

Relazione progetto

Pogeshi

Guidi Stefano

Paolucci Matteo

Verazza Claudio

Marcaccini Francesco

Indice

I. Analisi	3
1.1 Requisiti	4
1.2 Analisi e modello del dominio	5
II. Design	6
2.1 Architettura	6
2.2 Design dettagliato	9
III. Sviluppo	24
3.1 Testing automatizzato	24
3.2 Metodologia di lavoro	27
3.3 Note di sviluppo	29
IV. Commenti finali	34
4.1 Autovalutazione e lavori futuri	34
4.2 Difficoltà incontrate e commenti per docenti	36
A. Guida utente	37
B. Bibliografia	42

I Analisi

Il team si impegna a sviluppare un gioco bidimensionale con visuale dall'alto, ispirato al gioco Slay the Spire.

L'obiettivo del gioco è affrontare tutti i nemici sullo schermo, fino a sconfiggere il boss finale che apparirà una volta sconfitti tutti gli altri, usando le carte inserite nel proprio mazzo (deck).

- Il deck può essere composto dal giocatore prima di entrare nella fase vera del gioco, oppure utilizzare quelli già composti in precedenza.
- Durante la fase di gioco, il giocatore si potrà muovere per raggiungere i nemici presenti sulla mappa, con i quali interagire per iniziare la fase di combattimento.
- La fase di combattimento sarà a turni, durante la quale il giocatore potrà usare le proprie carte, che possono essere di vario tipo: attacco, difesa e, opzionalmente, altri effetti.
- Giocare le carte richiede "mana", che si ricaricherà e aumenterà con lo scorrere dei turni.
- Il giocatore vincerà lo scontro quando la salute del nemico scende sotto lo zero, e perderà quando la propria finirà. La salute del giocatore NON si rigenererà fino a fine partita.
- Una volta sconfitto il boss: la partita finisce, il giocatore riceverà una carta casuale con cui poter modificare il proprio deck prima della prossima partita e verranno aggiornate le proprie statistiche di gioco (es. numero vittorie, mostri sconfitti, ecc.).

1.1 Requisiti

Requisiti funzionali:

- Creazione della mappa
- Movimento del giocatore
- Interazione con i nemici
- Lancio delle carte e dei turni
- Gestione mana e vita
- Gestione deck con aggiunta e rimozione delle carte
- Gestione statistiche di gioco, con salvataggio

Requisiti opzionali:

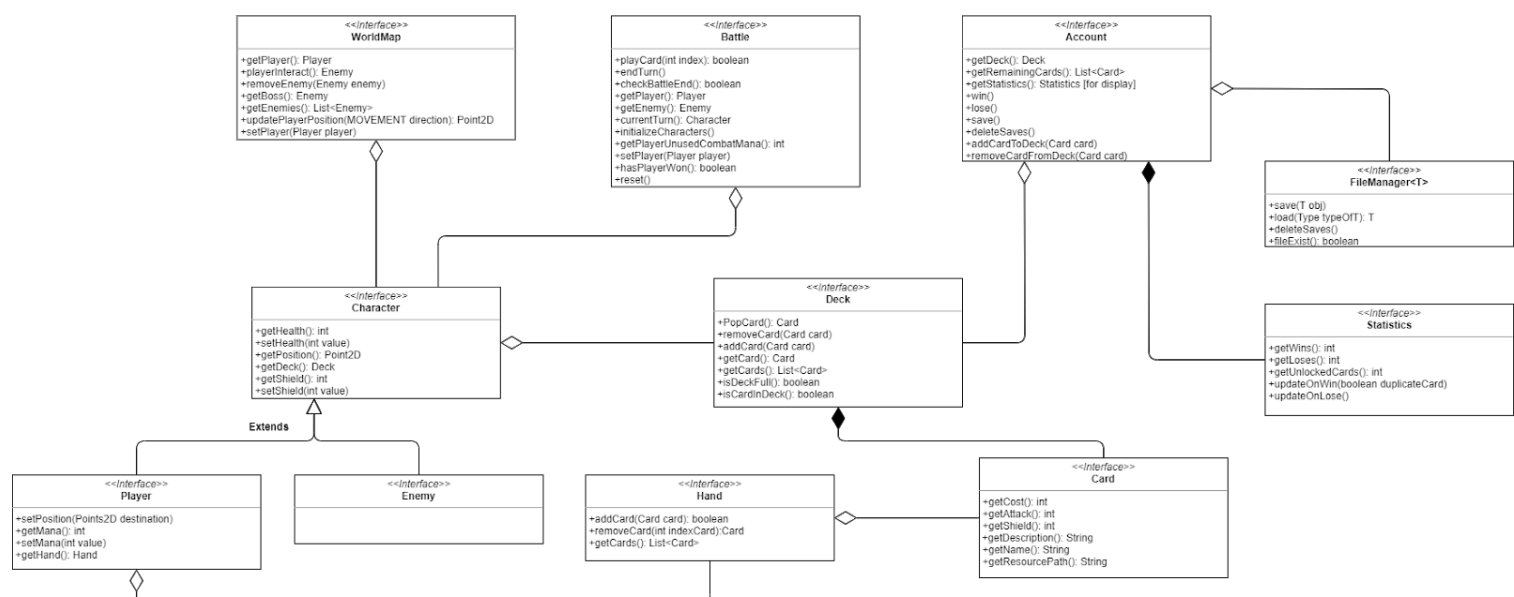
- Creazione di piani diversi
- scoreboard/leaderboard
- diversificazione nemici e boss
- implementazione di inventario e sua gestione
- implementazione di soldi, la loro gestione e un negozio in cui poter comprare oggetti, i quali avranno vari effetti.

