LastCrown

Giulia Albertini, Marco Battistini, Riccardo Carta

10 agosto 2025

Indice

Analisi

1.1 Descrizione e Requisiti

Il software dovrà essere un gioco di carte strategico ispirato alla tipologia "Tower Defense", in cui il giocatore deve schierare le proprie difese (personaggi e magie) per difendere il proprio villaggio dall'avanzare dei nemici.
All'inizio della partita, il giocatore avrà la possibilità di scegliere un eroe, che rappresenta una figura centrale nella sua strategia di difesa. Ogni eroe determina il tipo di approccio da adottare nella costruzione del mazzo e nelle scelte tattiche. Infatti, in base all'eroe, il giocatore avrà la possibilità di comporre un mazzo personalizzato, selezionando un certo numero di carte suddivise per categoria (Mischia, Distanza, Magia).
Ogni partita sarà suddivisa in round di difficoltà crescente, al termine di ciascuno, il giocatore potrà acquistare potenziamenti in un negozio dedicato. Per potenziamenti si intende nuove carte che aumenteranno le probabilità di sopravvivenza per la prossima ondata.
L'obiettivo principale del gioco è mantenere in vita l'eroe e la sua linea di difesa, resistere agli attacchi dei nemici e cercare di arrivare a più round possibili.
Ogni partita offrirà nuove sfide e una grande varietà di strategie, rendendo ogni esperienza di gioco avvincente

Requisiti Funzionali

- Tutti i personaggi coinvolti nella partita dovranno avere diverse animazioni a seconda della situazione in cui si trovano e/o l'azione che stanno compiendo
- I nemici dovranno essere di varia natura, da quelli più deboli fino ad arrivare ai temibili boss che concludono ogni ondata.
- Al di fuori della partita, il giocatore potrà controllare le proprie statistiche di gioco (n° di round vinti, monete in possesso, tempo di gioco totale...) e gestire la propria collezione di carte costruendo il mazzo, scegliendo accuratamente tra quelle disponibili per aumentare la probabilità di successo contro i nemici.
- Il mazzo di carte del giocatore potrà essere gestito in un'interfaccia dedicata, scegliendo dalla propria collezione di carte.
- Quando l'applicazione viene chiusa, all'accesso successivo il giocatore vedrà mantenuti i propri progressi di gioco.
- Ogni sezione principale dell'applicazione (menù iniziale, partita, negozio...) avrà una colonna sonora dedicata.

Requisiti non Funzionali

- L'applicazione dovrà garantire una frequenza di aggiornamento dell'immagine tale da risultare fluida e dovrà rispondere in maniera apprezzabile agli input dell'utente.
- Il software sarà portabile su tutti i maggiori sistemi operativi

- L'applicazione dovrà rispettare il pattern architetturale MVC, per risultare più chiaro e facilmente manutenibile in vista di future modifiche.

1.2 Modello del Dominio

Last Crown sarà suddiviso in più sezioni: il menù iniziale, l'interfaccia della partita, del negozio e della gestione del mazzo di carte. Il menù iniziale permette al giocatore di creare il proprio account per la prima volta oppure di accedere se lo possiede già. Una volta che il login è avvenuto con successo, al giocatore vengono presentate diverse possibilità: controllare le proprie statistiche di gioco (per esempio numero di partite giocate...), gestire la propria collezione di carte e/o modificare il proprio mazzo o infine cominciare una partita vera e propria.
Ogni partita a sua volta è organizzata in due fasi principali che si alternano: il match, in cui il giocatore deve fronteggiare l'avanzata dei nemici schierando le proprie difese (truppe/magie) e una fase di "riposo", dove il giocatore può interagire liberamente tra alcuni commercianti, acquistando nuove carte per avere più possibilità di successo nell'ondata successiva. Ognuna di queste due fasi sarà ben riconoscibile, in quanto caratterizzata da un'interfaccia di gioco unica. L'interfaccia "match" sarà organizzata in diverse sezioni. Rispettivamente da sinistra verso destra: zona delle carte del mazzo, zona di collocamento dei personaggi di tipo distanza, di tipo mischia (compreso l'eroe), muro difensivo, campo di battaglia dove avanzano i nemici (da destra verso sinistra).

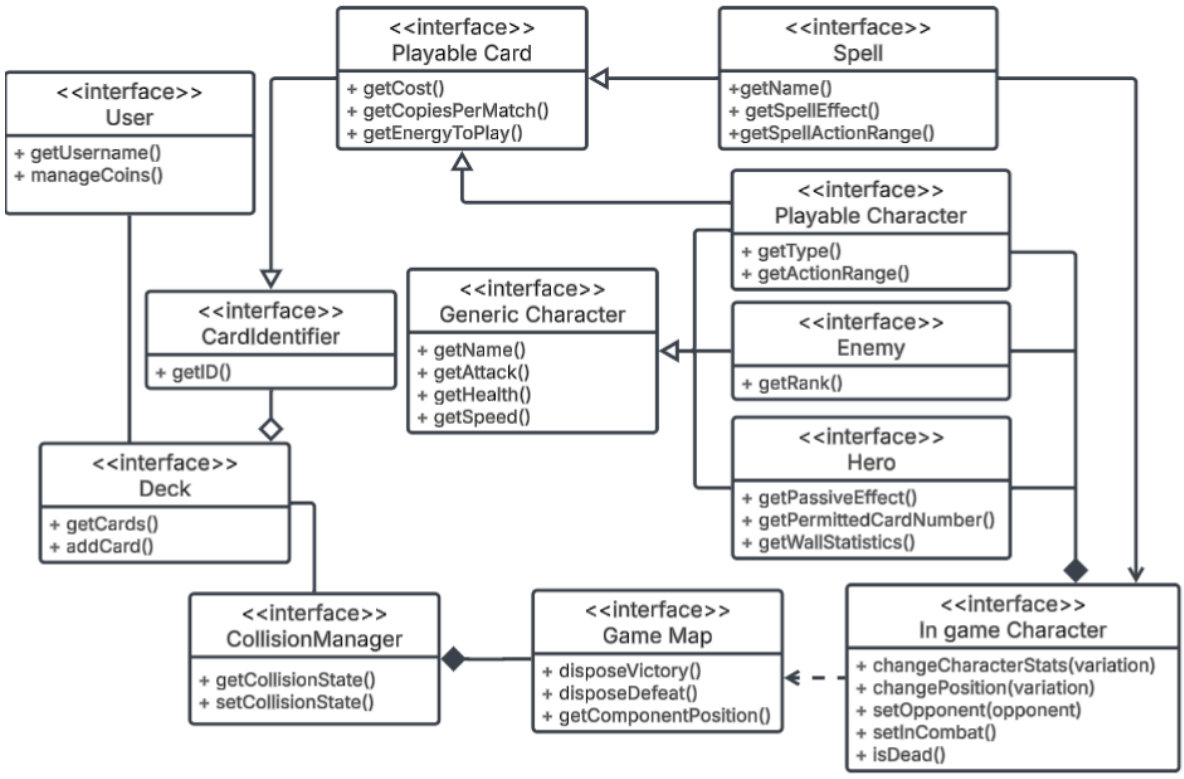

Figura 1: Modello del Dominio

Capitolo 2

Design

2.1 Architettura

Il software si basa sull'architettura MVC, e rispetta quindi tutte le funzionalità previste:

- L'utente interagisce con la View (per esempio cliccando un bottone).
- Il Controller riceve l'input, lo elabora e decide l'azione da intraprendere.
- Il Controller aggiorna il Model, se necessario.
- Il Model cambia eventualmente il proprio stato (per esempio modificando le carte della collezione dell'utente oppure cambiare l'attacco e/o la salute corrente di un personaggio in partita)
- La View si aggiorna per riflettere il nuovo stato del model.

Nel nostro caso specifico, data la dualità dell'applicazione, statica in fase di menù oppure dinamica in fase di partita, si è preferito adottare adottare una declinazione particolare di MVC in cui ci sono due controller principali, che costituiscono infatti i due entry point principali di questa sezione. Per la componente "statica" l'entry point è il "MainController" che si occupa di gestire le classi necessarie al login, inizializzare l'account del personaggio che è stato selezionato durante la fase di login e passare alla "MainView" dove è presente il menu principale.
Per la componente più dinamica invece l'entry point è rappresentato dal "MatchController", il quale si occupa di gestire le fasi principali del match, come per esempio la comparsa dei personaggi giocabili (PlayableCharacter), dei nemici oppure l'inizio della bossfight.
Per quanto riguarda la View l'entry point è la sopra citata MainView, la quale si occupa principalmente di gestire il cambio di schermata mostrata, quando richiesto dal controller, ma anche di gestire gli input dell'utente tramite alcune sottoclassi da lei dipendenti.
L'unica classe considerabile come entry point del modello è quella dedicata all'utente ("Account") che si occupa principalmente di modificare i dati di gioco dell'utente, quando richiesto dalla sezione controller dell'applicazione. Tutte le altre classi del modello invece vengono utilizzate in determinate fasi dell'applicazione, come per esempio quelle dedicate ai personaggi, utili per esempio in fase di gestione del mazzo oppure controllo della collezione.
La struttura MVC realizzata permette al Model di non essere minimamente influenzato dalla View, in quanto quest'ultima non interagisce mai direttamente con il model (per esempio la non cambierà mai direttamente le statistiche di un personaggio in partita). Nemmeno il controller dovrebbe essere soggetto a cambiamenti drastici in seguito a delle modifiche apportate alla View, a patto ovviamente di realizzarne una versione che rispetti i metodi e i parametri delle interfacce originali.

