

Clash Royale

Mattia Panni
Riccardo Fiorani
Simone Bollini
Alexandro Salvato

31 agosto 2022

Indice

1 Analisi	3
1.1 Requisiti	3
1.2 Analisi e modello del dominio	4
2 Design	6
2.1 Architettura	6
2.1.1 Interazione Model-Controller	6
2.1.2 Interazione Controller-View	7
2.2 Design dettagliato	8
2.2.1 Mattia Panni	8
2.2.2 Riccardo Fiorani	13
2.2.3 Simone Bollini	16
2.2.4 Alexandro Salvato	22
3 Sviluppo	25
3.1 Testing automatizzato	25
3.2 Metodologia di lavoro	26
3.2.1 Mattia Panni	26
3.2.2 Riccardo Fiorani	27
3.2.3 Simone Bollini	28
3.2.4 Alexandro Salvato	28
3.3 Note di sviluppo	28
3.3.1 Mattia Panni	28
3.3.2 Riccardo Fiorani	29
3.3.3 Simone Bollini	29
3.3.4 Alexandro Salvato	29
4 Commenti finali	30
4.1 Autovalutazione e lavori futuri	30
4.1.1 Mattia Panni	30
4.1.2 Riccardo Fiorani	31

4.1.3	Simone Bollini	31
4.1.4	Alexandro Salvato	31
4.2	Difficoltà incontrate e commenti per i docenti	32
4.2.1	Mattia Panni	32
4.2.2	Simone Bollini	32
5	Guida utente	33
5.1	Menu	33
5.2	Deck	34
5.3	statistics	34
5.4	Game	35
6	Esercitazioni di laboratorio	37
6.1	Mattia Panni	37
6.2	Simone Bollini	37

Capitolo 1

Analisi

L'obbiettivo del progetto è quello di sviluppare un software che emuli il gioco *Clash Royale*, noto videogame della categoria tower-defence. Ogni giocatore, munito di un deck, composto da quattro carte, dovrà affrontare degli avversari dei quali cercherà di distruggere una o più torri per vincere la partita. La vittoria si ottiene distruggendo per primo tutte e tre le torri avversarie oppure, al termine del tempo prestabilito, avendone abbattendo più di quante ne abbia distrutte l'avversario. Al termine della partita, se il giocatore ha vinto guadagnerà degli XP (che gli serviranno per avanzare di livello e potenziare le carte in suo possesso) e dei trofei, altrimenti gli verranno sottratti questi ultimi.

1.1 Requisiti

Requisiti funzionali

- Il gioco dovrà essere in grado di salvare le statistiche di un giocatore, quali
 1. Partite vinte
 2. Torri distrutte
 3. Record di trofeie farle persistere anche qualora si chiudesse e si riaprisse in un secondo momento.
- Durante la partita, trascinando una carta fra quelle a propria disposizione sul campo, non oltre la propria metà, sarà possibile schierarla in quel punto.

- Le truppe dovranno muoversi in completa autonomia verso la truppa nemica schierata più vicina o, qualora non ve ne fosse nessuna o fossero tutte più distanti, verso la torre nemica ancora non abbattuta più vicina.
- Le truppe schierate dovranno evitare tutti gli ostacoli presenti sul percorso.
- Il combattimento fra le truppe dovrà avvenire senza l’intermediazione del giocatore e rispettando i range di ingaggio di ciascuna di esse. La truppa che riesce ad uccidere prima l’altra rimarrà schierata nel campo, l’altra scomparirà.
- La rotazione delle carte all’interno del deck dovrà seguire una logica ben precisa. La prima carta schierata nel campo, e dunque rimossa dal deck, sarà anche la prima che ricomparirà al suo interno.
- Non sarà possibile schierare carte il cui costo, calcolato in termini di elixir, è maggiore dell’elixir a disposizione del giocatore in un dato istante.
- La partita termina allo scadere del timer di gioco, o appena uno dei due contendenti distrugge tutte e tre le torri avversarie.

Requisiti non funzionali

- Il calcolo del percorso di ogni truppa dev’essere fatto efficientemente.
- L’incremento, la diminuzione dell’elisir, l’avanzamento del timer di gioco, devono funzionare correttamente e concorrentemente.

1.2 Analisi e modello del dominio

In Clash Royale, ogni utente, controllato dal videogioco o da una persona reale indifferentemente¹, avrà due minuti di tempo per riuscire a distruggere più torri avversarie possibili. Per fare ciò avrà a disposizione un set di carte (truppe o edifici) collocabili solamente nella propria metà campo e non oltre. Si possono schierare solo carte il cui costo risulta minore o uguale all’elixir correntemente posseduto dall’utente in questione. Quest’ultimo partirà da 0 e si ricaricherà autonomamente se non vengono schierate entità, per arrivare ad un massimo di 10. Le entità, una volta collocate in campo, dovranno

¹Per questioni di tempistiche, la modalità online verrà implementata in seguito

muoversi in autonomia all'interno della mappa, in direzione dell'entità (torre o carta che sia) nemica più vicina, passando solamente per uno dei due ponti posti fra le due metà campo, separate da un torrente. Ogni entità potrà cominciare ad attaccarne un'altra, solo se quest'ultima si troverà a una distanza minore o uguale al range di attacco della prima. Le torri potranno difendersi attaccando a loro volta, con modalità del tutto analoghe a quelle delle carte già spiegate sopra, le entità che si avvicineranno ad essa. La torre centrale (quella del Re), si attiverà, potendo difendersi attivamente dagli attacchi, solamente se una delle due torri laterali (delle Regine) vengono distrutte. Al termine della partita verranno calcolati xp e trofei che spettano al vincitore e quelli da sottrarre a colui che è stato sconfitto. Di seguito viene riportato uno schema UML che rappresenta il dominio appena descritto.

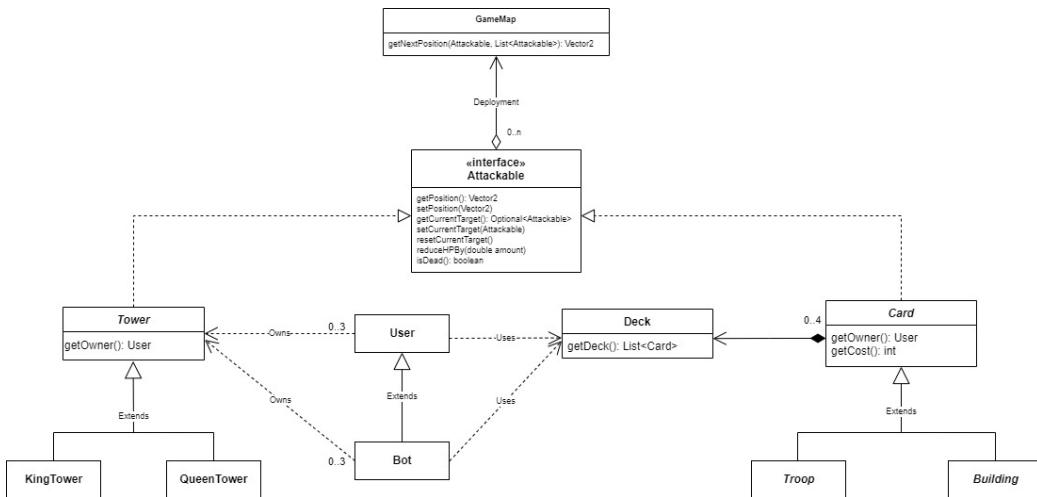


Figura 1.1: Schema UML del dominio applicativo.

Capitolo 2

Design

2.1 Architettura

L'applicativo realizzato segue le linee guida del pattern **MVC**, ossia è strutturato in macro componenti di Model, View e Control.