Requisiti non funzionali:

- Fluidità generale del gioco

1.2 Analisi e modello del dominio

L'utente possiede un proprio account dove vengono mantenute le proprie statistiche di gioco e il deck di carte creato, che può essere modificato prima dell'inizio della partita. Una volta iniziata una partita, il giocatore, rappresentato dal suo personaggio, potrà spostarsi all'interno della mappa di gioco ed interagire con i nemici per sfidarli in una battaglia. Quest'ultima consiste in un duello fra giocatore e nemico, nel quale, a turni, potranno giocare delle carte che permetteranno di attaccare e/o difendersi dall'avversario. Le carte verranno pescate dal deck all'inizio di ogni turno e aggiunte alla propria mano. Una volta sconfitti tutti i nemici sarà possibile combattere un boss, nelle stesse modalità descritte precedentemente. In caso di vittoria il giocatore sarà ricompensato con una nuova carta, che potrà aggiungere al proprio deck, mentre le sue statistiche verranno aggiornate anche in caso di sconfitta.



Il Design

2.1 Architettura

L'architettura del software segue il pattern architetturale MVP (Model-View-Presenter). Più nello specifico è stata adottata la variante MVP Passive View (cioè la View è una componente passiva nell'arco dell'esecuzione che non svolge attivamente nessuna operazione, ma può solo venire aggiornata dal Controller).

Vi è un controller principale (il MainController) che gestisce il cambio di contesto tra i vari controller, e viene notificato da essi quando hanno finito di svolgere le loro funzioni, o richiedono comunque uno spostamento del workflow del programma.

Sostanzialmente il MainController è un Observer dei Controller, dai quali viene notificato ogni qualvolta è necessario trasmettere dei messaggi ad altri controller .

Per quanto riguarda l'entry-point del Model si è scelto di gestirlo in maniera composita: vi sono 3 submodel(WorldMap, Battle e Account) referenziati dai rispettivi sotto controller

Il compito principale del MainController è intercettare le notifiche spedite dai vari controller, elaborarle e, nel caso, reindirizzare il flusso di esecuzione sul sotto-controller corretto (il MainController è concettualmente molto simile ad un FrontController, ma avendo funzionalità più ridotte risulta più simile ad un Router, oggetto adibito a reindirizzare i messaggi all'entità interessata, che, in questo caso, sono le informazioni di contesto necessarie per far operare un controller).