2.2 Design Dettagliato

2.2.1 Giulia Albertini

Collisioni e interazioni

- **Problema**: Nel contesto di gioco, i personaggi devono poter rilevare e gestire correttamente le collisioni con altre entità, generando eventi coerenti con la natura dell'interazione.
- **Soluzione**: Ogni entità è dotata di una hitbox e di un raggio, modellato tramite la classe *Radius*. Il rilevamento delle collisioni è affidato all'interfaccia *EntityTargetingSystem*, che mantiene una mappatura delle entità e delle rispettive hitbox. Sfruttando il raggio di ciascuna entità, il sistema individua tutte le hitbox nelle vicinanze di un giocatore e seleziona quella più prossima, generando un evento specifico in base al tipo di entità rilevata. Per garantire la massima flessibilità, i partecipanti alla collisione sono modellati come oggetti di tipo *Collidable*, ciascuno caratterizzato da un *CardIdentifier* e da una *Hitbox*. L'evento di collisione, rappresentato dalla classe *CollisionEvent* e contenente i due *Collidable* coinvolti, viene propagato tramite il pattern *Observer*. In questo schema, il *CollisionManager* agisce come *observable* tramite il *MatchController*, emettendo l'evento e notificando al *CollisionResolver*, il quale, in qualità di *observer*, indirizza l'evento al resolver appropriato in base alle tipologie delle entità coinvolte.

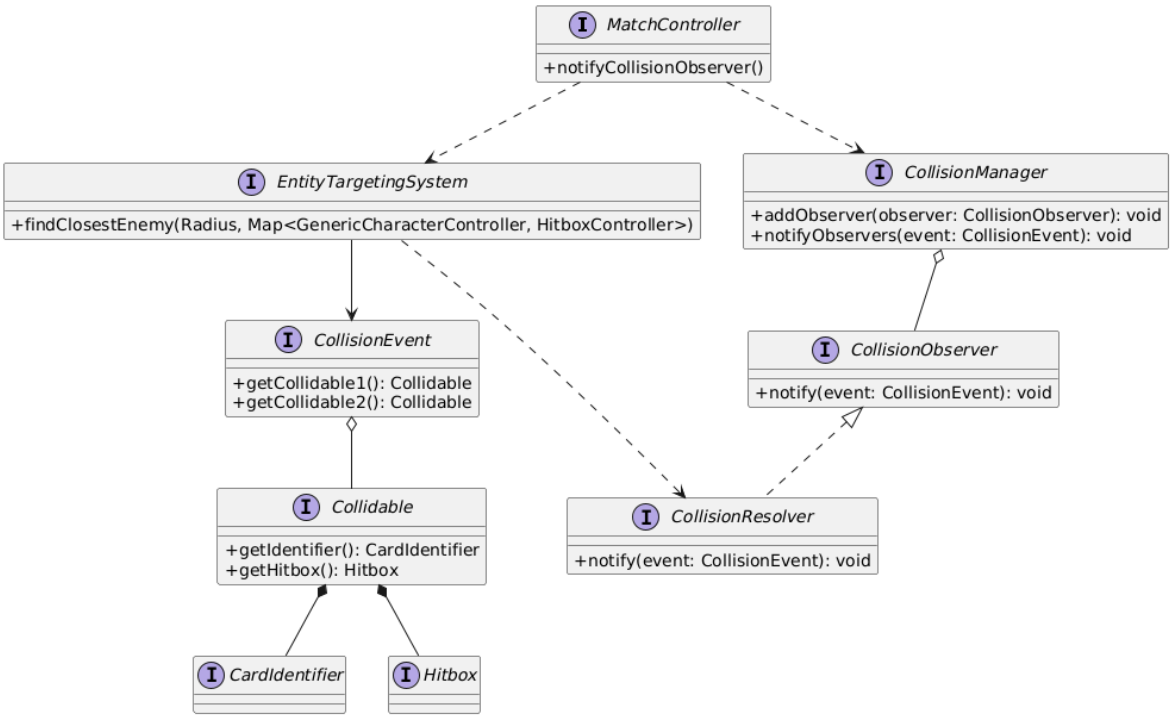

Figura 2: Schema UML delle classi e interfacce coinvolte sistema di collisione con il pattern Observer

- **Problema**: Ogni personaggio è rappresentato visivamente tramite un pannello contenente un'immagine che non ritrae solo il personaggio, ma anche un'area vuota circostante.
- **Soluzione**: Per gestire le collisioni tra i personaggi, è stata creata la classe *Hitbox*. Per determinare l'area esatta occupata dall'entità nel pannello, la classe *HitboxMask* analizza il canale alfa dell'immagine del personaggio, identificando i bordi della hitbox. In questo modo, viene generata una maschera che delimita il personaggio, dove l'area opaca rappresenta effettivamente il corpo del personaggio. Questo approccio consente una gestione precisa delle collisioni tra le entità. In fase di progettazione, per mantenere una netta separazione delle responsabilità nel codice, la posizione non è stata assegnata direttamente al personaggio, ma alla sua hitbox. Questa scelta ha reso più complicata la gestione delle collisioni nelle fasi successive dello sviluppo.

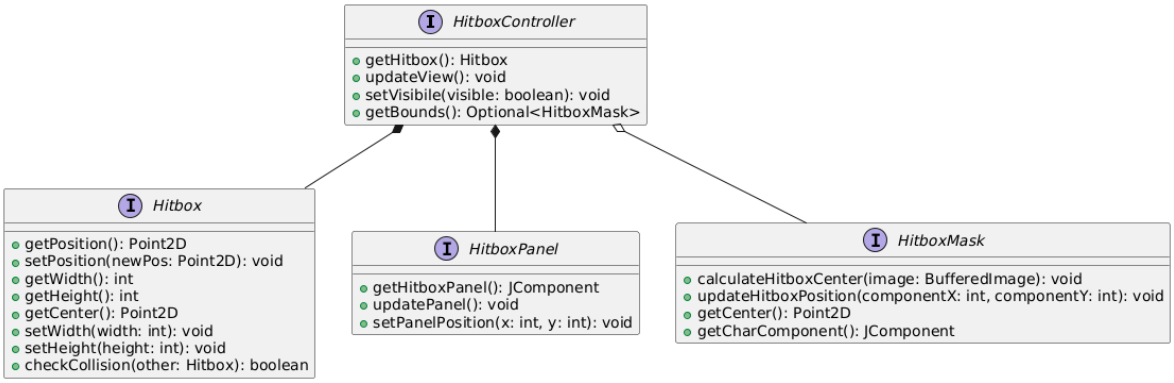

Figura 3: Schema UML delle classi e interfacce coinvolte nella creazione della hitbox specifica per il personaggio

- **Problema**: Ogni personaggio nel gioco segue un ciclo di vita identico, composto da una serie di fasi che si ripetono in sequenza durante il gioco. L'obiettivo è modellare questo ciclo in modo che possa essere eseguito in maniera automatica e ripetitiva per ciascun personaggio, mantenendo la coerenza e l'integrazione con le dinamiche di gioco.
- **Soluzione**: Il ciclo di vita di un personaggio nel gioco è gestito tramite uno stato che può cambiare dinamicamente a seconda degli eventi. Ogni personaggio può trovarsi in uno di questi stati: IDLE (inattivo), FOLLOWING (sta seguendo un obiettivo), COMBAT (combattimento), STOPPED (fermo), o DEAD (morto). Gli eventi vengono generati tramite la classe *EventFactory*, che associa ogni stato del personaggio a un gestore specifico *CharacterState Handler*. Ogni gestore definisce le azioni e le transizioni per lo stato in cui il personaggio si trova. La coda di eventi *EventQueue* raccoglie gli eventi in arrivo e li elabora uno alla volta. Quando un evento è processato, modifica lo stato del personaggio. Quest'ultimo è gestito da un *StateHandler* che determina la logica di transizione tra gli stati, come muoversi verso un obiettivo, combattere un nemico o fermarsi. Il ciclo termina quando il personaggio entra nello stato DEAD, segnando la sua "morte" e

l'assenza di ulteriori azioni. Questo processo permette una gestione fluida e dinamica del comportamento dei personaggi, con eventi che influenzano direttamente il loro stato e le loro azioni nel gioco.

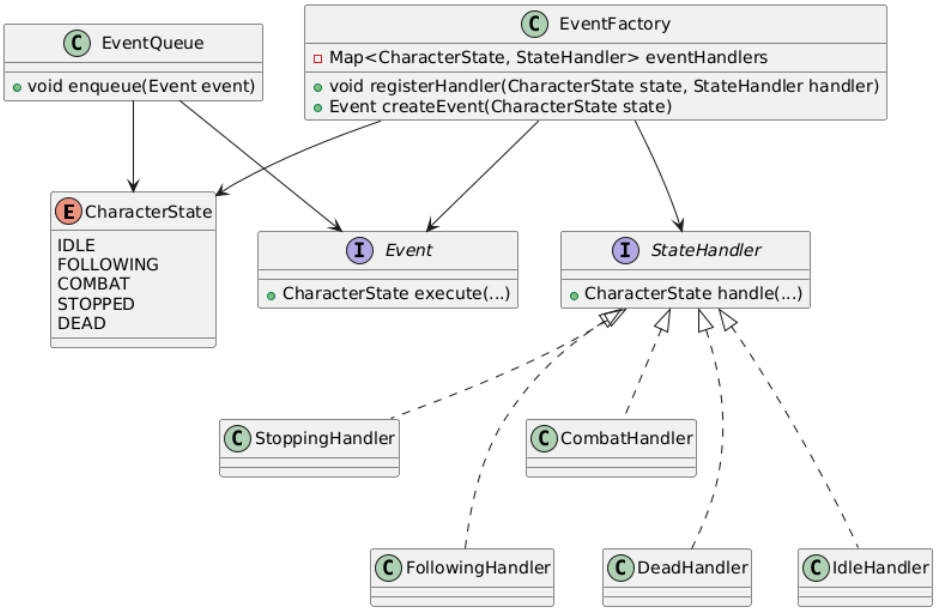

Figura 4: Schema UML delle classi e interfacce coinvolte nella gestione degli eventi

- **Problema**: Creare una struttura centralizzata che coordini le entità nel gioco. Molteplici oggetti devono essere inizializzati, monitorati e aggiornati in tempo reale durante il combattimento. Inoltre, il gioco deve essere in grado di gestire eventi come collisioni, ingaggi e la fine della partita in modo efficiente e sincronizzato, senza compromettere le prestazioni.
- **Soluzione**: La classe *core* del gioco è il *MatchController*. Quest'ultimo orchestra le interazioni tra i vari sistemi specializzati come l'EnemyEngagementManager, l'EnemySpawner e l'EntityManager. Durante il ciclo di gioco, ogni aggiornamento viene processato dalla chiamata del GameLoop per gestire gli ingaggi e aggiornare la vista.