2.1.1 Interazione Model-Controller

All'interno del progetto, l'interazione fra model e controller è banalmente gestita inserendo un riferimento dei vari model all'interno dei singoli controller. Il punto d'ingresso al model è rappresentato dalla classe *GlobalData* che di fatto si occupa di raccogliere, istanziare e fornire a tutti i controller che ne abbiano bisogno, entità di centrale importanza come User, Bot, Deck, le quali, per questioni di facilità implementative, sono state progettate come dei *singleton*. In fase di caricamento della partita il *GameController* richiede a *GlobalData* gli oggetti di cui dispone per poter configurare in un secondo momento se stesso e la view. Poiché da specifiche, i dati relativi ad ogni User devono essere salvati e fatti persistere, *GlobalData* si appoggia ad un controller *SavingsController* il cui scopo è quello di istanziare qualora non esistesse, o caricare da file in caso contrario, le entità User e Deck. È riportato di seguito un UML relativo all'interazione fra model e controller.

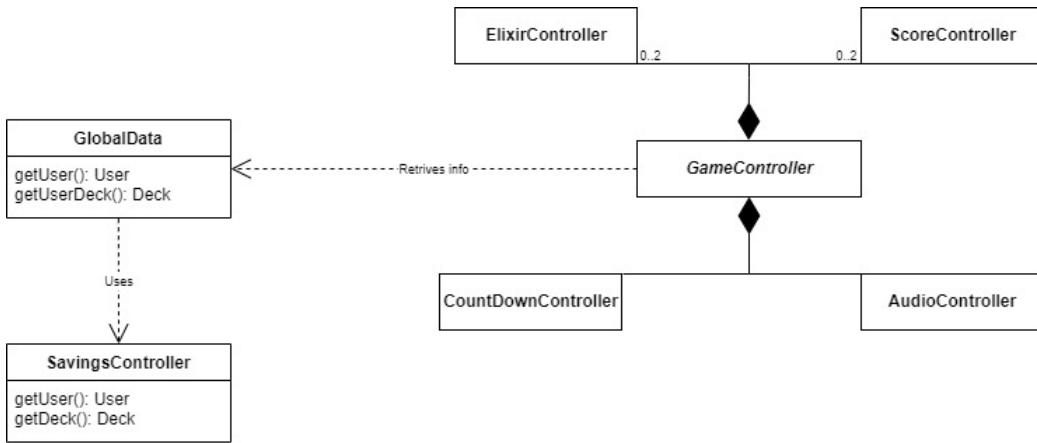


Figura 2.1: Schema UML interazione model controller.

2.1.2 Interazione Controller-View

Per quanto riguarda l’interazione fra controller e view ci siamo ampiamente appoggiati alla libreria grafica libGDX, la quale ha già internamente implementate tutte le meccaniche che definiscono il loop di gioco. In particolare offre un’interfaccia *Screen* che rappresenta una schermata dell’applicazione¹ (menu, schermata di gioco, impostazioni ecc.) da noi implementata per soddisfare gli obiettivi preposti per il progetto. Come si vede dall’immagine 2.2, dalla classe *BaseScreen*, che implementa Screen, dove viene mantenuto un riferimento al proprio Controller, viene richiamato, in maniera del tutto trasparente alle altre componenti di MVC, un metodo *update(float dt)* definito dentro la classe astratta *Controller*. Questo metodo verrà implementato dalle sottoclassi che estendono Controller, mantenendo fede al pattern *Template method*, in modo da inserirvi tutto l’occorrente per aggiornare model e view ad esso associati; inoltre, per rendere piacevole l’esperienza per l’utente, l’aggiornamento ottenuto con questa architettura viene effettuato sessanta volte al secondo. Con la suddetta architettura, sostituire in blocco la view risulta un lavoro semplice in quanto, tale operazione, risulta trasparente per model e controller, la cui unica dipendenza con la view risiede appunto nel metodo *update()*, che sarà sufficiente chiamare dalla view ogni qualvolta si voglia aggiornare ciò che è rappresentato sullo schermo. Questo indipendentemente da come la view stessa sia internamente implementata.

¹Si veda: <https://javadoc.io/doc/com.badlogicgames.gdx/gdx/latest/com/badlogic/gdx/Screen.html>

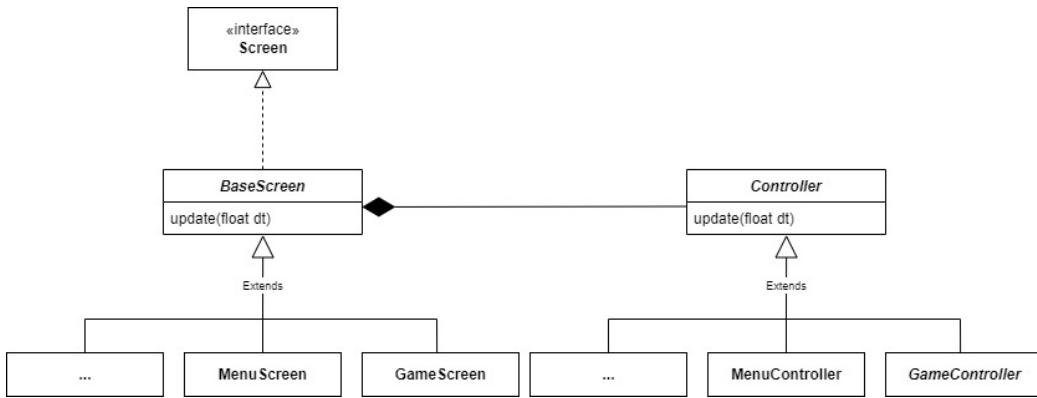


Figura 2.2: Schema UML interazione controller view.

2.2 Design dettagliato

2.2.1 Mattia Panni

Il mio ruolo all'interno del progetto è stato quello di sviluppare l'entry point e il loop di gioco, definire la meccanica con cui ci si muove da una schermata all'altra del videogioco, implementare la mappa di gioco e un sistema che consenta alle truppe, una volta schierate, di muoversi autonomamente verso il nemico o torre più vicini. Oltre a ciò ho anche definito la gerarchia di carte e torri e sviluppato la struttura dei controller per le differenti schermate di gioco.

Entry point al gioco

Come entry point al gioco, come già accennato al paragrafo 2.1.2, ho sfruttato l'API fornita da libGDX. In particolare ho esteso la classe astratta `Game` fornita dal framework in questione e, leggendo la documentazione fornita,² ho deciso di progettare tale estensione come Singleton in quanto è necessario che sia sempre presente una ed un'unica istanza facilmente accessibile da parte dei controller, affinché questi ultimi possano effettuare opportune transizioni da una finestra all'altra del gioco. Utilizzando singleton è la classe stessa che garantisce l'esistenza e l'unicità della propria istanza.

²Lee Stemkoski. *Java Game Development with LibGDX. From Beginner to Professional, Second Edition.* Apress, 2018.

Loop di gioco

Da un punto di vista di workflow, il loop di gioco può essere riassunto come mostra la figura 2.3 che segue

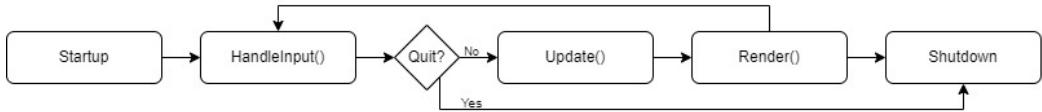


Figura 2.3: Workflow loop di gioco.

Dopo una fase di startup iniziale in cui vengono caricate le risorse grafiche, si attende un input dall’utente al seguito del quale, qualora non esprimesse la volontà di chiudere il gioco, si aggiorna il model, si renderizza la view sulla base di quanto viene deciso dal controller e si ricomincia il loop. Procedendo più in dettaglio, il loop di gioco è garantito dal fatto che libGDX, dopo aver verificato che in un dato istante esista e sia impostata come attiva una data finestra, richiamerà il metodo *render()* situato in *Game* (si veda il paragrafo 2.2.1) , il quale renderizzerà tale finestra richiamando a sua volta il metodo *render(float delta)* di *Screen*. Per quanto riguarda le fasi di gestione input e update, sono state progettate applicando il pattern *Template Method*, in quanto, sebbene siano fasi comuni a tutte le finestre, che dunque mantengono lo stesso scheletro, è necessario che le sottoclassi definiscano nel dettaglio cosa comporterà l’update. Come mostrato in figura 2.4, è stata implementata una classe astratta *BaseScreen* che raccoglie tutti quei task comuni alle varie schermate, primo fra i quali, rappresentato da un metodo astratto *update()* che verrà implementato liberamente dalle sottoclassi. Affinché questa cruciale fase venga inclusa nel loop di gioco, il metodo viene richiamato all’interno di *render(float delta)* prima di eseguire tutte le altre attività legate prettamente alla view. Di seguito è riportato uno schema UML che mostra come viene gestito il loop a livello di classi.