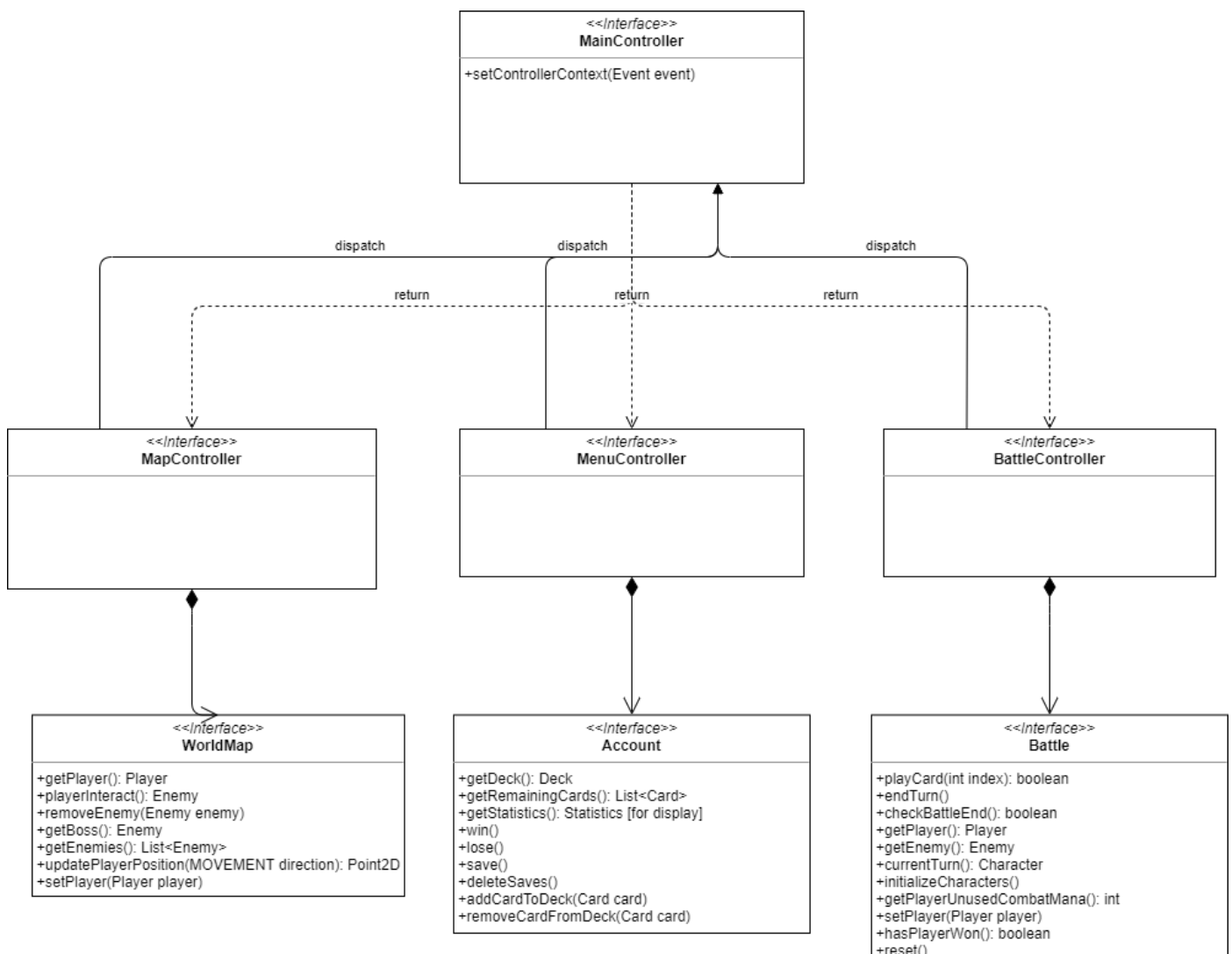
I vantaggi dell'adottare questa architettura sono i seguenti:

- Una maggiore modularità nella gestione del passaggio di contesto tra i vari Controller, in quanto devono notificare un controller comune che gestisce i passaggi di contesto, senza assolvere quest'ultima operazione loro stessi.
- Grazie all'adozione dell' MVP Passive View il software ottiene una maggiore testabilità delle sue componenti (la maggior parte del flusso di esecuzione è gestito nei Controller).

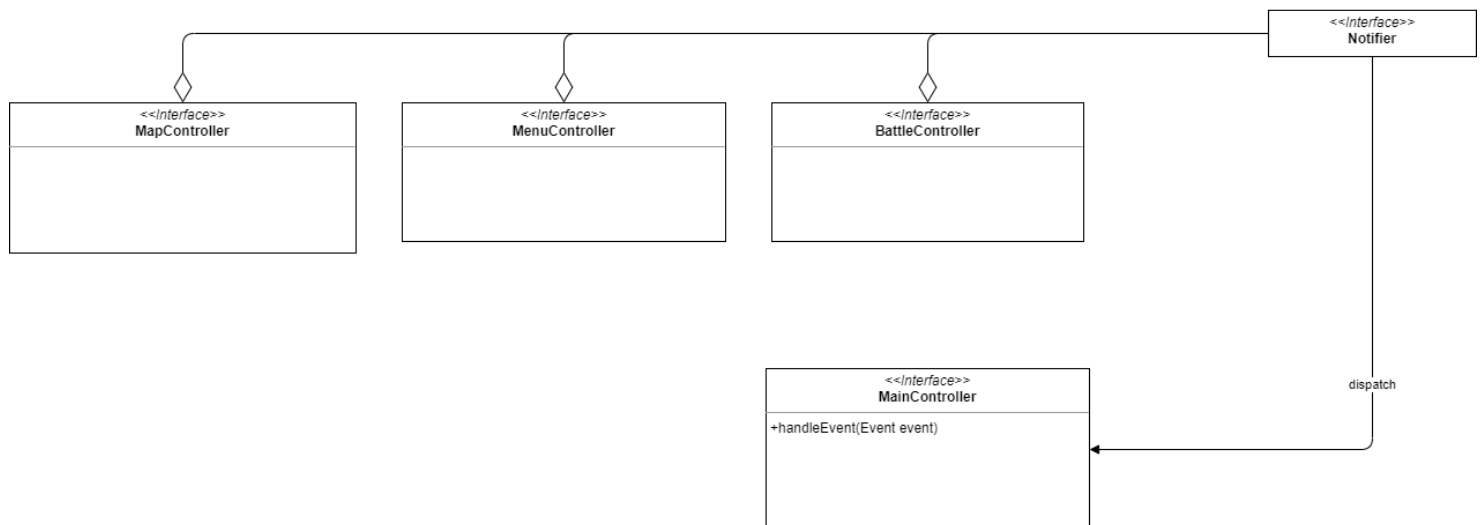
- L'eventuale sostituzione della View non impatta in alcun modo sul Controller né tantomeno sul Model(a patto questa rispetti il contratto dichiarato) in quanto ogni interazione deve essere delegata al Controller.