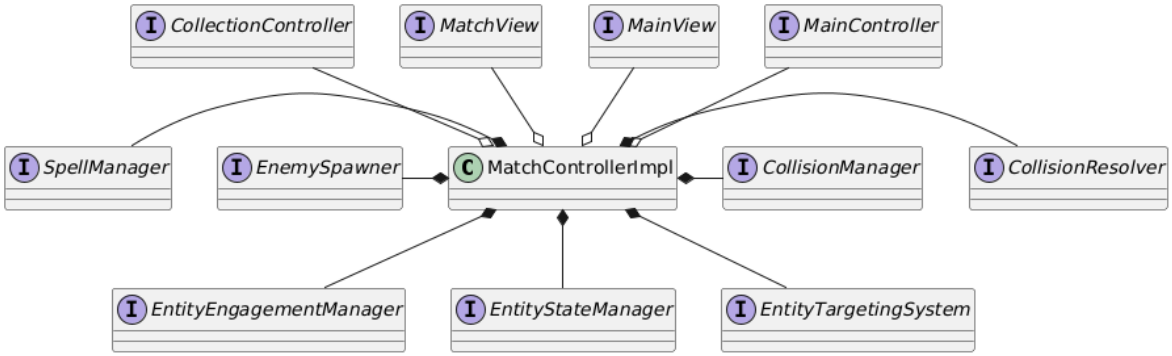

Figura 5: Schema UML delle interfacce coinvolte nella classe cuore

- **Problema**: Gestire il combattimento per i diversi tipi di entità.
- **Soluzione**: Il sistema di combattimento si basa su una struttura ad eventi, in cui ogni scontro è univoco: un solo player può combattere contro un singolo nemico alla volta. Questo vincolo è gestito tramite l'*EntityEngagement-Manager*, che sincronizza le coppie player-nemico prima dell'attivazione dell'evento. Il combattimento viene attivato tramite gli eventi di collisione notificati al *CollisionResolver*. I melee seguono una traiettoria tracciata dalla curva di *Bézier* verso il nemico, gestita da *HandleFollowEnemy,* e una volta a contatto, i *Collidable* vengono inseriti in una lista controllata dai *CharacterStateHandler* per aggiornare lo stato del personaggio. I ranged, invece, attaccano da fermo: anche in questo caso i collidable vengono tracciati e monitorati per attivare i relativi cambiamenti di stato. Nel caso dei boss, viene ampliato il raggio d'azione dei player per intercettare la hitbox più facilmente, permettendo così il combattimento simultaneo di più player. Allo stesso modo, anche la collisione con i muri consente l'interazione con più nemici contemporaneamente.

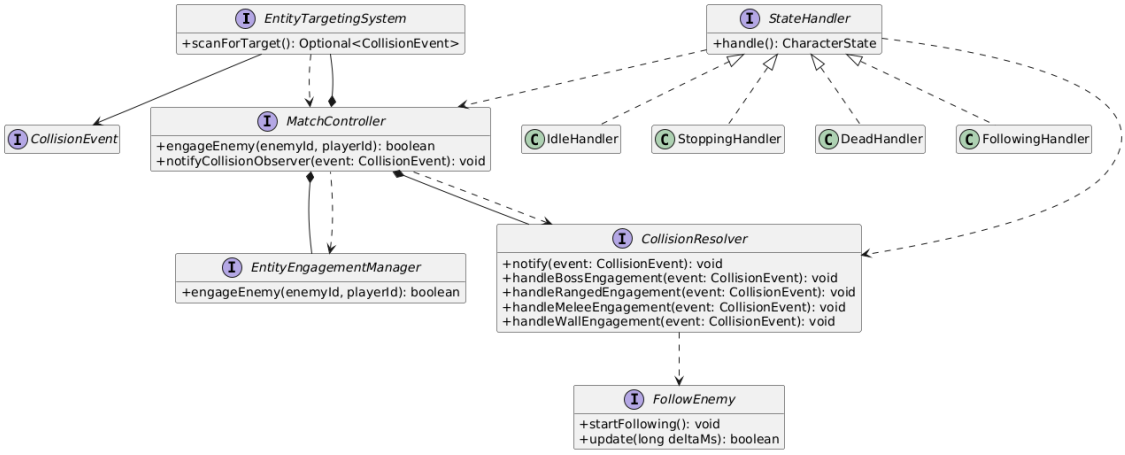

Figura 6: Schema UML delle classi e interfacce coinvolte nel combattimento

Gestione Elementi Passivi della partita - Magie

- **Problema**: Le magie sono state modellate come entità indipendenti e distinte dai personaggi. Non è stata adottata una struttura basata su entità generiche da cui derivare sia i personaggi che le magie; al contrario, le magie sono state progettate separatamente. Proprio per questa ragione, richiedono un sistema di gestione dedicato e distinto, che ne cura specificamente il funzionamento e le dinamiche all'interno del gioco.
- **Soluzione**: La gestione delle magie parte dall'input dell'utente, rilevato dalle interfacce *MatchPanel* e *Deckzone*, che notificano al *MatchController* la selezione di una carta e il click sulla mappa. Il *MatchController* inoltra questi dati allo *SpellManager*, che prepara l'incantesimo e ne memorizza il bersaglio. L'esecuzione avviene asincronamente nel ciclo di *update* dello *SpellManager*, mentre il MatchController, tramite la *MatchView*, mostra l'animazione. Contemporaneamente,

lo *SpellManager* applica l'effetto logico, identifica il target dallo *SpellEffect* e ne modifica lo stato.

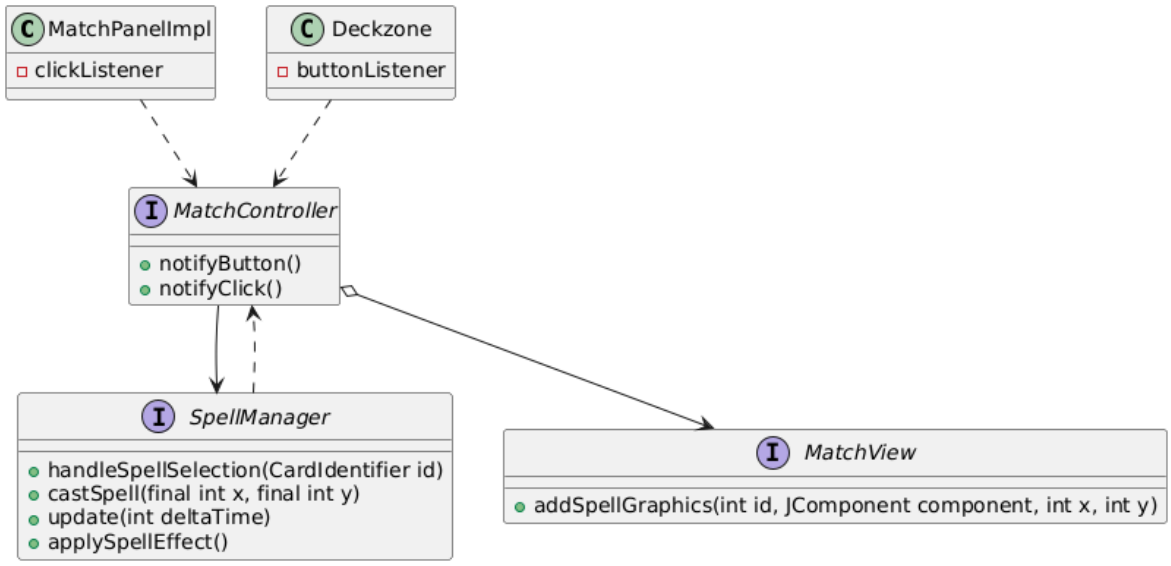

Figura 7: Schema UML delle classi e interfacce coinvolte nella gestione delle magie

Gestione del Main Loop

- **Problema**: Le entità di gioco devono essere periodicamente aggiornate per rendere effettive le modifiche al loro stato, come il movimento, le azioni e le interazioni con l'ambiente.
- **Soluzione**: Il processo di gioco inizia su input della *MainView*, che delega l'avvio al *MatchStartObserver*. Questo si occupa di inizializzare il *MatchController* con la logica del match e di lanciare il *GameLoop*. Una volta avviato, il GameLoop entra in un ciclo costante. A ogni iterazione, interroga il *MainController* per ottenere l'istanza attiva del MatchController e subito dopo ne chiama il metodo *update()*. Il ciclo si interrompe quando il *MatchController* determina la fine del gioco. A quel punto, notifica l'evento al *MatchStartObserver*, che procede a fermare il thread del GameLoop, chiudendo la partita.

