dei resource boundle

Utilizzati per garantire un supporto multilingua efficiente Permalink

Utilizzo di method inference

Usati in più parti del codice, qui un esempio. Permalink

Utilizzo di lambda expressions

Per fornire i metodi necessari senza ricorrere a classi anonime. Utilizzato in varie parti del codice, qui un esempio. Permalink

Utilizzo della libreria Gson

Per gestire la permanenza delle impostazioni del gioco Permalink

Utilizzo del costrutto record

Per facilitare il salvataggio delle impostazioni del gioco Permalink

3.2.2 Note di sviluppo Barbieri

Utilizzo della libreria JavaFX

Utilizzata per creare il menu Permalink

Utilizzo di Stream

Utilizzati per ciclare su tutta la lista di nemici Permalink

Utilizzo di Method Inference

Utilizzato inseme allo stream Permalink

Precisazione

I due permalink precedenti sono considerati, anche a livello di git blame di Crescentini. Ma è perché ha riadattato il codice in un nuovo file per sistemare l'interazione con il Game Engine. Questo è il commit dove l'ho fatto io sul file iniziale: CommitLink

Parti riadattate da codice generato:

Alcuni metodi specifici della view, come il createMenuButton: Permalink
Ed Il metodo createPowerUpCard: Permalink
Tutte le parti fatte da me all'interno di GameEngine o che comunicano con esso.

3.2.3 Note di sviluppo Crescentini

Utilizzo della libreria Gson

Utilizzata per la permanenza dei dati di salvataggio in file in formato JSON Permalink

Utilizzo della libreria JavaFX

Utilizzata per la creazione della interfaccia di gioco e memorial.
MemorialScreen: Permalink
GameScreen: Permalink
GameRenderer: Permalink

Utilizzo degli Stream

Utilizzati per la gestione dello stato degli input da parte dell'utente. Permalink

Utilizzo dei Record

Utilizzati per il salvataggio dei dati delle sessioni di gioco strettamente necessari al memorial e per il passaggio dei dati di render.
SessionRecordImpl: Permalink
GameSnapshot: Permalink
RenderObjectSnapshot: Permalink

Utilizzo di AI-LLM per assistenza

È stata utilizzata IA per la generazione di parti della View. In generale è stata usata per ottenere prime bozze di codice ove necessario, non come copia-incolla ma capire come procedere in alcuni punti, è stato poi riadattato e refactor-ato manualmente per assicurare continuità e correttezza all'interno del progetto. Unica eccezione sono i Test con blame sul sottoscritto che sono stati principalmente generati con IA. I permalink portano alle due classi la cui struttura di base è stata generata (supervisione umana sempre e comunque presente).
Permalink
Permalink

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Autovalutazione Foschi

Lo svolgimento di questo progetto è stato complicato. Non avevamo mai fatto un lavoro di così grandi dimensioni e mi sono trovato in seria difficoltà all'inizio ad orientarmi in tutti gli step di progettazione del software.
Lo sviluppo è stato ulteriormente complicato dal fatto che, a causa di altri impegni dei miei compagni, mi sono trovato per praticamente tutto lo sviluppo ad essere qualche giornata di lavoro avanti rispetto a loro. Coordinarsi è quindi stato più difficile di quanto avrei voluto e mi sono trovato molto spesso a strutturare framework che sono stati poi utilizzati dai miei compagni. Ciò ha purtroppo influito anche con la mia incapacità di portare a termine alcuni obiettivi opzionali che mi ero preposto. A causa di queste circostanze mi sono impegnato a produrre del codice il più facile da implementare o utilizzare possibile, cosa che mi ha permesso di fare molta esperienza sul campo di ciò che dovrebbe essere del codice ben fatto. Riguardo alla gestione del tempo a malincuore non è stato possibile consegnare entro la deadline. Ritengo che ciò sia dovuto ad una sottostima fatta nella complessità e tempo necessario per lo svolgimento di alcuni punti fondamentali quali il gameloop dettata dalla nostra inesperienza. Sono tutt'ora insoddisfatto di ciò, ma mi servirà da lezione per eventuali progetti futuri.