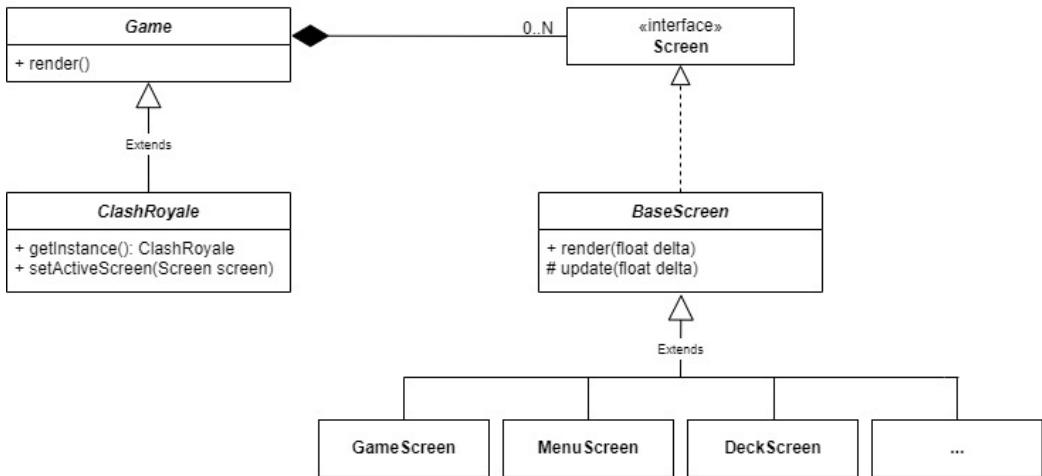


Figura 2.4: Schema UML loop di gioco.

Transizione fra schermi

Affinché il pattern MVC sia rispettato, ho ritenuto opportuno che questo punto, di fondamentale importanza, sia appannaggio dei controller. Ciascun controller conosce esattamente che parte di view controlla, pertanto, sarà sufficiente istanziarne uno e richiamare il metodo `Controller::setCurrentActiveScreen()`, per fare in modo di impostare come attiva la schermata da esso controllata. Il metodo appena descritto, opera accedendo all'istanza di Game citata nei paragrafi precedenti e richiamando il metodo `ClashRoyale::setActiveScreen(Screen screen)`³. Nel concreto, come mostra la figura 2.5 il metodo `Controller::setCurrentActiveScreen()`, definito in una super classe astratta `Controller`, è stato progettato come Template Method in quanto, sebbene sia un aspetto comune a tutti i controller di schermate, sarà compito delle istanze concrete specificare che Screen occorre attivare. Affinché, all'avvio del videogioco, compaia come prima schermata `MenuScreen`, viene istanziato all'interno del metodo `ClashRoyale::create()`, un nuovo MenuController e su di esso viene richiamato il metodo `Controller::setCurrentActiveScreen()` citato poc'anzi. Il metodo `create()`, ereditato dalla superclasse di Game, `ApplicationListener`⁴ fornita da libGDX, viene richiamato non appena l'applicazione viene lanciata, garantendo di avere subito pronta la finestra del menu. Questa architettura si presta facilmente a estendere il progetto in quanto, qualora si volesse aggiungere una nuova schermata, la transizione ad essa sarà facilmente eseguibile a

³Si veda la figura 2.4

⁴Si veda: <https://javadoc.io/doc/com.badlogicgames.gdx/gdx/latest/com/badlogic/gdx/ApplicationListener.html>

partire dal suo controller, tramite un metodo il cui scheletro è già predisposto nella super classe Controller.

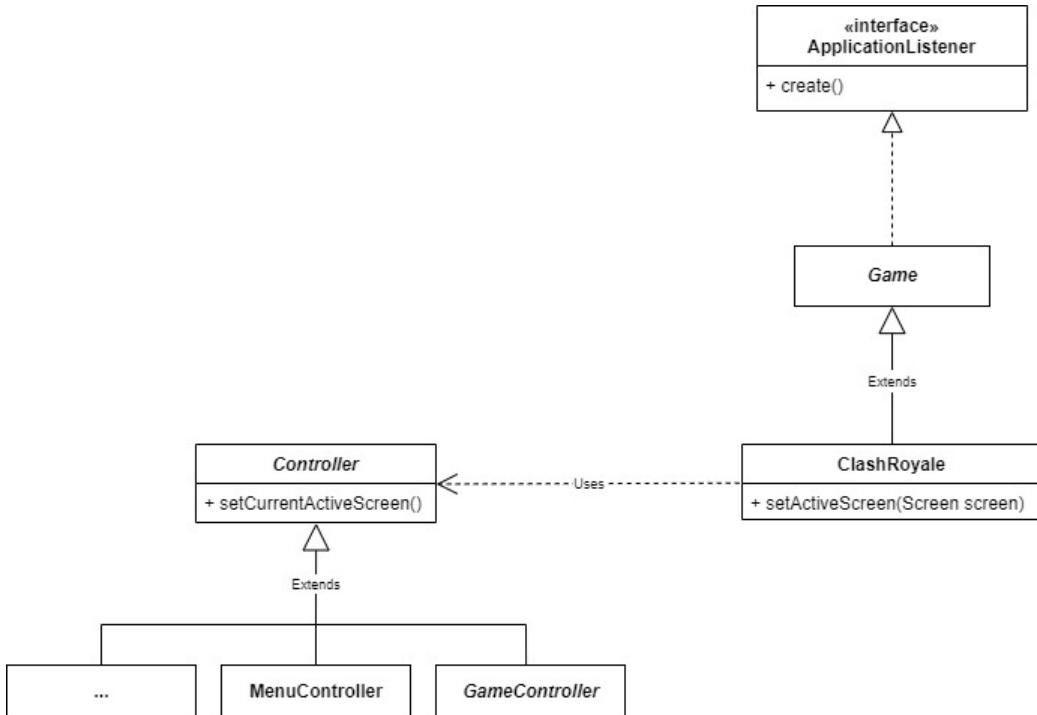


Figura 2.5: Schema UML transizione fra schermi.

Mappa di gioco e calcolo percorso per le truppe

Per questa parte, poiché strettamente di model, si è presentata la necessità di costruire una mappa di gioco che fosse totalmente svincolata dalla sua effettiva rappresentazione grafica. A tal proposito ho pensato, contestualmente con la progettazione di una strategia per il calcolo del percorso delle truppe (che tratterò successivamente in questa sezione), di modellare sudetta mappa come un grafo orientato, aciclico, non pesato. In particolare ho costruito la mappa come un insieme di rettangoli, chiamati *MapUnit*, che complessivamente formano una schacchiera. Tale scelta è mossa dal fatto che, dopo una serie di test sperimentali, è apparso impossibile costruire la mappa come insieme di punti (x,y), poiché rendeva impossibile, da un punto di vista computazionale, il calcolo dei percorsi. Ciascun MapUnit si comporta come nodo all'interno del grafo, collegato ai suoi vicini tramite degli archi non pesati. Per l'effettiva implementazione del grafo ho deciso di utilizzare

la libreria *JGraphT*⁵ in quanto possiede già implementati algoritmi di path finding fortemente ottimizzati.