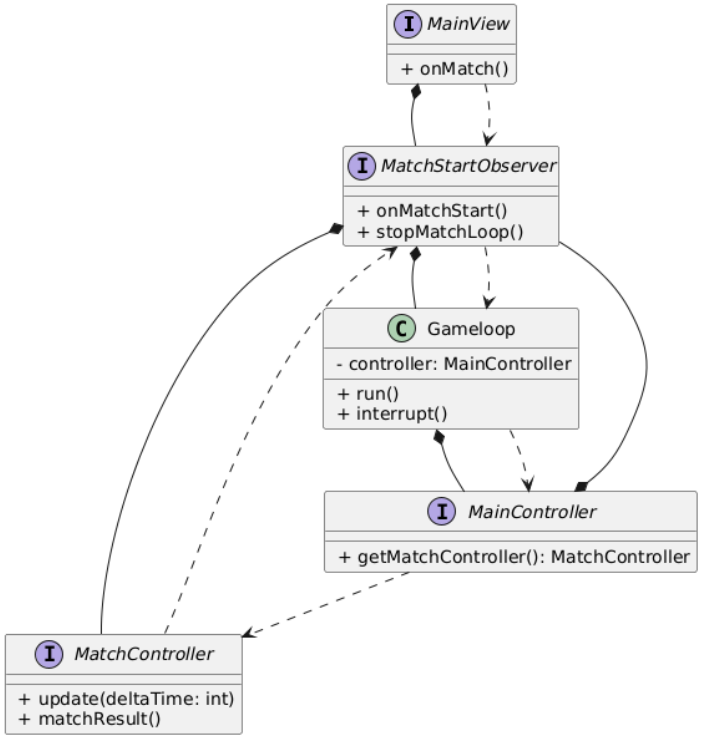

Figura 8: Schema UML delle classi e interfacce coinvolte nella gestione del GameLoop

2.2.2 Marco Battistini

Creazione delle entità di gioco coinvolte

- **Problema**: il primo problema da risolvere riguardante la mia sezione è stato quello di trovare una struttura di interfacce e classi per poter definire tutte le tipologie di personaggi e le magie coinvolte durante il gioco.
- **Soluzione**: Si è pensato quindi di partire da due interfacce principali: PlayableCard, ossia una carta che può essere giocata durante il match e GenericCharacter, un personaggio generico che contiene informazioni comuni a tutti i personaggi. Da queste due interfacce di partenza si sono poi costituite tutte le altre: l'interfaccia Spell per le magie e le interfacce per i personaggi specifici, che estendono quella del personaggio generico. Un caso particolare che ha richiesto più attenzione degli altri è la tipologia del personaggio definito "giocabile" (PlayableCharacter) che estende sia l'interfaccia GenericCharacter ma anche PlayableCard, poiché, salvo possibili espansioni future, è l'unica tipologia di personaggio che il giocatore può fisicamente far scendere il campo durante la partita, giocando la carta corrispondente. Date le diverse tipologie di carte/personaggi realizzati e ammettendo l'ipotesi di ulteriori aggiunte con possibili espansioni future, è stata realizzata un'enumerazione denominata CardType in modo da poter facilmente risalire al nome delle categorie di carte presenti nell'applicazione senza dover ricorrere all'utilizzo di stringhe.

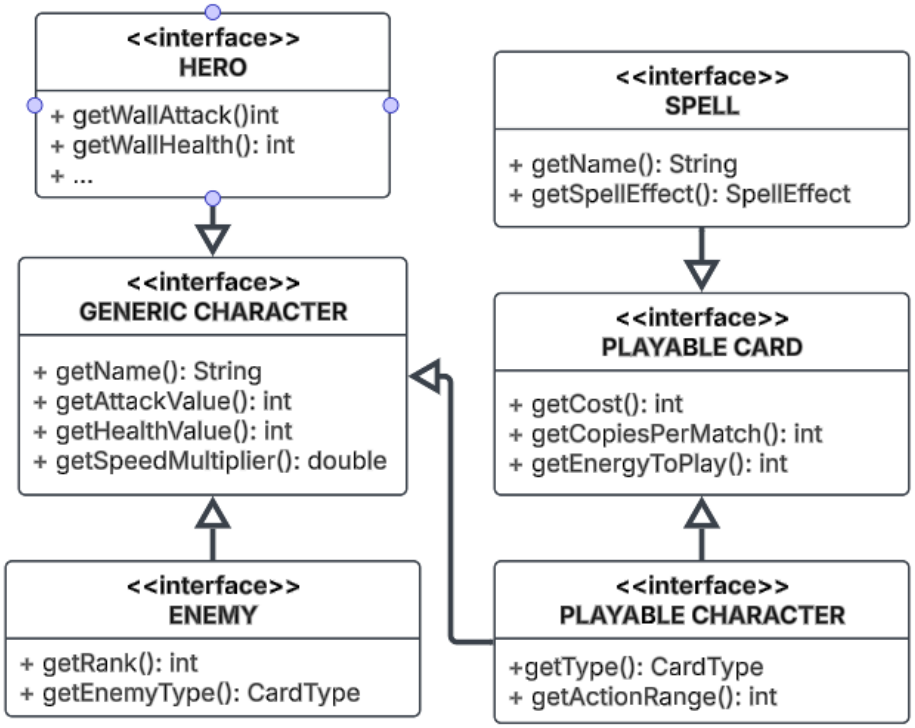

Figura 9: Interfacce che modellano i personaggi

- **Problema**: definite inizialmente le interfacce e le implementazioni dei personaggi è stato necessario risolvere subito un altro problema. Data la "doppia" natura del gioco (statica in fase di controllo della propria collezione di carte e dinamica in fase di partita) si è pensato che le interfacce sopra definite e le loro implementazioni, costituite prevalentemente da getters, non fossero adatte alla dinamicità della partita, dove le statistiche dei vari personaggi sono soggette a numerosi cambiamenti, per esempio in seguito a un colpo subito.
- **Soluzione**: ideazione di un'ulteriore interfaccia che racchiude tutti i possibili personaggi: InGameCharacter, ossia come il nome suggerisce un personaggio che si trova schierato sul campo durante la partita. Gli oggetti che derivano dall'inizializzazione della classe che implementa questa interfaccia saranno creati in fase di partita dal controller relativo a un singolo personaggio (vedere sezione coesione tra le parti e comunicazione con l'esterno). In questo modo si può di fatto aggirare la staticità delle classi delineate precedentemente, le quali sono utili solamente durante la consultazione della collezione di carte. Con questa tecnica ci sono molteplici vantaggi ma anche qualche svantaggio da tenere a mente. Come detto precedentemente, rappresentare un personaggio specifico ed avere dei metodi che permettono di modificare le statistiche dello stesso senza modificare i dati originali, poter inizializzare più copie dello stesso personaggio potendoli comunque distinguere con un codice numerico. A questo si aggiunge anche la possibilità di svincolare e separare meglio gli oggetti protagonisti della fase di gioco a quella più statica della collezione. Lo svantaggio principale che questa soluzione comporta è la ridondanza, poiché di fatto si introducono due volte gli stessi dati per le carte che figurano nel mazzo del giocatore durante la partita e che sono ovviamente presenti anche nella collezione di carte del gioco.

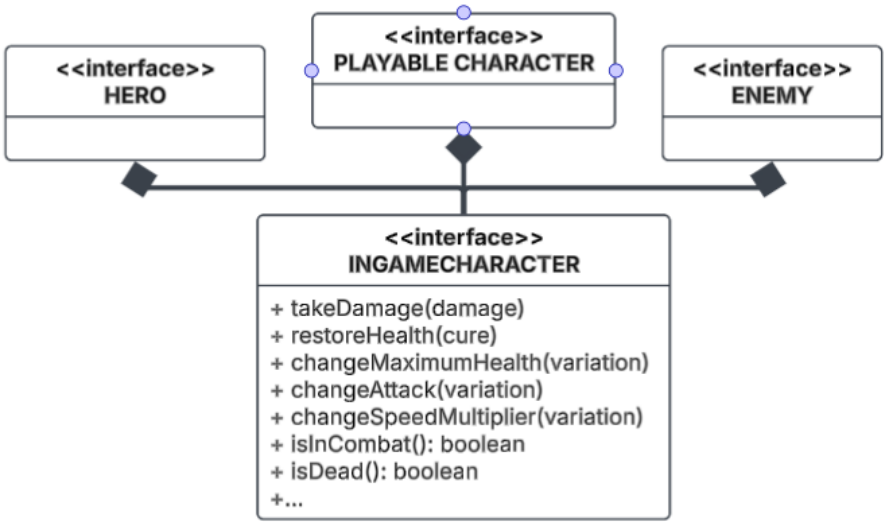

Figura 10: Interfaccia InGameCharacter, che si basa sui personaggi precedenti

- **Problema**: Date le numerose tipologie di interfacce presenti, si è subito pensato all'implementazione di un sistema efficace che potesse disaccoppiare la dichiarazione di un nuova entità qualsiasi essa sia dalla implementazione effettiva.

- **Soluzione**: si è adottato il pattern Factory, creando una specifica utility class Factory per ogni tipologia di personaggio e magia. Questo non solo permette di poter utilizzare queste interfacce in maniera completamente indipendente dalla loro implementazione effettiva, ma il codice nel suo complesso risulta decisamente più manutenibile e facilmente estendibile in previsione di possibili ampliamenti futuri, in quanto basterebbe cambiare semplicemente l'oggetto restituito nella Factory, una volta che si è creata una nuova implementazione per la tipologia di personaggio. Lo schema UML seguente mostra l'esempio relativo al Playable Character, ma la stessa struttura è stata adottata anche per le altre interfacce analizzate.