Il meccanismo di notifica(logica Observable-Observer) è utilizzato nella comunicazione tra Controller(Observable) e MainController(Observer); il MainController comunica con i controller fornendogli i dati necessari alla loro esecuzione.

Infine ogni controller gestisce una propria view della quale ha la responsabilità di aggiornare i dati una volta terminate le operazioni sul proprio model.



Architetture generale di interazione Model-Controller: il MainController imposta il contesto di esecuzione passando il controllo ad uno dei 3 controller; questi interagiscono liberamente con i relativi Model. I tre Model non possono interagire liberamente tra loro.



Tutti i sub controller devono implementare un'interfaccia di notifica(Notifier) che realizza la logica di comunicazione Observer-Observable.

Il controller è istanziato con in composizione il suo rispettivo model e view(MVP).

La view è una componente passiva aggiornata direttamente dal Controller e i cui eventi vengono intercettati da funzioni di handler del controller.

Fintanto che il contratto della view è rispettato, i controller possono aggiornare le proprie view senza problemi di consistenza (anche se le view dovessero lavorare con framework gui diversi tra loro ciò non consisterebbe un problema).

NOTA SULLA NOTAZIONE

Nonostante l'uso di un Presenter, ci si è riferiti ad essi(sia nella definizione all'interno nel software che nel documento corrente) come Controller. Il motivo di ciò è porre attenzione sul ruolo svolto dal Controller(quello di Presenter dei dati dal model alla View), cercando di mantenere però un termine di notazione comune alle architetture derivate da MVC).

2.2 Design dettagliato

Matteo Paolucci

Le carte

Le carte sono un POJO. L'unico problema qui sta nell'istanziarle poiché possiedono molti parametri e i costruttori ogni volta sarebbero interminabili. Per risolvere il problema ho utilizzato il pattern: **Builder Pattern For Class Hierarchies**¹. Ho preferito questa variante del pattern builder per privatizzare il costruttore e per creare un template da cui partire, nel caso in cui si volesse creare un nuovo tipo di carta con nuove caratteristiche ed effetti. In questo caso la classe astratta da estendere è `AbstractCard` e il Builder da estendere è la classe statica `AbstractCard.Builder` innestata nella classe `AbstractCard`. In questo pattern è presente anche il pattern **Template Method**², infatti il metodo `build()` di `AbstractCard.Builder` è il template method di cui va fatto l'override di volta in volta per ritornare la giusta carta e per verificare che l'oggetto sia costruito correttamente.