Architettura dei controller di schermate

Il problema principale nello sviluppo dei controller è stato quello, in ottica di maggiore estensibilità, di svincolare l’astrazione dalle sue implementazioni specifiche. Questo aspetto è particolarmente vero per i controller della schermata di gioco: le modalità di gioco sono potenzialmente infinite (contro un’AI, online, due contro due, ecc.), occorre pertanto che l’interfaccia con cui si relaziona lo schermo, non sia legata permanentemente ad un’unica implementazione di controller. Al contempo, però, è necessario fornire all’utente, rappresentato in questo caso dagli Screen, un’astrazione per la quale, cambiamenti nelle implementazioni non impattino su di esso, così da ridurre al minimo le interdipendenze fra view e controller. Per fare ciò, come mostra la figura 2.6, ho definito una classe astratta *Controller* che includesse tutti quei metodi comuni ai vari controller di finestre. Fatto ciò, estendere tale super classe risulta un lavoro molto semplice sfruttando l’ereditarietà messa a disposizione dal linguaggio. Per quanto riguarda i controller relativi alle schermate di gioco, la questione non era risolvibile con una singola sottoclasse a causa delle molteplici modalità di gioco implementabili. A tal proposito la sottoclasse *GameController*, costituisce solamente uno scheletro che definisce tutti i metodi comuni ai vari game controller, oltre a contenere tutta la logica relativa all’utente che ha avviato la partita (in tutte le modalità la sua gesitone risulta identica). Per quei metodi che richiedono un’interazione tra l’utente e l’avversario (che sia avversario online o controllato da un’AI), come ad esempio il controllo del vincitore (che richiede di avere informazioni sia sull’utente che sul nemico), ho definito nel *GameController* dei metodi progettati come *Template Methods* che poi verranno implementati dalle sottoclassi.

⁵Si veda <https://jgrapht.org/>

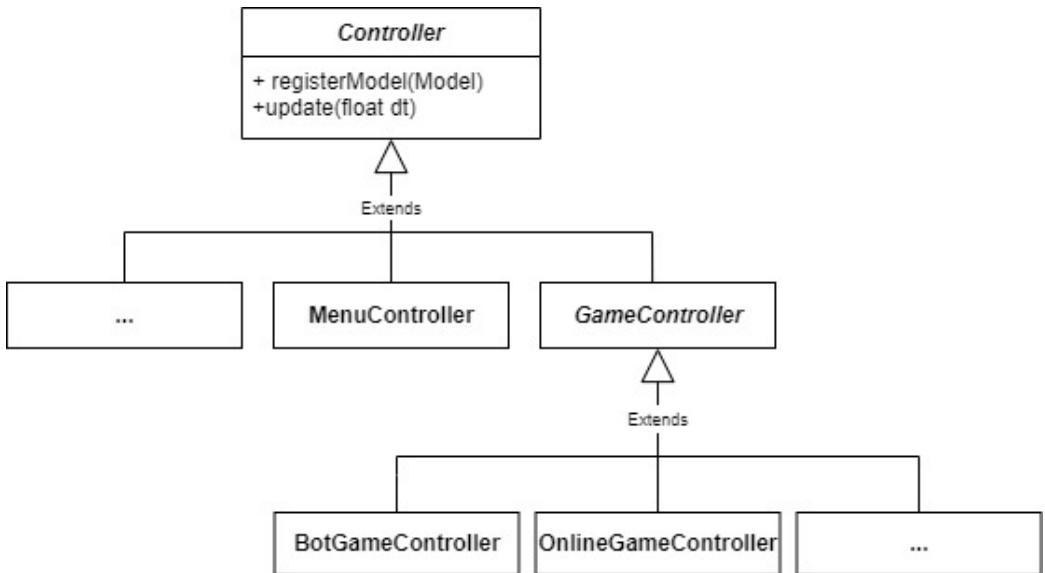


Figura 2.6: Schema UML architettura controllers.

Con il senso di poi, avrei potuto progettare tutto il sistema dei controller utilizzando il pattern *Bridge* per evitare la proliferazione di sottoclassi. A causa della mancanza di tempo, tuttavia, ho preferito optare per questa soluzione, favorendo la chiarezza a livello di codice.

2.2.2 Riccardo Fiorani

Il mio ruolo all'interno del gruppo prevedeva principalmente di sviluppare le meccaniche di gioco come la gestione delle risorse, la durata dello scontro e il piazzamento delle carte. Ho anche gestito i suoni e le animazioni degli attori.

Gestione dell'elixir

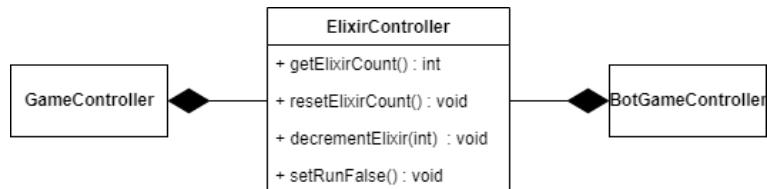


Figura 2.7: Schema UML interazioni con elixircontroller.

Problema Gestione dell'elixir sia per il bot che per il player.

Soluzione Per risolvere il problema ho creato una classe ElixirController che viene istanziata sia dal BotGameController che dal GameController, ed ha un contatore che viene incrementato di uno ogni secondo(utilizzando la libreria Timer di java che crea un nuovo task in multithreading) dall'avvio del gioco fino a un massimo di 10 e decrementato ad ogni piazzamento di una carta prendendo il proprio costo.

Durata dello scontro

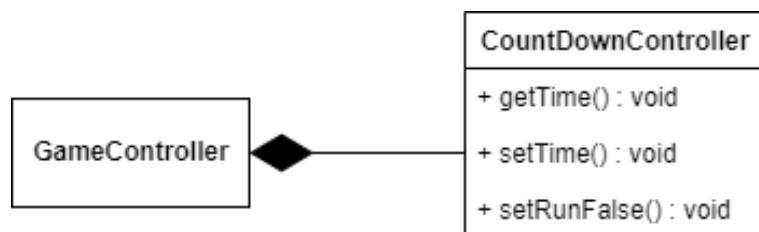


Figura 2.8: Schema UML interazioni con CountDowncontroller

Problema Gestione della durata dello scontro.

Soluzione Ho creato una classe CountDownController che viene istanziato nel GameController ed ha un contatore che viene decrementato ogni secondo(utilizzando la libreria Timer di java che crea un nuovo task in multithreading) dall'avvio del gioco, il GameController tramite il metodo getTime controlla se il tempo è uguale a zero, se è uguale il gioco viene fermato e ritorna nel menu principale.

Suoni

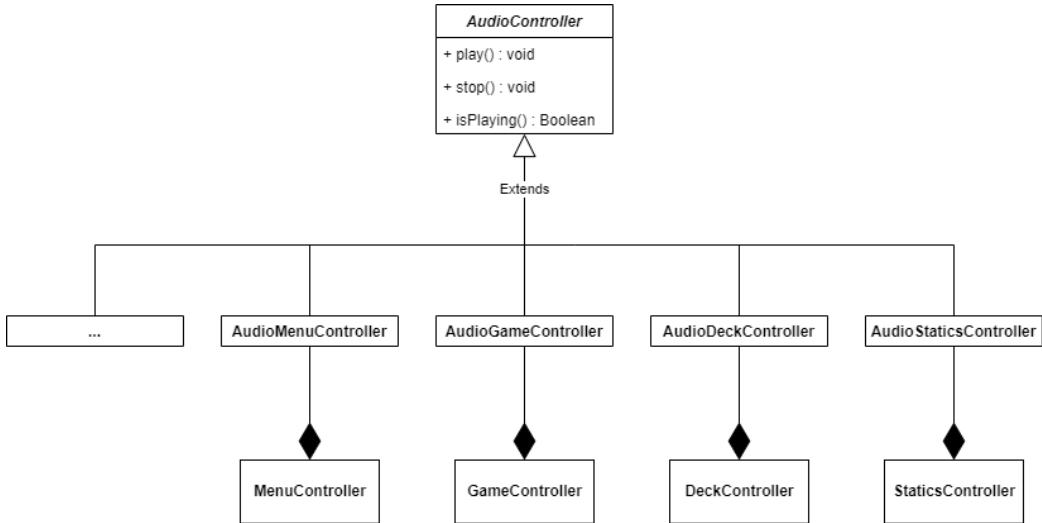


Figura 2.9: Schema UML interazioni con `AudioController`.