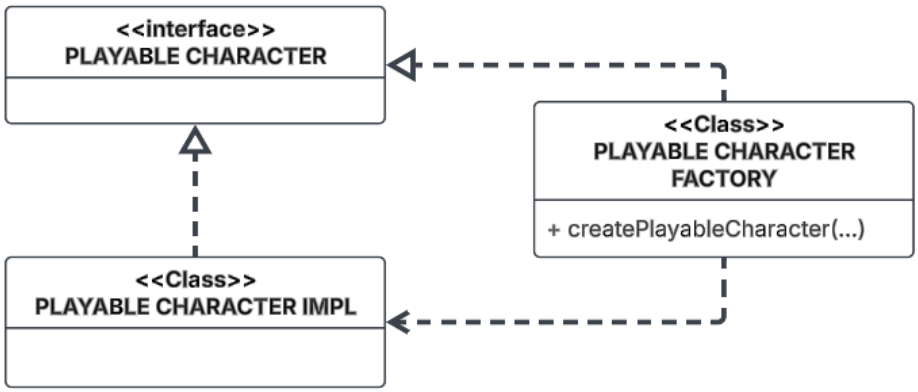

Figura 11: Pattern factory applicato all'interfaccia Playable Character

Creazione della componente grafica relativa ad un singolo personaggio/magia

- **Problema**: Una volta definito il modello dei personaggi si è passati a studiare un metodo efficace per poter visualizzare le animazioni dei personaggi e delle magie, privilegiando la suddivisione dei compiti tra diverse classi, l'estendibilità e il riuso del codice. Aspetto cruciale è quello di poter permettere a classi esterne all'ambito grafico di decidere quando cambiare l'immagine visualizzata del personaggio, in modo da generare l'animazione con un numero ben preciso di fps.
- **Soluzione**: la classe che funge da perno della componente grafica dei personaggi/magie è sicuramente la GUI (differenziata a seconda del personaggio o della magia), la quale si occupa sostanzialmente di creare il componente grafico vero e proprio su cui dovrà apparire l'entità considerata (una versione customizzata di JPanel che estende l'interfaccia AnimationPanel). Anche la GUI presenta un'interfaccia di partenza per il personaggio generico e altre interfacce che estendono la prima e rappresentano personaggi specifici. Data la varietà di animazioni implementate (per esempio stop, corsa, attacco...) è stata ideata un'enumerazione (Keyword) che le racchiude tutte e semplifica l'utilizzo delle stesse, evitando il ricorso alle stringhe che può facilmente portare a bug dovuti a imprecisioni e/o errori di battitura.

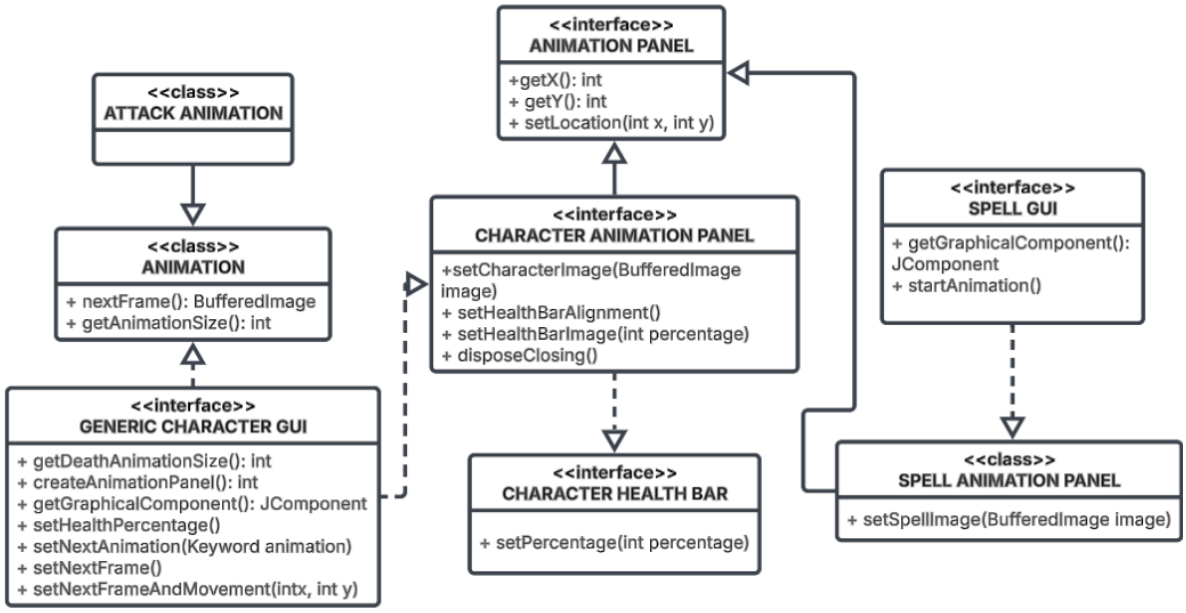

Figura 12: Componente grafica di personaggi e magie

Coesione tra le parti e comunicazione con l'esterno

- **Problema**: Definita la macrostruttura della componente grafica dei personaggi e delle magie si è passati a come poter modellare un sistema di controller che permettesse una buona coesione tra il modello, la grafica e di accettare input oppure comunicare con il controller che gestisce il match, ossia la fase di combattimento tra i personaggi.
- **Soluzione**: Analogamente alla parte di modello si è realizzata una interfaccia madre (GenericCharacterController) che modella tutti gli aspetti comuni ai controller dei personaggi: creare l'oggetto InGameCharacter relativo allo specifico personaggio assegnato e modificare le sue statistiche, inizializzare la componente grafica ed esporre dei metodi che permettano dall'esterno di iniziare nuove animazioni. Di rilievo anche l'implementazione dell'interfaccia CharacterHitObserver a carico dei controller dei personaggi. Questa interfaccia permette infatti di agganciare i personaggi come "avversari" e creare uno scontro tra due o più di questi ultimi, attraverso per esempio il metodo "takeHit()" che permette ad un personaggio di inviare un danno al suo avversario. Questa interfaccia è di fatto un'applicazione del pattern Observer. In questo caso particolare ogni personaggio coinvolto in uno scontro rappresenta allo stesso tempo un oggetto Observer e Observable, poiché gli deve essere notificato quando subisce un danno, ma deve notificare all'avversario quando questo lo deve ricevere a sua volta. Il controller dei personaggi permette anche di cedere all'esterno la versione "protetta" dell'oggetto Animation Panel sotto forma di JComponent, in modo che possa essere aggiunto alla mappa.

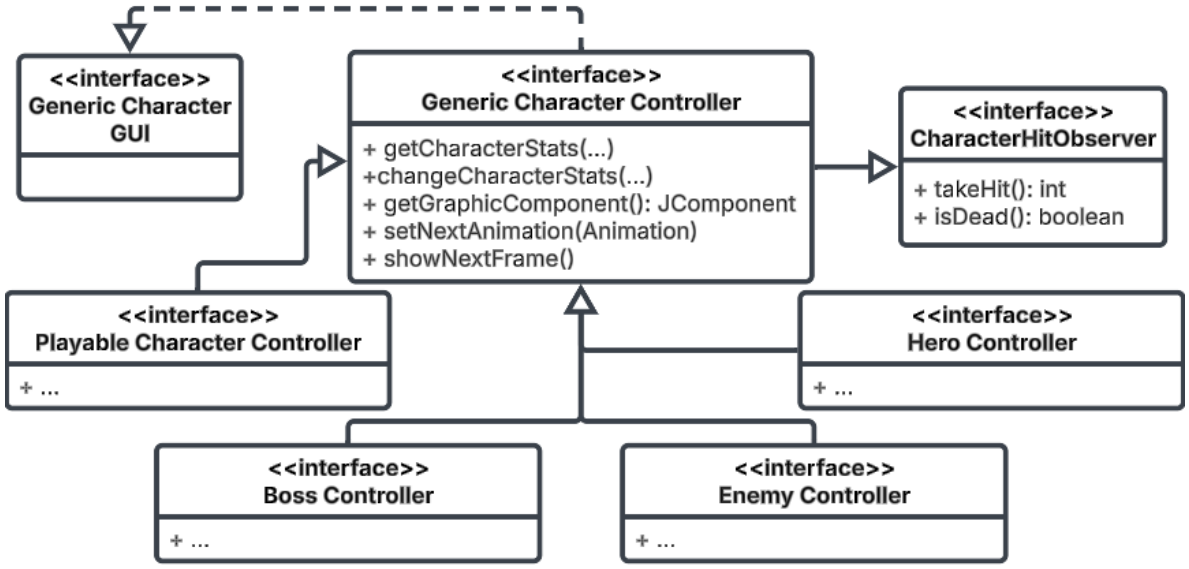

Figura 13: Struttura di controller dedicati ai personaggi

Personaggi situati nel negozio

- **Problema**: La partita vera e propria del giocatore contro i nemici, che avviene nella mappa sopra delineata, deve essere alternata da una fase di acquisto e gestione che avviene in un "negozio", dove il giocatore può acquistare nuove carte da poter essere subito inserite nel mazzo, in modo da poterle già adoperare e testare nella partita successiva.
- **Soluzione**: Per simulare un vero e proprio negozio, si è pensato di introdurre alcuni personaggi particolari: I Traders (commercianti). Essi sono caratterizzati da alcuni tratti in comune con le altre tipologie di personaggi, per esempio dispongono di diverse animazioni che vengono utilizzate a seconda del contesto. Tuttavia, affinché la sezione del negozio sia completamente indipendente dal controller che gestisce il match (e di conseguenza scandisce anche il ritmo di animazione dei personaggi), si è ideato una classe denominata TraderPanel che estende sempre JPanel e gestisce autonomamente le animazioni del Trader ad esso associato. Per rendere possibile la sequenza di animazioni senza fare affidamento sul motore centrale di gioco, si è realizzata la classe CustomLock, una classe che fornisce due metodi efficaci che permettono ai diversi thread responsabili delle animazioni di operare in mutua esclusione.