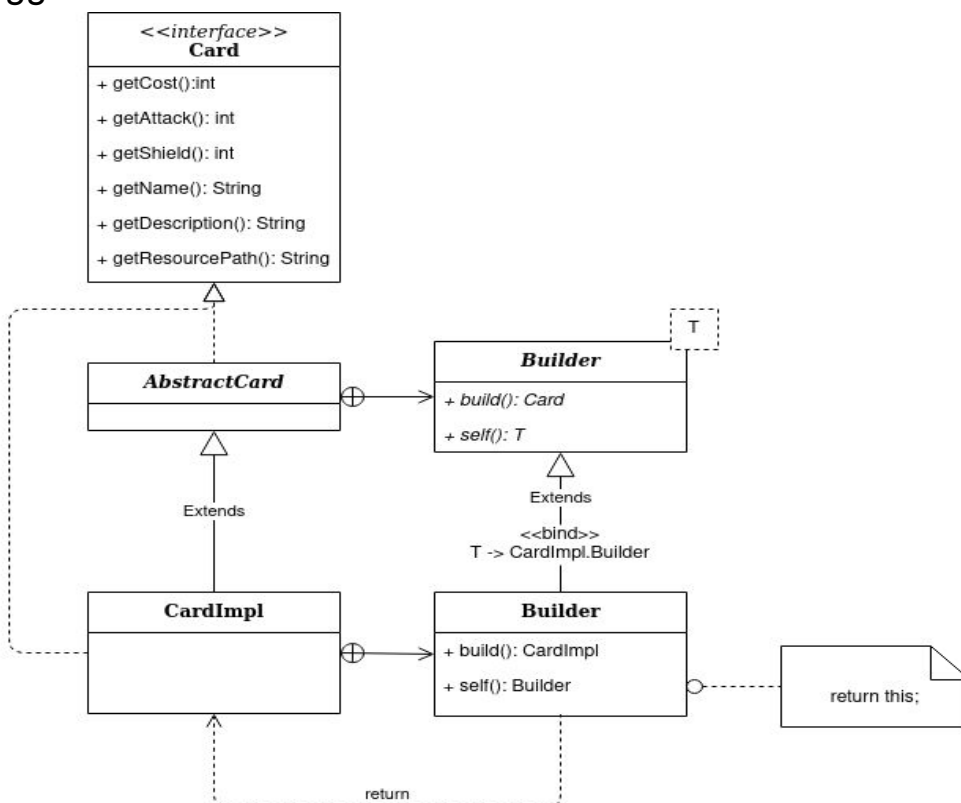


Figura 2.1 rappresentazione UML del pattern Builder For Class Hierarchies

DeckCreationController

Per strutturare il controller del Deck ho utilizzato il pattern **Strategy**². In questo caso gli algoritmi interscambiabili (Strategy) sono sia il Model (in questo caso Account) che la View (in questo caso DeckCreationView) che grazie al polimorfismo e l'incapsulazione possono essere sostituiti a run-time, mentre il Context è il DeckCreationController.