Problema Gestione dei suoni per il gioco utilizzando la libreria GDX.

Soluzione Ho creato una classe per ogni controller che estende la classe astratta `AudioController` contenente i metodi base per la gestione dei suoni(`play,stop`) così che ogni schermata può avere una propria base già impostata nelle varie classi.

Animazione delle entità

Problema Gestione dell'animazioni delle carte.

Soluzione Ho creato il metodo `setRotation` in `BaseActor` che riesce a sfruttare la stessa animazione per tutte le angolazioni calcola l'angolo considerando due punti il punto in cui è e il punto in cui deve andare che può essere la prossima posizione del percorso o la posizione di un nemico, ho creato anche il metodo `updateCardAnimations` in `GameController` controlla se un'entità ha un target per cambiare l'animazione da `WALKING` a `FIGHTING` e viceversa.

Piazzamento delle carte

Problema Gestione del corretto piazzamento delle carte.

Soluzione Nella classe BotGameController ho creato i metodi checkPosition, updateActorPosition, placeBotActor e placePlayerActor, il primo metodo controlla se viene piazzata all'interno della sua parte di mappa e se si ha abbastanza elisir, se le condizioni vengono rispettate restituisce un booleano TRUE, il secondo si occupa di chiamare il calcolo del percorso per ogni attore e aggiorna la propria posizione logica, utilizza moveTo di libgdx per lo spostamento a video e setRotation per aggiornare l'angolo dell'animazione, il terzo controlla il piazzamento delle truppe del player da schermo, chiama checkPosition se restituisce TRUE piazza la truppa, viene segnata come già piazzata e viene decrementato l'elisir quindi non si può più spostare, il quarto si occupa del piazzamento automatico delle carte del bot grazie alla classe BotAicontrollor che gestisce in un thread a parte quando piazzare la truppa con una posizione random all'interno della sua parte di mappa data dal metodo randomPosition e il tipo di carta scelta random tra le carte del deck usando il metodo randomCard ed esegue gli stessi controlli del primo, se i controlli non vengono rispettati le carte vengono ripristinate alla posizione iniziale.

2.2.3 Simone Bollini

All'interno del progetto ho implementato il menu principale e il menu deck che permette di scegliere le carte da utilizzare durante una partita. Inoltre ho gestito il salvataggio dei dati dell'utente in modo da essere ricaricati in nuove partite. Infine ho gestito la vittoria della partita, l'incremento o decremento di punteggio e l'eventuale avanzamento di livello.

Menu Principale

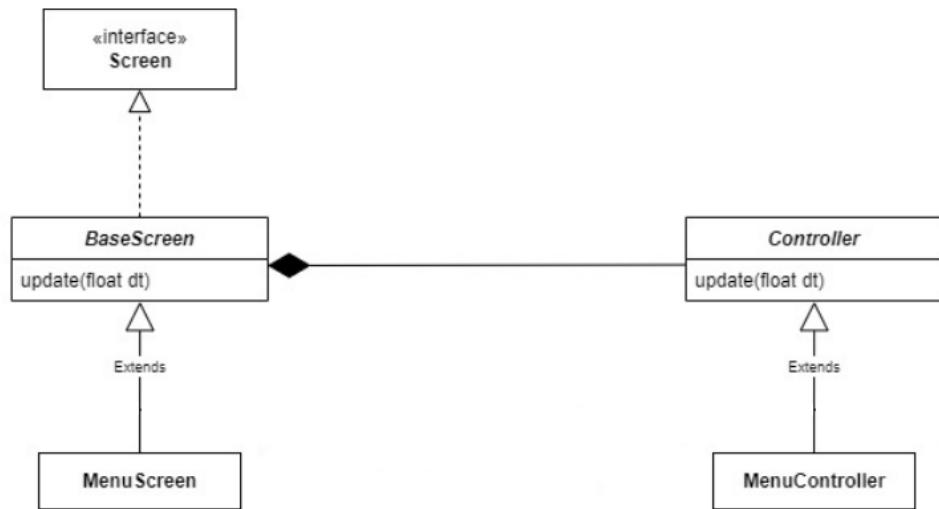


Figura 2.10: Schema UML interazioni con Menu Screen.

Problema Creare un menù per il gioco utilizzando la libreria GDX

Soluzione Studiando la libreria GDX ho creato i file menuSkin.json e menuSkinLabel.json che contengono le informazioni lette dagli Skin in GDX che servono poi per settare le singole label. Per memorizzare le posizioni delle texture ho usato il tool gdx-texturepacker.jar per la libreria GDX. Il controllo dello screen è gestito da MenuController che crea il nuovo screen selezionato nel menù creando il relativo controller.

Deck

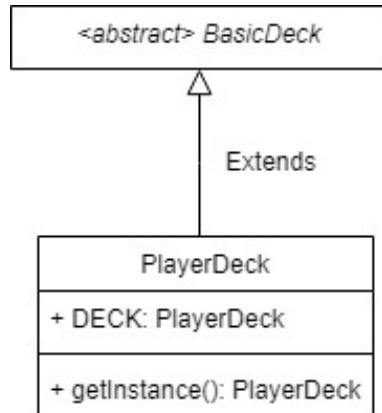


Figura 2.11: Schema UML Model DECK.

Problema Creare un deck di carte utilizzabili durante una partita e allo stesso tempo modificabili da parte dell’utente all’interno della sezione DECK.

Soluzione In questo prototipo di gioco è stato implementato solo il Deck del giocatore ma è stato scelto di partire dalla classe astratta BasicDeck in modo da poter creare nell’eventualità altri tipi di Deck. Per la classe PlayerDeck è stato utilizzato il pattern Singleton, questa classe è instanziata una sola volta ed è unica per ogni tipologia di gioco.

Utility scelta delle carte da inserire nel Deck

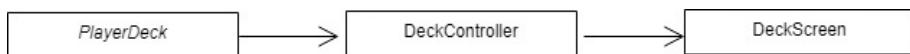


Figura 2.12: Schema UML MVC sceltaDeck.

Problema Permettere al giocatore di scegliere le 4 carte da utilizzare nella partita. .

Soluzione Come riportato in figura 2.12 in linea con lo schema del progetto ho deciso di sfruttare il pattern MVC per la corretta divisione dei compiti. Come per i dati dell’User ho cercato di salvare in un file json il deck ma questo dava problemi di reference essendo la classe Card astratta. Inizialmente funzionava creando la carta come object e poi eseguendo per ogni carta il cast allo specifico tipo, ma rimanevano comunque problemi per le carte che nella costruzione richiamavano altri metodi astratti. Abbiamo provato a costruire un GsonBuilder personalizzato per la serializzazione/deserializzazione ma senza successo.

Salvataggio

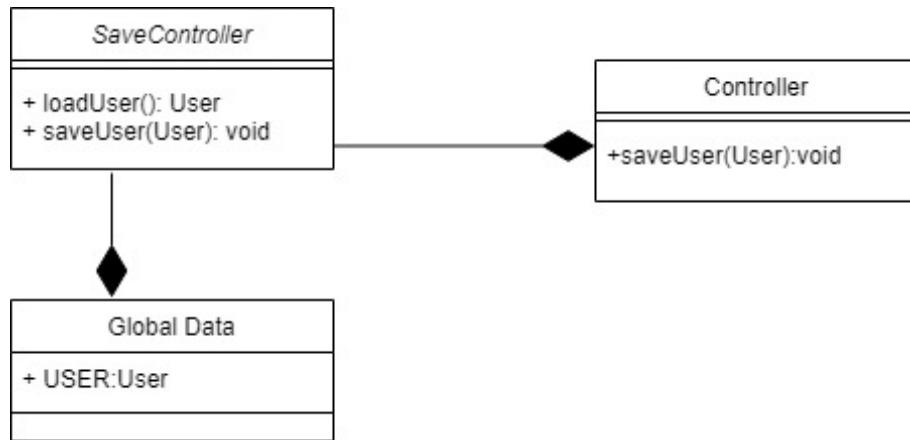


Figura 2.13: Schema UML Salvataggio Dati.