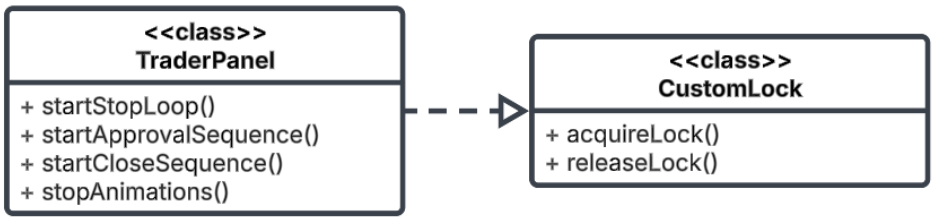

Figura 14: schema Trader e Custom Lock

2.2.3 Riccardo Carta

Lettura e salvataggio di informazioni su file

- **Problema**: Il sistema ha la necessità di leggere e scrivere su file diversi oggetti del dominio (come Account, Enemy, Hero, ecc.), ciascuno con una propria struttura dati.
- **Soluzione**: La maggior parte dei file devono semplicemente essere letti e l'unico a dover essere anche modificato è il file dell'utente. La lettura effettiva viene fatta da controller usando oggetti di tipo FileHandler e ReadOnlyFileHandler in modo da separare la logica di lettura/scrittura dall'utilizzo vero e proprio dei dati. Si utilizza quindi una classe astratta BaseFileHandlerI che fornisce l'implementazione di base per la lettura da file sfruttando un oggetto di tipo Parser (Template). Le implementazioni delle interfacce FileHandler e ReadOnlyFileHandler usano poi questa classe per quanto concerne le operazioni di lettura. L'interfaccia Parser ha poi diverse implementazioni in base al tipo di dato da leggere (Startegy). FileHandler viene utilizzato per la scrittura (e lettura) dei dati dell'account avvalendosi di un oggetto Serializer, che analogamente al Parser può avere diverse implementazioni (in questo caso ne viene fatta solo una).

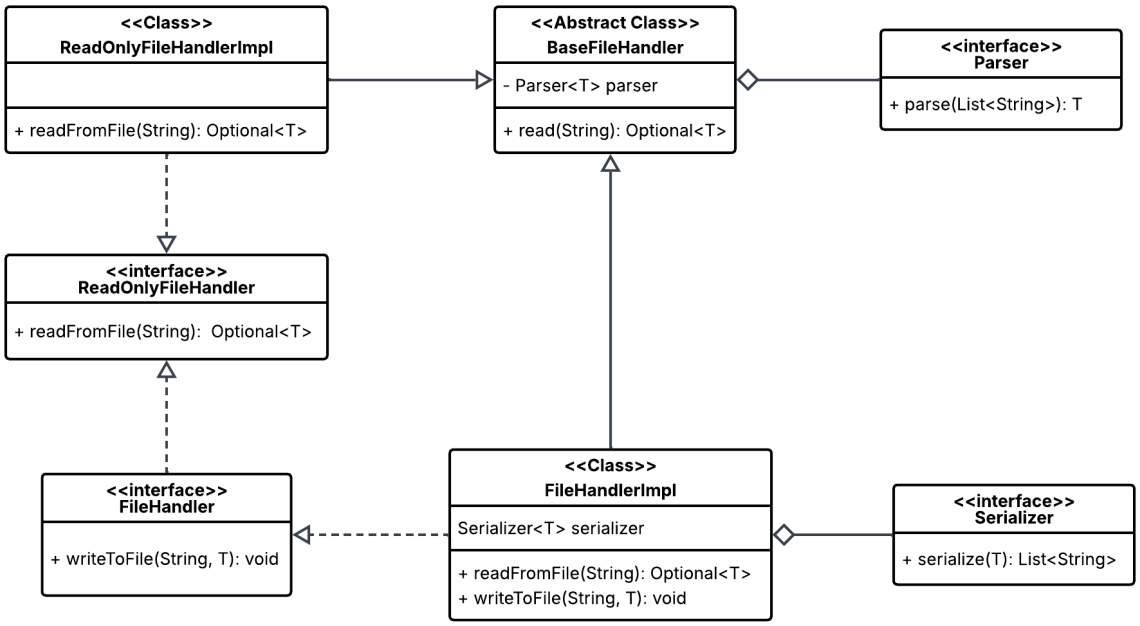

Figura 15: Schema UML delle classi e interfacce coinvolte per la gestione dei file

Gestione dell'utenza

- **Problema**: Caricare i dati dell'utente dopo il login
- **Soluzione**: La view di login sfrutta dapprima un UsernameController per verificare che il nome inserito sia corretto. Dopo aver avuto la conferma dal controller

che il nome inserito è corretto e dall'utente che intende procedere, richiama il metodo goOverLogin del MainController dell'applicazione che inizializza un nuovo AccountController il quale, partendo dallo username inserito dall'utente va a creare un nuovo Account con relativo file di salvataggio dei dati se lo username non era mai stato utilizzato, altrimenti va semplicemente a caricare i dati presenti nel file dell'account.

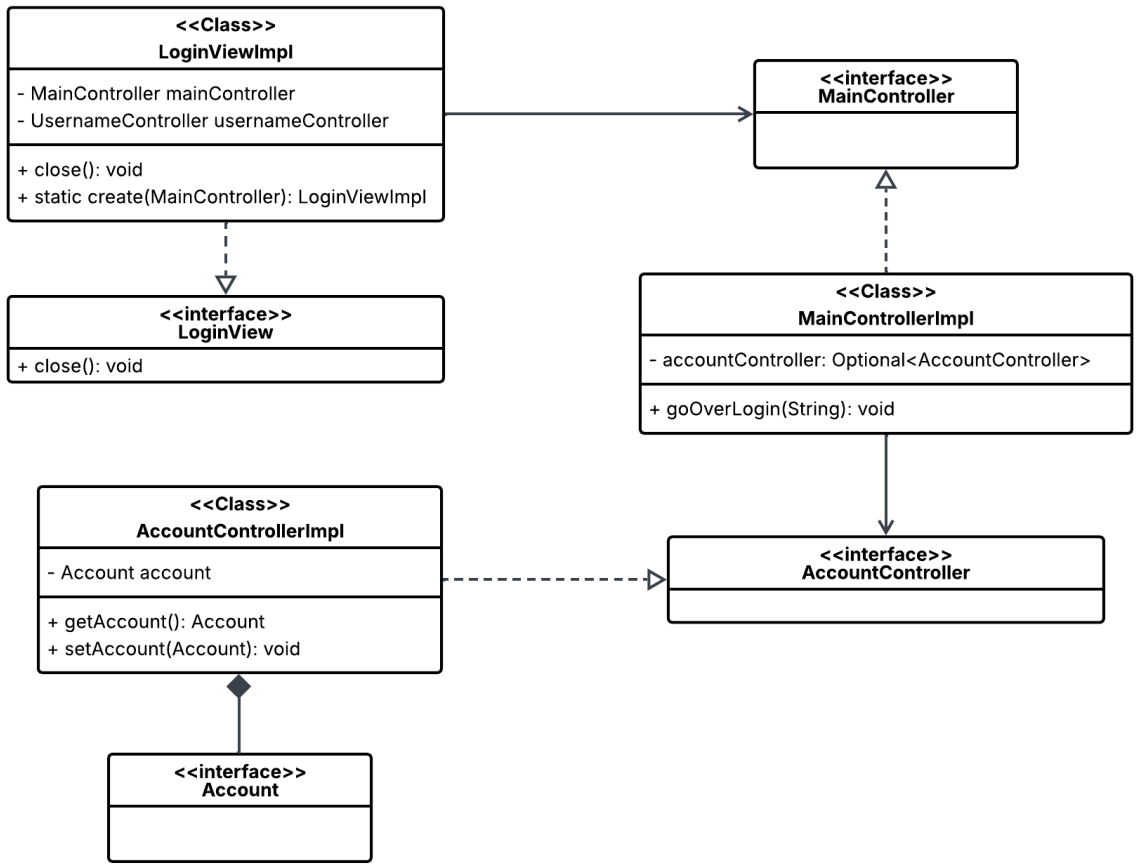

Figura 16: Schema UML delle classi e interfacce coinvolte per la gestione dell'utenza

Gestione del mazzo dell'utente fuori dalla partita

- **Problema**: L'utente deve essere in grado di creare e gestire il proprio mazzo in maniera semplice e aggiornata rispetto alla sua collezione
- **Soluzione**: Per poter modificare il proprio mazzo, l'utente interagisce semplicemente con una DeckView andando a visualizzare le carte presenti attualmente nella sua collezione e potendole selezionare per aggiungerle/rimuoverle dal mazzo. La DeckView si interfaccia a sua volta con un DeckController creato dal MainController subito dopo aver superato la fase di login e che serve a gestire in maniera coordinata il mazzo dell'utente. Il DeckController utilizzato dalla DeckView viene aggiornato con la creazione di una nuova sua istanza dalla MainViewImpl che

contiene la DeckView. Il DeckController prende come parametro nel costruttore la collezione dell'utente e contiene dei getter per poter passare ad altre classi le informazioni del mazzo, oltre a due metodi removeCard e addCard che permettono rispettivamente di rimuovere o aggiungere carte dal mazzo. Il mazzo vero e proprio è un campo privato di tipo Deck presente nel DeckController creato adottando il pattern Static Factory; infatti, viene usata una StandardDeckFactory nell'ottica di rendere il controlller indipendente da possibili future implementazioni differenti del Deck. Questo approccio sposta la conoscenza della classe concreta (DeckImpl) fuori dal controller, che rimane dipendente solo dall'interfaccia Deck. I limiti da rispettare nella creazione del mazzo imposti dall'eroe sono infine gestiti internamente dall'oggetto Deck nella sua implementazione DeckImpl. Il Set di CardIdentifier rappresentante il Deck viene passato alla MatchView per poter arrivare alla DeckZone dove viene gestito per quanto concerne il suo utilizzo in partita tramite l'InGameDeckController. In questo modo si riesce a tenere separati ma aggiornati tra di loro, la gestione del mazzo fuori dalla partita, che concerne principalmente la selezione delle carte da inserire, dalla gestione del marzo dentro la partita che invece concerne ad esempio la gestione delle carte presenti nel mazzo come copie da poter usare o dell'alternanza delle carte disponibili al momento.