Autovalutazione Barbieri

Il progetto è stato complesso. Avevo fatto qualcosa di progettazione, in altri contesti, ma come junior dev: solo a livello di richieste e poi implementazione quindi niente di profondo. Al termine di questo progetto mi sento ovviamen-

te più preparato ad un lavoro di effettiva analisi iniziale e solo poi arrivare al codice effettivo.
In generale penso di poter fare meglio, a livello di codice ed anche di progettazione, a livello di pattern sono riuscito a trovare qualcosa che potesse essere utile al progetto ma comunque non avendo un'esperienza tale da permettermi di giudicare tutto quello che ho fatto, obiettivamente, io penso di aver fatto quello che potevo fare nel tempo a disposizione. Sicuramente ho migliorato la mia capacità di progettazione e la valutazione delle tempistiche. Nel complesso è stata un'esperienza dura, ma che ha formato

Autovalutazione Crescentini

Lo sviluppo del progetto è stato lungo, complesso, difficile e frustrante. Poiché questa è stata, personalmente, la prima effettiva esperienza di progettazione di un software completo ci sono state molte difficoltà sia a livello implementativo che, soprattutto, a livello di progettazione. Per le difficoltà riscontrate e che continuamente emergevano durante il processo è stato impossibile rispettare la deadline.
Per quanto sia stata affrontata una parte di progettazione e design preventiva, alla effettiva implementazione è stato necessario riadattare diverse idee per poter rispettare la logica di Java o comunque è stata chiara una sottostima dei tempi di sviluppo. Molto probabilmente un eventuale prossimo progetto risulterebbe in un risultato migliore sia dal punto di vista di qualità di codice che di strutturazione del lavoro e concepimento di un design più direttamente applicabile, grazie al processo costante di trial-and-error necessariamente affrontato per la creazione di questo applicativo.

4.2 Difficoltà incontrate e commenti per i docenti

Come gruppo ci siamo trovati impreparati ad affrontare un lavoro di progettazione di questo livello. Perché riteniamo che nell'effettivo non ci siano stati ripartiti sufficienti insegnamenti a riguardo. Abbiamo perciò dovuto imparare molte cose in corso d'opera, portandoci anche a commettere errori altrimenti evitabili. Ci siamo sentiti costretti a ricorrere a LLM e guide online, non tanto per l'implementazione del codice, quanto per chiedere consigli di progettazione, informandoci su ciò che viene normalmente applicato allo stato dell'arte. Ci sentiamo di consigliare caldamente ai docenti di porre una maggiore attenzione quindi alla progettazione del codice in sé, per quanto possiamo capire che i limiti di tempo sono quelli che sono, sarebbe di grande

aiuto per gli studenti futuri.
Riguardo ad altre difficoltà incontrate, abbiamo pagato a caro prezzo il non aver aggiunto e utilizzato il plugin di quality assessment sin dal principio. Questo ha portato ad un ulteriore ritardo nella consegna, dovuto alla necessità di correggere tutti gli errori di stile. Consigliamo quindi di essere più specifici sull'importanza e le ragioni per qui è meglio basarsi sul template fornito, anche se si sa già creare un progetto da zero. Spiegando o accennando anche solo brevemente i plugin da usare in sede di build.

Appendice A

Guida utente

Note: In quanto immaginiamo che avere cartelle inutili possa causare fastidio notifichiamo ai professori che l'utilizzo di questa demo porterà alla creazione di una cartella .crabinvaders nella propria userFolder.

- La navigazione di questo gioco va svolta esclusivamente tramite la tastiera
- Scorrere i menù preferibilmente con Tab, ma anche con le frecce direzionali. Selezionare le opzioni con lo Spazio o il tasto Invio
- Nella storia delle partite scorrere la lista con le frecce direzionali, per distogliere il focus e usare il pulsante di ritorno al menù usare Tab
- Controllare il giocatore con le frecce direzionali sinistra e destra, sparare con Spazio.
- Per mettere in pausa durante il gioco premere Esc e usare Tab per distogliere il focus dal Canvas