Problema Salvare i dati dell’utente in modo che vengano ricaricati in automatico in successivi accessi all’applicazione.

Soluzione La Classe SaveController gestisce i salvataggi utilizzando la libreria Guava, inizialmente avevamo optato per usare la classe Json delle utils di GDX ma poi parlando i miei compagni abbiamo deciso di usare Guava perché ci sembra più versatile. Il compito di SaveController è tramite metodi statici di passare il caricamento e il salvataggio alle altre classi. Nella schema sopra Global Data è responsabile del caricamento mentre il Controller del salvataggio.

Controllo vincitore singola partita e assegnazione punti e avanzamento livello

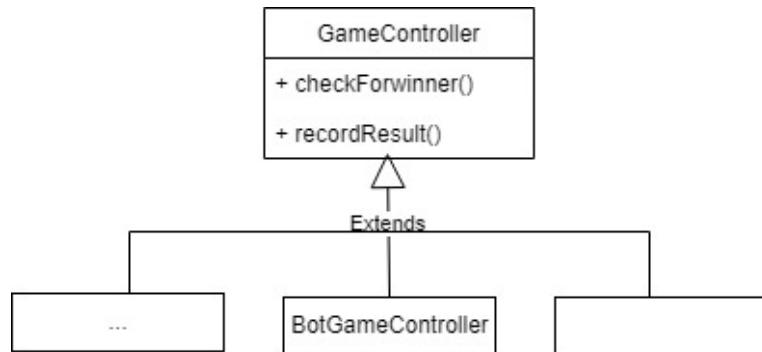


Figura 2.14: Schema UML per il controllo vincitore applicando Template Method.

Problema Come gestire concretamente la vittoria della partita.

Soluzione In linea con l'architettura del progetto i metodi `checkForwinner()` e `recordResult()` definiti nella classe astratta `GameController` sono stati progettati come Template Method in quanto sarà compito delle istanze concrete fare il super e specificare come devono essere implementati nello specifico Screen.

2.2.4 Alexandro Salvato

Riutilizzabilità delle carte

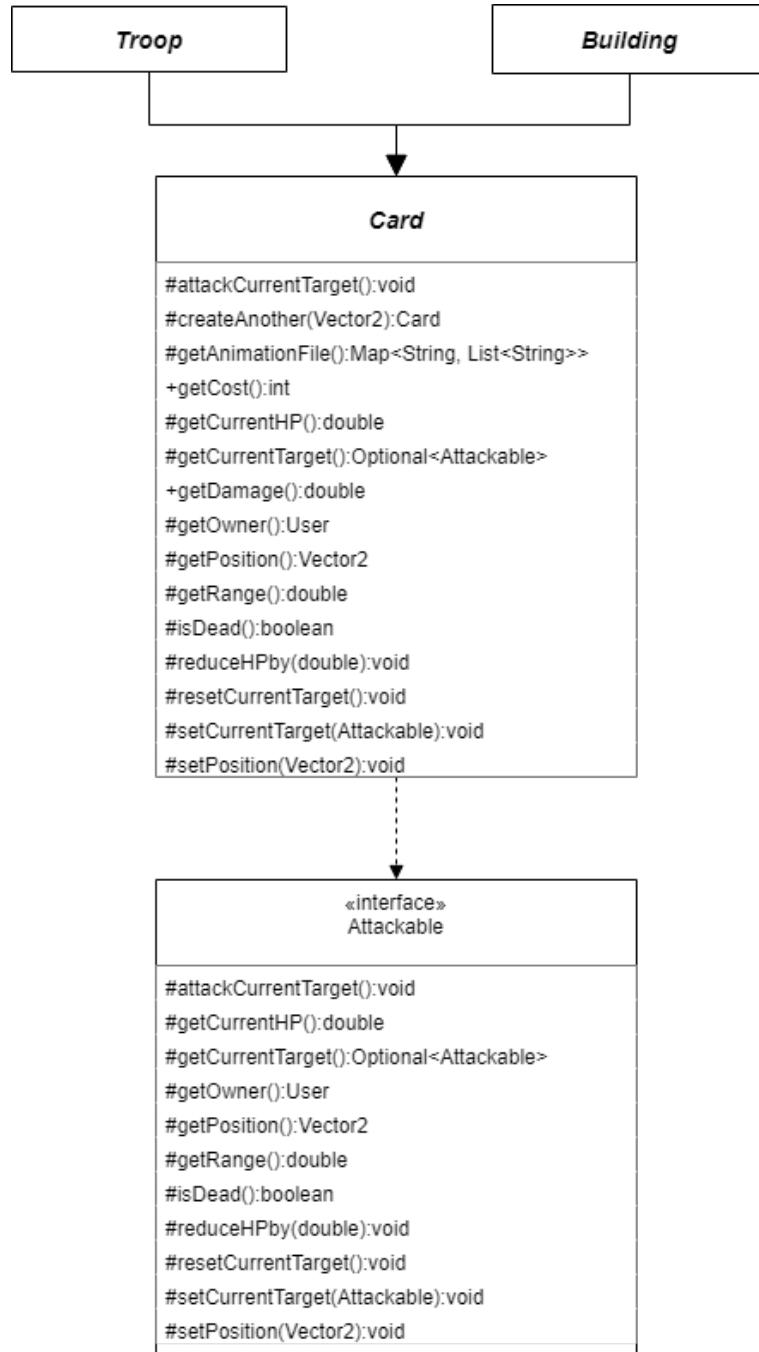


Figura 2.15: Rappresentazione UML della modellizzazione delle carte utilizzate nel gioco

Problema Esistono due tipi di carte (truppe ed edifici), che pur avendo comportamenti diversi devono essere utilizzati dall’utente allo stesso modo. Nel caso in cui si voglia aggiungere una carta non ci si deve preoccupare di riformulare la struttura del gioco.

Soluzione Il sistema per la gestione delle carte utilizza una classe astratta più generica (*Card*), ed un’interfaccia (*Attackable*) che insieme mettono a disposizione i principali metodi utilizzati all’interno delle partite, a prescindere dal tipo di carta, sia esso truppa o edificio. Ad estendere la classe *Card* ci sono altre due classi astratte, *Troop* e *Building*, a cui fanno riferimento tutte le carte utilizzabili. Per aggiungere una carta occorrerà semplicemente estendere una delle due classi appena descritte.

Aggiornamento delle statistiche di gioco

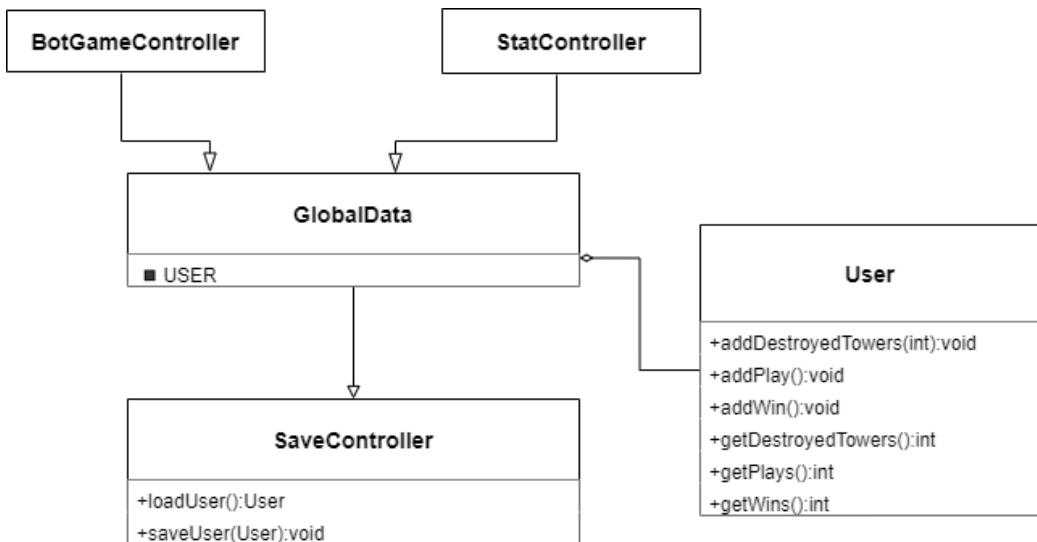


Figura 2.16: Come vengono gestiti i salvataggi e i caricamenti dei dati per il calcolo delle statistiche