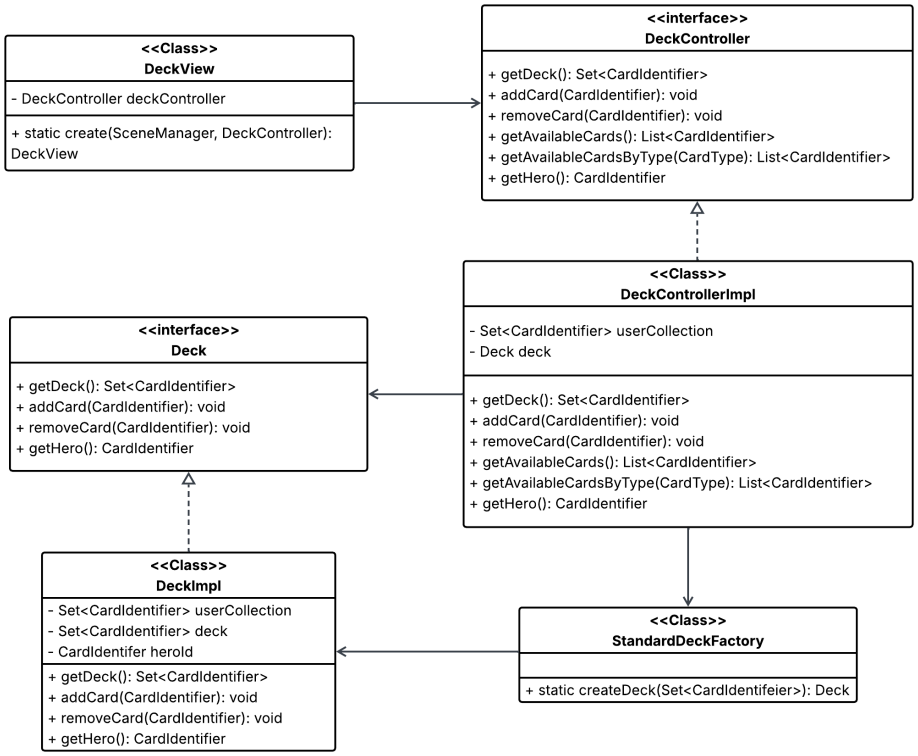

Figura 17: Schema UML delle classi e interfacce coinvolte per la gestione del mazzo dell'utente

Gestione della view

- **Problema**: Le view devono essere visibili una alla volta e poter essere scambiate tra di loro
- **Soluzione**: Per poter rappresentare e gestire le view del gioco in maniera centralizzata si è fatto uso del layout CardLayout di Java Awt. Tale layout ha permesso di rendere le view tra loro indipendenti e fare in modo che fosse l'implementazione di MainView a creare un'istanza per ciascuna di esse e ad occuparsi internamente di mostrarle una alla volta scambiandole tramite il metodo changePanel che prende in input il nome del pannello chiamante e quello del pannello di destinazione. Il metodo changePanel di MainViewImpl (che al suo interno chiama il metodo show di CardLayout) viene chiamato unicamente da uno SceneManager. Tutte le classi che vogliono invocare uno scambio di pannelli interagiscono quindi con lo SceneManager e non direttamente con la MainView. Dato che le view principali (es. MenuView, DeckView, MatchView...) hanno metodi in comune, estendono tutte una classe astratta AbstractScene che contiene l'implementazione di base di questi metodi comuni. A sua volta quest'ultima implementa l'interfaccia Scene, che descrive oggetti che abbiano i due metodi per ottenere il nome del pannello ed il pannello rappresentato, fondamentali per il CardLayout.

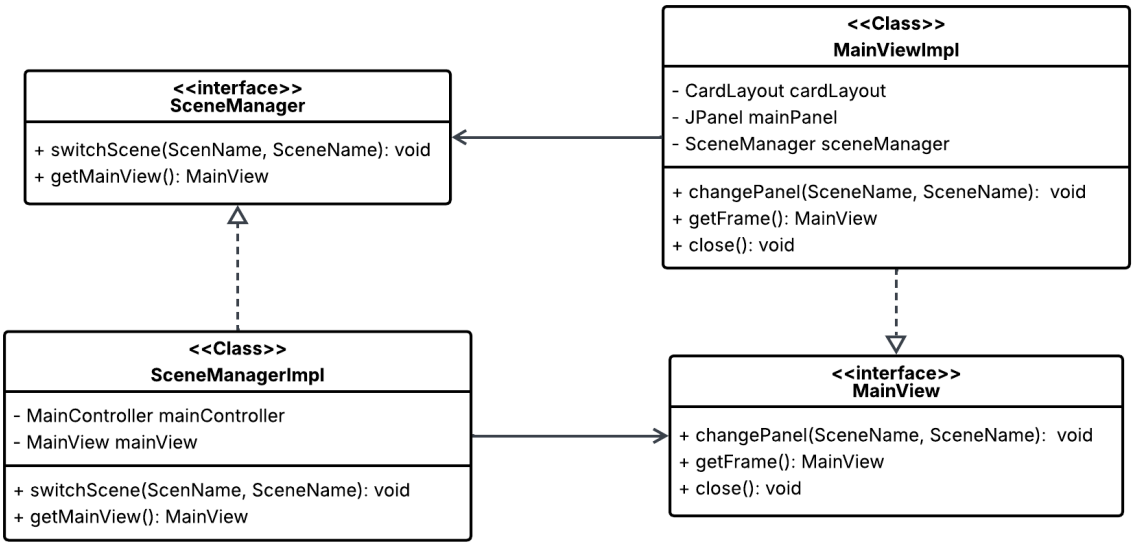

Figura 18: Schema UML delle classi e interfacce coinvolte per la gestione delle view principali

Capitolo 3

Sviluppo

3.1 Testing Automatizzato

Per quanto riguarda il testing automatizzato si è utilizzata la libreria JUnit e si sono realizzati test su alcune classi di Model e Controller che abbiamo ritenuto importante testare separatamente dal resto dell'applicazione, come per esempio i personaggi, le magie, l'account del giocatore e la sua collezione. Ciò è stato fatto per garantire che i principali elementi di gioco funzionassero indipendentemente dall'architettura circostante. La sezione di Controller più generica è stata messa alla prova in fase di sviluppo delle nostre parti e successivamente durante il collaudo dell'applicazione. Per quanto riguarda invece la componente prettamente grafica abbiamo preferito controllarla in maniera molto approfondita anch'essa nella fase finale dell'elaborazione del progetto, in quanto purtroppo non siamo riusciti ad approfondire dinamiche di testing automatizzato che ci avrebbero permesso di verificare il corretto funzionamento della stessa in un momento precedente. In linea generale, gli aspetti su cui ci siamo maggiormente soffermati sono i seguenti:

- **Personaggi**: sono stati effettuati molteplici test su tutte le diverse tipologie sopra presentate, ponendo particolare attenzione all'effettiva correttezza dei metodi che vanno a cambiare le statistiche dei personaggi in partita
- **Magie**: anche le magie sono state messe alla prova per verificare il corretto funzionamento dei getters del modello
- **Account**: è stato verificato la corretta gestione delle risorse e delle statistiche di un account
- **Deck**: viene testato il corretto funzionamento di aggiunta/rimozione delle carte
- **Hitbox**: viene testata la creazione della hitbox, l'aggiornamento dei suoi attributi e il calcolo del suo centro geometrico. Inoltre, viene testata in dettaglio la logica di collisione, includendo i casi di contatto, non contatto e il comportamento ai limiti con un'area di buffer
- **Radius**: viene testato il corretto funzionamento del raggio assicurano che vengano individuati soltanto i bersagli posti di fronte e dentro una specifica distanza. Inoltre, viene controllata la capacità di identificare il nemico più vicino quando più bersagli validi sono presenti nell'area.
- **Collisione**:viene testata la collisione fra personaggi assicurano che, in seguito a un evento di collisione, il resolver registri correttamente gli "ingaggi" tra le entità, sia in scontri 1-vs-1 che in combattimenti di gruppo. Inoltre, viene verificata la capacità di terminare e resettare questi ingaggi in modo selettivo o completo.

3.2 Note di Sviluppo

3.2.1 Giulia Albertini

- **Gestione dei Thread** al permalink `https://github.com/GiuliaAlbertini/OOP24-LastCrown/blob/7e9b3abb88382ba05861530668fc5afb161fc2d0/src/main/java/it/unibo/oop/lastcrown/controller/collision/impl/Gameloop.java`
- **Lambda Expressions e Stream** al permalink `https://github.com/GiuliaAlbertini/OOP24-LastCrown/blob/7e9b3abb88382ba05861530668fc5afb161fc2d0/src/main/java/it/unibo/oop/lastcrown/controller/collision/impl/EntityStateManagerImpl.java`
- **Optional** al permalink `https://github.com/GiuliaAlbertini/OOP24-LastCrown/blob/7e9b3abb88382ba05861530668fc5afb161fc2d0/src/main/java/it/unibo/oop/lastcrown/controller/collision/impl/EntityTargetingSystemImpl.java`
- **Record** al permalink `https://github.com/GiuliaAlbertini/OOP24-LastCrown/blob/7e9b3abb88382ba05861530668fc5afb161fc2d0/src/main/java/it/unibo/oop/lastcrown/controller/collision/impl/EntityEngagementManagerImpl.java`
- **Function** al permalink `https://github.com/GiuliaAlbertini/OOP24-LastCrown/blob/7e9b3abb88382ba05861530668fc5afb161fc2d0/src/main/java/it/unibo/oop/lastcrown/controller/collision/impl/SpellManagerImpl.java`

  Codice preso da altri

  - Per la realizzazione della classe Pair al link: `https://bitbucket.org/mviroli/oop2024esami/src/master/a01b/e1/Pair.java`
  - Per la realizzazione delle classi Point2D e Vect2D:
    - `https://github.com/aricci303/game-as-a-lab/blob/91336ae1dc0c58594af0b733330518c3d335b960/Game-As-A-Lab-Step-3-collisions/src/rollball/common/P2d.java`
    - `https://github.com/aricci303/game-as-a-lab/blob/91336ae1dc0c58594af0b733330518c3d335b960/Game-As-A-Lab-Step-3-collisions/src/rollball/common/V2d.java`

3.2.2 Marco Battistini

- Utilizzo di lambda function. Sono state utilizzate in diversi punti, un esempio è https://github.com/GiuliaAlbertini/OOP24-LastCrown/blob/a1162d27580a65320d82bf950cda63dba53215d7/src/main/java/it/unibo/oop/lastcrown/view/map/MatchViewImpl.java#L102C30-L105C10
- Utilizzo di Optional nell'interfaccia ContainerObserver e conseguentemente nella classe ShopViewImpl https://github.com/GiuliaAlbertini/OOP24-LastCrown/blob/a1162d27580a65320d82bf950cda63dba53215d7/src/main/java/it/unibo/oop/lastcrown/view/shop/ShopViewImpl.java#L45

3.2.3 Riccardo Carta

Di seguito sono elencati singoli esempi di parti di codice in cui vengono usate funzionalità avanzate. All'interno del codice tuttavia appaiano più situazioni in cui vengono usati.