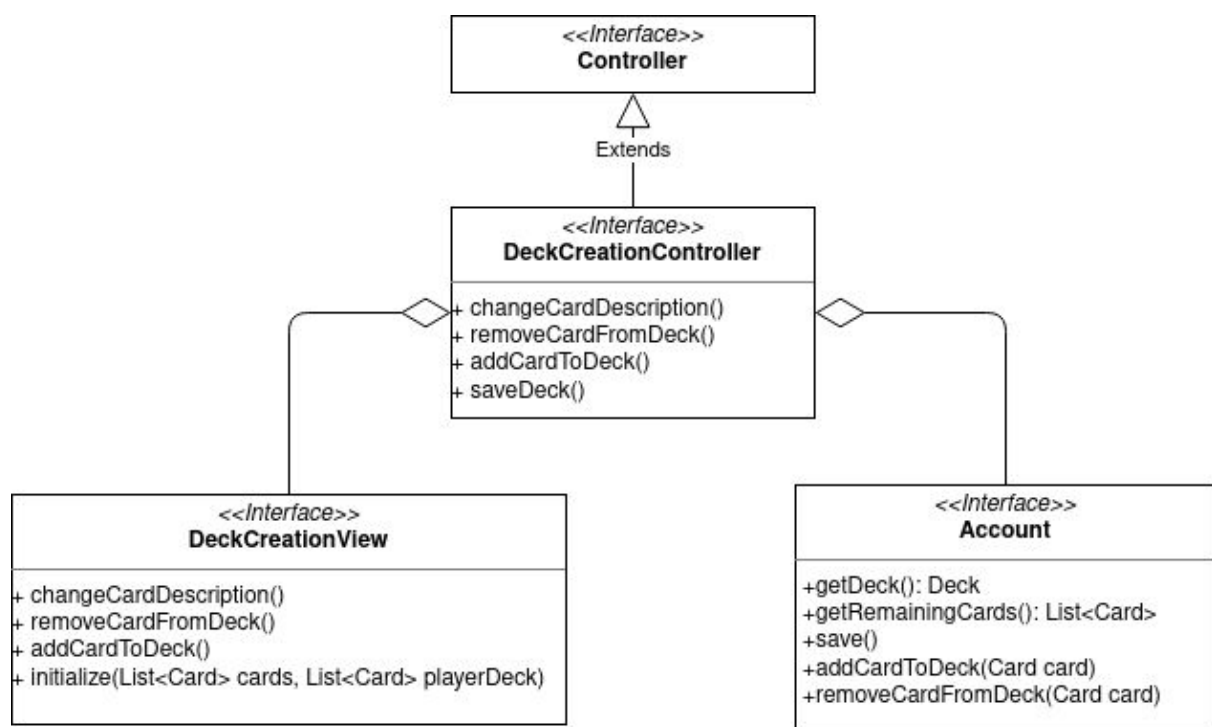


Figura 2.2 rappresentazione UML del pattern Strategy

Francesco Marcaccini

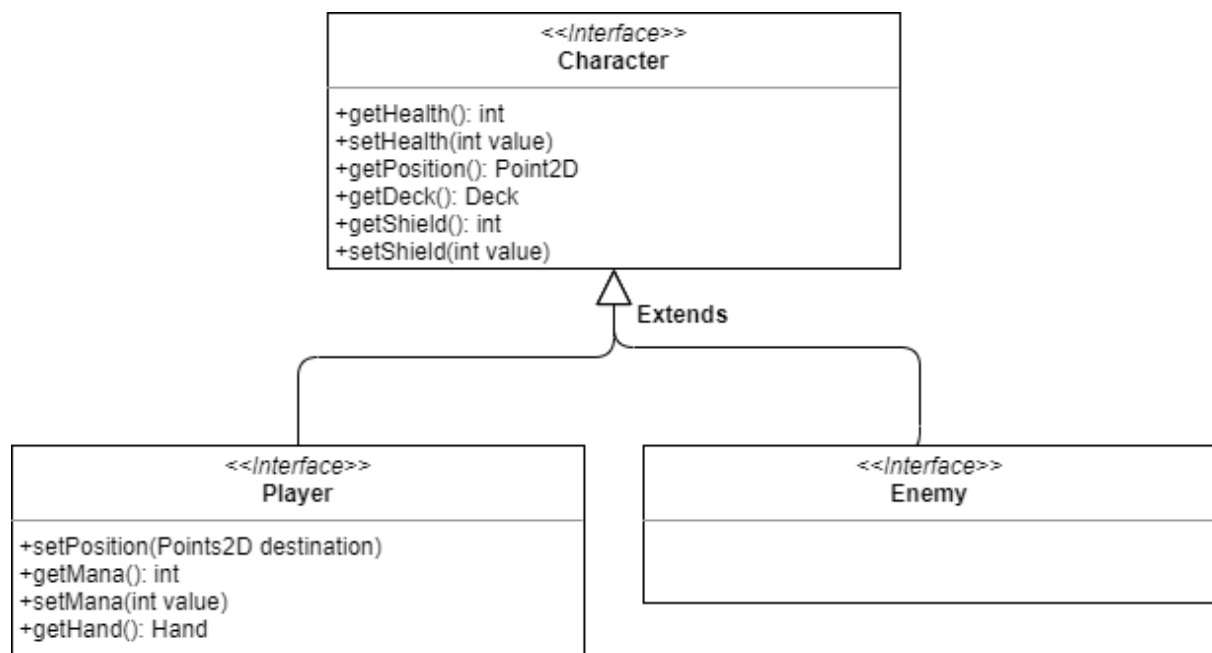
La parte da me gestita in questo progetto e' quella che riguarda:

- Character, Player e Enemy
- Account, Statistics e FileManager

Player, Enemy & Character

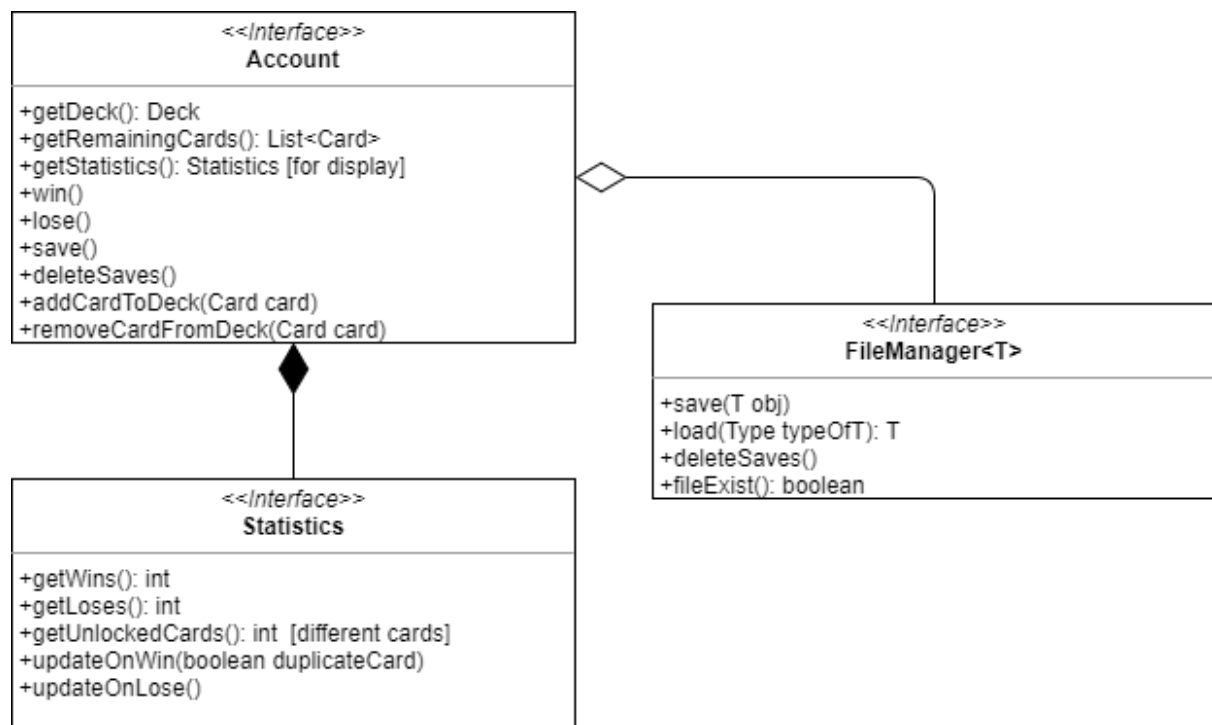
Questa prima parte riguarda il Player e i Nemici.