Problema Mantenere i progressi dell’utente anche dopo la chiusura dell’applicazione

Soluzione Per risolvere il problema occorre appoggiarsi ad un file sul quale poter salvare, alla fine di ogni partita, i dati necessari per elaborare le statistiche di gioco. Per fare ciò vengono utilizzati dei metodi messi a disposizione dalla classe *User* per aggiornare le statistiche, viene poi utilizzata la classe

SaveController per salvare i dati su file. La stessa classe viene poi utilizzata per leggere e caricare i dati nel momento in cui l'utente apre la schermata dedicata alle statistiche dal menù. Un'altra possibile soluzione poteva essere quella di creare un'apposita classe per la lettura e scrittura da file per tenere traccia esclusivamente delle statistiche. Siccome era già presente un'altra classe che si occupava di lettura/scrittura si è deciso di aggiungere alcuni metodi per la gestione delle statistiche al posto di una classe apposita.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il progetto supporta l'esecuzione di una serie di test automatizzati basati su JUnit¹. È anche possibile eseguirli da linea di comando mediante Gradle.

Ad accompagnare l'applicazione vengono forniti i seguenti test:

- **TestUsers:** si occupa di verificare il corretto funzionamento degli User effettuando dei test su tutti i metodi pubblici messi a disposizione dalla classe User.
- **TestTroops:** analizza il comportamento delle truppe. Vengono testati diversi metodi propri delle truppe come la creazione di unità, l'attacco di altre truppe e la gestione del danno.
- **GdxTest:** classe che, di fatto non testa alcun aspetto dell'applicazione. Inserita poiché rappresenta uno scheletro per tutte quelle suite di test che hanno bisogno di funzionalità messe a disposizione da libGDX. Il framework in questione, infatti, è in grado di funzionare solamente se prima viene istanziato un *ApplicationListener*. Fare ciò nei test era problematico, in quanto, durante l'esecuzione di ognuno di essi, sarebbe dovuta comparire un'interfaccia grafica. A tal proposito libGDX offre un'implementazione di ApplicationListener backend pensata per i server². Questa implementazione, infatti, non istanzia alcuna interfaccia pur creando a tutti gli effetti un ApplicationListener.

¹In particolare è stata usata la versione Jupiter

²Si veda <https://github.com/libgdx/libgdx/tree/master/backends/gdx-backend-headless>

- **GameMapTest:** pensata per verificare il corretto funzionamento della mappa di gioco. Inclusi i metodi per il calcolo dei percorsi basati sull'algoritmo A*.
- **AudioControllerTest:** verifica il corretto funzionamento del audio di gioco.
- **CountDownControllerTest:** verifica il corretto funzionamento della durata del gioco.
- **ElixirControllerTest:** verifica il corretto funzionamento dell'elisir durante il gioco.

3.2 Metodologia di lavoro

Per consentire a ogni membro del gruppo di avere una discreta autonomia nello sviluppo del progetto, abbiamo fatto largo utilizzo del DVCS **git**. La modalità di lavoro è stata quella di aprire un nuovo branch per lo sviluppo di feature di particolare rilevanza. Dopo aver testato il loro corretto funzionamento, avendo stabilito che fossero pronte, sono state integrate nel main branch **MASTER**. Per quanto riguarda i ruoli, è stata mantenuta, in linea di massima, la suddivisione che ci eravamo preposti. Tuttavia, fasi di progettazione critiche, come ad esempio lo sviluppo delle dinamiche di attacco o costruzione della mappa di gioco, sono state affrontate in gruppo.

3.2.1 Mattia Panni

- Progettazione dell'architettura delle entità in gioco all'interno del progetto. In particolare ho definito l'interfaccia **Attackable**, le classi **Card**, **Troop**, **Building**, **Tower** e le relazioni che intercorrono fra di loro, l'effettiva implementazione è stata fornita da Riccardo Fiorani e Alessandro Salvato.
- Progettazione e implementazione di **User** e **Bot** con annessi test automatizzati.
- Realizzazione del pattern MVC, suddivisione del progetto in aspetti di view: **BaseScreen** e sottoclassi, **BaseActor**, **DragAndDropActor** e sottoclassi, controller **Controller** e sottoclassi e model: **GameModel** e sottoclassi. L'implementazione di alcuni metodi, tra cui quelli per dare una rotazione alla texture degli attori (*BaseActor::setRotation*, *BaseActor::setAngle*), e per aggiornare le posizioni di questi ultimi, nei

game controller (*BotGameController::updateActorPositions*, *BotGameController::placeBotActor*).

- Realizzazione del loop di gioco facendo uso delle API fornite da libGDX, tra cui **Game** e **Screen**.
- Sviluppo della mappa di gioco composta da **GameMap** e **MapUnit**, sviluppo degli algoritmi per il calcolo dei percorsi. Riccardo Fiorani e Alessandro Salvato hanno contribuito a correggere bug relativi alla costruzione del grafo.
- Creazione dell'entry-point al videogame tramite singleton **ClashRoyale**.
- Sviluppo delle utilities per i vettori **VectorUtilities** e per le animazioni **AnimationUtilities**.
- Ho contribuito alla realizzazione di **SaveController** e alla sua effettiva istanziazione all'interno dei GameController.

3.2.2 Riccardo Fiorani

- progettazione delle meccaniche di gioco creando le classi **ElixirController**, **CountDownController** e **BotAiController**.
- rotazione degli attori con l'implementazione di alcuni metodi, tra cui quelli per dare una rotazione alla texture degli actor (*BaseActor::setRotation*, *BaseActor::setAngle*)
- movimento degli attori con il metodo *BotGameController::updateActorPositions*
- piazzamento degli attori con i metodi (*BotGameController::placeBotActor*, *GameController::placePlayerActor*)
- mi sono occupato anche della risoluzione dei bug grafici regolando con degli offsetti le classi **BaseActor**, **DragAndDropActor**, **GameMap** e **MapUnit** e ho lavorato insieme a Mattia Panni sulle classi **GameModel** **BotGameModel** e **SaveController**.

3.2.3 Simone Bollini

3.2.4 Alexandro Salvato

- Implementazione dell’interfaccia Attackable tramite le classi: Card, Troop, Building, Archer, Barbarian, Giant, MiniPekka, Valkyrie, Wizard, InfernoTower;
- StatController: agisce da tramite tra SaveController e StatScreen;
- StatScreen: gestisce la parte di view relativa alla finestra delle statistiche utente.

3.3 Note di sviluppo

3.3.1 Mattia Panni

- Stream e lambda expressions
- Uso di Optional
- Librerie esterne: LIBGDX per il loop di gioco e le interfacce grafiche di base. JGRAPHT per la modellazione della mappa di gioco come grafo e l’utilizzo di A* per il calcolo dei percorsi. GOOGLE GUAVA per la serializzazione e deserializzazione di dati. Mockito per la creazione di mock objects (oggetti fintizi) della classe *GL20*³ in fase di testing.
- JUnit Jupiter per il testing automatizzato.
- Utilizzo del build system Gradle.

La classi *BaseScreen* e *DragAndDropActor* sono tratte dal libro JAVA GAME DEVELOPMENT WITH LIBGDX citato al paragrafo 2.2.1. Per il calcolo dei percorsi delle truppe, tramite algoritmo A*, ho tratto ispirazione dal progetto <https://github.com/DanySK/Student-Project-OOP20-Campanozzi-Corradino-Micheli-Zammarchi-HotlineCesena>. Per lo sviluppo della classe scheletro *GdxTest*, volta a consentire l’uso dei tool forniti dal framework in fase di testing, ho riadattato il codice trovato qui <https://stackoverflow.com/q/53031633/10580410>

³API di OpenGL.