- **Stream** al permalink https://github.com/GiuliaAlbertini/OOP24-LastCrown/blob/a1162d27580a65320d82bf950cda63dba53215d7/src/main/java/it/unibo/oop/lastcrown/model/user/impl/CompleteCollectionImpl.java#L108-L123
- **Espressioni Lambda**: al permalink https://github.com/GiuliaAlbertini/OOP24-LastCrown/blob/a1162d27580a65320d82bf950cda63dba53215d7/src/main/java/it/unibo/oop/lastcrown/controller/user/impl/AccountControllerImpl.java#L52C24-L56C14
- **Input e Output Stream**: ai permalink https://github.com/GiuliaAlbertini/OOP24-LastCrown/blob/a1162d27580a65320d82bf950cda63dba53215d7/src/main/java/it/unibo/oop/lastcrown/model/file_handling/impl/BaseFileHandler.java#L63 https://github.com/GiuliaAlbertini/OOP24-LastCrown/blob/a1162d27580a65320d82bf950cda63dba53215d7/src/main/java/it/unibo/oop/lastcrown/model/file_handling/impl/FileHandlerImpl.java#L56C22-L56C40
- **Optional**: al permalink https://github.com/GiuliaAlbertini/OOP24-LastCrown/blob/a1162d27580a65320d82bf950cda63dba53215d7/src/main/java/it/unibo/oop/lastcrown/controller/app_managing/impl/MainControllerImpl.java#L66C34-L67C48

Capitolo 4

Commenti Finali

4.1 Autovalutazione e lavori futuri

4.1.1 Giulia Albertini

Personalmente, questo è stato il primo progetto di tali dimensioni e complessità a cui ho partecipato. All'inizio mi sono trovata un po' disorientata, soprattutto nelle fasi iniziali, ma con il progredire del lavoro ho acquisito maggiore consapevolezza degli strumenti necessari per portare a termine il progetto. Un momento critico è stato l'uscita di un componente del gruppo, che ha comportato un prolungamento dei tempi di consegna e diverse difficoltà. Il lavoro prodotto da questo componente non era utilizzabile, sia dal punto di vista della correttezza del codice sia della funzionalità, e di conseguenza abbiamo dovuto rivedere e riprogettare alcune funzionalità, riorganizzando le responsabilità interne al team. Questa situazione non è stata l'unico ostacolo: a livello personale ho trovato complesso lavorare all'interno del gruppo. Durante l'integrazione delle diverse parti, ho dovuto adattare il mio codice a componenti sviluppati senza una comunicazione efficace con la mia parte, e considerando l'approssimarsi della scadenza, una riscrittura completa non era più fattibile. Inoltre, a mio parere, la suddivisione delle responsabilità è stata gestita in modo troppo rigido. Pur riconoscendo l'importanza che ogni membro abbia la propria area di competenza, credo che per alcune componenti fosse necessaria una maggiore collaborazione, che invece è stata in qualche modo limitata, anche a scapito della complessità finale del codice. Dal punto di vista personale, ho trovato difficile gestire la pressione costante e la supervisione su aspetti di cui ero già consapevole. Le numerose comunicazioni hanno contribuito ad aumentare il senso di stress. Nonostante tutte queste difficoltà, sono soddisfatta del risultato finale, anche se riconosco di non aver espresso al massimo le mie capacità. All'inizio infatti ho affrontato il progetto con idee non del tutto chiare, che solo nelle fasi finali si sono concretizzate. Tuttavia, a causa dei vincoli legati agli aspetti sopra menzionati, alcune delle soluzioni adottate non sono risultate ottimali e avrebbero potuto essere migliorate con più tempo e condizioni diverse. Questa esperienza si è comunque rivelata molto utile per comprendere meglio come affrontare un progetto di questa portata e come relazionarmi efficacemente con gli altri membri di un team. Inoltre, lo sviluppo del progetto mi ha permesso di approfondire notevolmente la materia, apprezzandola da diversi punti di vista.

4.1.2 Marco Battistini

Questo è stato per me il primo progetto che richiedesse un lavoro di gruppo di una durata decisamente significativa, mentre nella mia carriera universitaria precedente avevo sempre svolto progetti che richiedessero meno tempo e che potessero svolgersi singolarmente. Tuttavia, è stato molto stimolante accordarsi con i compagni fin dalle primissime fasi di ideazione della tipologia di applicazione per cercare una soluzione insieme. Purtroppo, l'uscita dal gruppo di uno dei quattro membri originali ha causato diversi problemi all'effettivo termine dei lavori di sviluppo e revisione del software e ci

ha costretto a chiedere settimane di tempo in più rispetto alla data fissata inizialmente. Questo imprevisto, inoltre, ci ha costretti a rivedere parzialmente le funzionalità e le caratteristiche complessive dell'applicazione che ci figuravamo di realizzare, specialmente quelle riguardanti la mappa di gioco e il negozio, che ci siamo dovuti spartire per accelerare i tempi. Detto questo mi ritengo nel complesso soddisfatto di quanto svolto perché, almeno per quello che riguarda la mia sezione di lavoro originale, rispecchia nei risultati ciò che mi ero inizialmente prefissato. Ritengo che come applicazione la nostra sia decisamente aperta anche a future modifiche e ampliamenti, per esempio l'aumento del numero di carte disponibili o l'introduzione di miglioramenti ai personaggi che rimangono permanenti anche in sessioni di gioco successive.

4.1.3 Riccardo Carta

Alla fine della realizzazione di questo progetto mi ritrovo abbastanza soddisfatto. È stato il primo progetto che ho affrontato in gruppo e, specialmente nella fase iniziale, ho riscontrato alcune difficoltà nell'adeguarmi e nel comunicare efficacemente con i miei colleghi. Tuttavia, nonostante alcuni intoppi, come l'abbandono del gruppo da parte di uno dei partecipanti, ritengo che siamo riusciti a collaborare agevolmente. In particolare, riconosco di aver acquisito maggiore sicurezza nel valutare le mie capacità tecniche e di gestione, per supportare al meglio il resto del team.

D'altro canto, devo ammettere di non aver svolto il mio lavoro al massimo delle mie possibilità, soprattutto nella progettazione della mia parte. Ho iniziato a scrivere il codice con idee di base non ancora ben definite; questo mi ha portato, nel momento di ultimare la mia sezione, a scoprire che alcune classi o soluzioni su cui avevo lavorato nei mesi precedenti erano mal strutturate o implementate. Risolvere questi problemi avrebbe comportato la riscrittura di gran parte del mio lavoro, cosa che a quel punto era ormai troppo tardi per fare per intero.

L'esperienza mi è stata sicuramente utile, perché ora sono più consapevole di come approcciarmi ad un progetto importante e di come lavorare con gli altri. Inoltre, durante lo sviluppo ho potuto migliorare e consolidare le mie conoscenze in ambito Java e di progettazione.

Appendice A

Guida Utente

Inizialmente l'**utente** inserisce il proprio nome. Nel caso in cui sia già registrato, verranno ripristinate le statistiche di gioco a com'erano alla chiusura precedente dell'applicazione. Nel caso in cui venga inserito un nuovo nome verrà creato un nuovo nominativo per l'utente e salvate le sue statistiche. Ad un nuovo accesso vengono già fornite alcune monete per poter permettere qualche acquisto senza aver ancora sconfitto alcun nemico. All'accesso l'utente avrà a disposizione diverse opzioni, tra cui il pulsante Match che darà il via a una nuova sessione di gioco vera e propria, caratterizzata dall'alternanza tra il negozio e il combattimento con i nemici. Durante la sessione di **combattimento**, per **selezionare una carta** da giocare basta cliccare sopra l'icona corrispondente. Al passaggio del mouse sulle icone, verrà evidenziata la zona in cui bisogna giocare la carta scelta. Per schierarla in campo si esegue un ulteriore click sulla zona evidenziata precedentemente. A questo punto il personaggio/magia comincerà una routine precisa e ben definita e non sarà soggetto al controllo dell'utente. La corretta **strategia** da seguire è posizionare le **carte mischia** quanto più possibile in linea retta davanti ai nemici, in modo che possano ingaggiare un combattimento velocemente. Le **carte magia** vanno invece posizionate alla destra del muro difensivo ed applicano un effetto globale a tutti i personaggi della categoria indicata che sono attualmente presenti sul campo di battaglia. **N.B.**: se l'**energia** disponibile (che si ricarica progressivamente) non è sufficiente per giocare una carta, questa non verrà schierata in campo fino a che non si raggiunge la soglia corretta, nemmeno se il click avviene nella zona predefinita. **Ulteriori istruzioni** su come costruire correttamente il proprio mazzo, come procedere con gli acquisti e come giocare le proprie carte durante il combattimento vengono fornite all'interno dell'applicativo stesso, facilitando così ulteriormente la comprensione dei comandi principali.