Uno dei problemi da risolvere e' quello di unire i metodi in comune che queste due entità hanno (es. `getHealth()`, `getShield()`, ...). Per questo ho deciso di creare una superclasse `Character`, la quale contiene tutti i metodi in comune, e dalla quale far poi estendere sia `Player` che `Enemy`, i quali avranno poi i propri metodi specifici, che non devono essere condivisi. Facendo questo in caso in futuro sia necessaria l'implementazione di nuovi tipi di personaggi basterà far estendere anche a questi `Character`, così da evitare di dover riscrivere sempre gli stessi metodi, evitando così errori, e nel caso il codice vada modificato basterà accedere alla superclasse una sola volta al posto di dover accedere a tutte le classi dei personaggi una per una, oltre ad una più elevata chiarezza sul fatto che queste classi abbiano aspetti in comune.



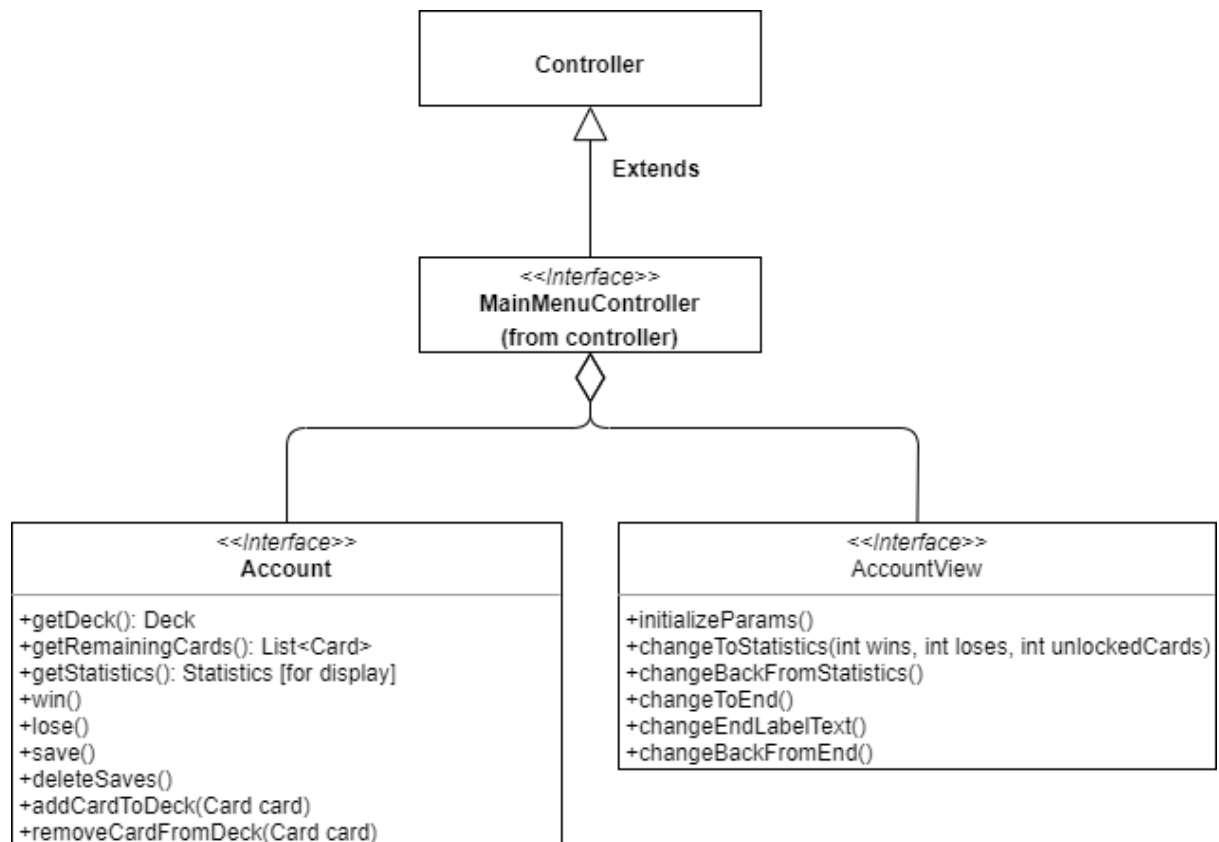
Account, Statistics & FileManager

Questa seconda parte invece parla di Account, Statistics e FileManager. In questa parte il problema principale da risolvere è il salvare su file diversi tipi di dato, senza però incorrere in ripetizioni e ovviamente errori. I vari tipi di dato che vanno salvati sono quelli delle statistiche, quelli del deck e quelli delle carte non nel deck. Per risolvere questo problema ho deciso di creare una classe FileManager di tipo anonimo che permette il salvataggio di dati su file a prescindere dal tipo. Creando questa classe è anche stata possibile l'aggiunta futura di nuovi tipi di dati salvabili, permettendo quindi l'implementazione di nuove feature senza il bisogno di riscrivere tutto il codice. Un altro aspetto è quello di aver distaccato in maniera più chiara due problemi, ossia la gestione dell'Account e la gestione dei File, rendendo così più intuibile lo scopo delle varie classi. Lo stesso principio vale anche per Statistics, il cui scopo è principalmente quello di separare le varie informazioni, per evitare di aggiungere troppe cose, anche non correlate, nella stessa classe.