3.3.2 Riccardo Fiorani

- Uso di Stream e lambda expressions
- Librerie esterne: LIBGDX, GOOGLE GUAVA
- Libreria per i test automatizzati: JUnit.
- Uso del build system Gradle.

3.3.3 Simone Bollini

- Stream e lambda expressions di base. Librerie esterne:
- LIBGDX per lo sviluppo del gioco e della grafica.
- GOOGLE GUAVA per il salvataggio dei dati.
- Il tool Gdx-texturepacker per gestire le texture richiesto dalla libreria GDX.

3.3.4 Alexandro Salvato

- Uso di librerie di terze parti: JavaFX, per la gestione delle immagini;
- utilizzo di Optional;
- uso di lambda expression;

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Mattia Panni

Credo di essere maturato come programmatore. Non solo nel contesto object-oriented, a me completamente sconosciuto prima dello scorso settembre. Più in generale, infatti, credo di aver acquisito maggiore consapevolezza di ciò che significhi sviluppare un progetto di una certa portata, svolto, per di più, in team: esperienza del tutto nuova per me. Tuttavia, per mancanza di esperienza, credo che il lavoro svolto abbia qualche lacuna. In particolare credo che la parte preliminare di progettazione sia un pò carente o quantomeno confusionaria. Questo, forse, anche a causa del fatto di aver sottovalutato la problematicità nel fare propria una libreria come libGDX, con un ruolo, per altro, preponderante all'interno del progetto. La situazione è stata ulteriormente aggravata dal fatto che la documentazione in rete di questo framework è poca, in prevalenza datata e a tratti discordante. A tal proposito è stato acquistato il libro JAVA GAME DEVELOPMENT WITH LIBGDX già citato in precedenza. Sebbene sia stato d'aiuto nel capire la struttura del framework, il codice fornito, per quanto corretto, è, a mio modesto avviso, qualitativamente pessimo e, pertanto, quasi inutilizzabile. Tutto questo ha sottratto tempo, come già detto, alla progettazione, che dunque potrebbe essere migliorata. In particolare, temo di aver adottato poche strategie progettuali note in letteratura (pattern). Se in futuro dovessi rimettere mano al progetto, la prima cosa che farei è senz'altro quella di approfondire questo aspetto, ad esempio usando Bridge nei controller. L'interazione con gli altri membri del gruppo è stata prevalentemente buona, anche se non sono mancate situazioni di sovrapposizione dei ruoli o peggio, di scarsa partecipazione da parte di qualcuno. Nel complesso, comunque, ritengo questa esperienza

estremamente formativa e affascinante. Ho apprezzato davvero tanto Java e più in generale la programmazione ad oggetti.

4.1.2 Riccardo Fiorani

Nel complesso mi ritengo soddisfatto di ciò che sono riuscito a fare, è stata una sfida realizzare questo progetto in team essendo la mia prima esperienza, il codice poteva essere strutturato meglio dall'inizio ma ci siamo concentrati su come usare la libreria perchè per noi era sconosciuta quindi tramite il libro JAVA GAME DEVELOPMENT WITH LIBGDX e i vari forum online abbiamo acquisito una buona conoscenza, di sicuro sono andato fuori dalle ore consigliate ma ho imparato molto sia a conoscere meglio java e libgdg che come gestire il tempo.

4.1.3 Simone Bollini

Sono molto contento dei progressi fatti dall'autunno scorso ad oggi in Java. Ho ancora tanto da imparare ma per me è stato fino a qui un bellissimo percorso. Devo ringraziare alcuni componenti del gruppo che da otto mesi a questa parte, ovvero da quando abbiamo deciso di fare il progetto insieme, mi hanno sempre aiutato prima nell'esame pratico e poi in ogni dubbio nell'elaborato. Penso che questo confronto sia stato utile a tutti e anche loro sono molto migliorati in questi mesi. Al di fuori dell'università ho iniziato a lavorare utilizzando Java e questo non sarebbe potuto accadere senza questo bellissimo corso.

4.1.4 Alexandro Salvato

Ritengo che ci sia stato un problema nella gestione del tempo, in quanto si è inizialmente dato ampio spazio alla parte grafica sacrificando parti più funzionali, ci si sarebbe potuti organizzare meglio, per esempio, sulla progettazione generale del gioco, anche in maniera più condivisa. Repeto, infine, questa esperienza altamente formativa poichè, oltre ad essere funzionale ad un lavoro futuro, rappresenta la mia prima esperienza di lavoro di gruppo su un periodo di tempo piuttosto lungo.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Mattia Panni

Come già detto, ritengo il corso di OOP uno fra i più affascinanti, ma anche impegnativi, da me affrontati. L'unico appunto che mi sento di fare è il seguente: per la mia formazione, credo che la conoscenza approfondita di pattern (che possono essere, di fatto, applicati a qualsiasi linguaggio), sia fondamentale. Penso perciò che quell'aspetto potrebbe, nei corsi futuri, essere trattato maggiormente.

4.2.2 Simone Bollini

Purtroppo l'utilizzo della libreria GDX ha portato via più tempo di quello da noi previsto per capirne le funzionalità. Per quel che riguarda il corso lo considero ben strutturato viste le ore disponibili che sarebbe stato bello se fossero state anche di più.

Capitolo 5

Guida utente

5.1 Menu

Come mostra la figura 5.1, al lancio dell'applicazione apparirà, per prima cosa, il menù principale.



Figura 5.1: menu

Da esso sarà possibile, semplicemente interagendo con dei click sui bottoni, far accadere quanto segue:

- BATTLE: consente di effettuare una nuova partita.
- DECK: consente di personalizzare il mazzo.

- STATS: permette di visualizzare le statistiche dell’utente.
- EXIT: termina l’applicazione.

5.2 Deck

Alla pressione del tasto DECK appare un’interfaccia che consente di selezionare le quattro carte che si vogliono usare in partita.



Figura 5.2: deck

Come da figura 5.2, si potrà rimuovere una voce da *Battle deck* (che rappresenta la lista di carte da portare in battaglia), selezionandola e premendo il bottone REMOVE. Se invece si vuole aggiungere una carta dalla *Card Collection* sarà sufficiente eseguire lo stesso procedimento visto sopra, premendo questa volta ADD, questo a patto che nel mazzo non siano già presenti quattro carte. Con RETURN è possibile ritornare al menù principale.

5.3 statistics

Alla pressione del bottone STAT appare una schermata relativa alle statistiche di gioco dell’utente che ha avviato l’applicazione.



Figura 5.3: statistics

- return ritorna al menu principale

5.4 Game

Alla pressione di GAME viene creata una nuova partita.



Figura 5.4: game

- Score mostra i punteggi correnti sia dell’utente che del bot (indica il numero di torri attualmente distrutte da entrambi).
- Elixir mostra la quantità di elisir attualmente a disposizione.
- Time left mostra il tempo rimanente prima della fine del match.

per giocare basta spostare una fra le quattro carte situate in basso sullo schermo, verso una casella della propria metà campo.

Capitolo 6

Esercitazioni di laboratorio

6.1 Mattia Panni

Purtroppo non ho mai allegato le mie soluzioni al forum, tuttavia sono comunque visionabili nel mio github.

- Laboratorio C#: <https://github.com/paniniDot/lab-csharp-simple>
- Laboratorio 10: <https://github.com/paniniDot/OOP2021-Lab10>
- Laboratorio 9: <https://github.com/paniniDot/OOP2021-Lab09>
- Laboratorio 8: <https://github.com/paniniDot/OOP-Lab08>
- Laboratorio 7: <https://github.com/paniniDot/OOP-Lab07>
- Laboratorio 6: <https://github.com/paniniDot/OOP-Lab06>

6.2 Simone Bollini

- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=88829#p138270>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=89272#p139421>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p140525>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=91128#p141508>