Main Menu Controller

Per il controller viene utilizzato un pattern Strategy, infatti passando al controller sia la View che il Model, questo ci permette di cambiarle anche a runtime entrambe senza problemi. Questo essenzialmente vuol dire che il controller è completamente separato sia dalla View che dall'implementazione del Model.



Claudio Verazza

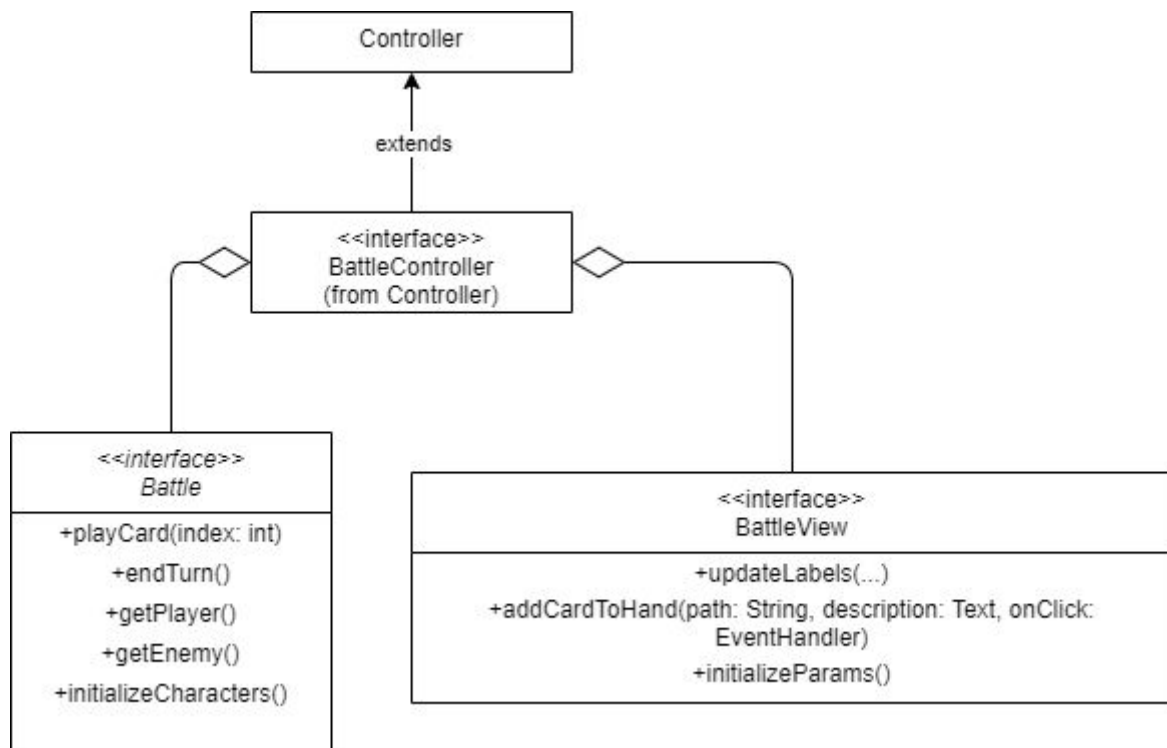
In questa sezione ci si concentrerà sulla creazione e l'elaborazione della fase della battaglia tra giocatore e nemico.

Siccome la radice della battaglia è il controller stesso (o meglio, il presenter), è stato adottato un pattern Strategy per l'implementazione del model e della view: entrambi sono incapsulati all'interno di BattleController. Sia model che view sono totalmente interscambiabili, senza che la logica del presenter debba essere modificata.

Il presenter è stato inteso come un oggetto il cui principale ruolo è il reindirizzamento delle azioni del giocatore sulla view: il suo compito principale, infatti, consiste nell'aggiornare la view con i dati del model, massimizzandone il riuso in quanto le operazioni eseguite da esso sono mantenute al minimo.

La sua seconda funzione è l'assegnamento dell'evento che corrisponde alla giocata di una carta. Per massimizzarne il riuso, viene passato un evento generico alla view che, in base alla sua implementazione, deciderà come farlo accadere.

Il presenter non ha alcuna informazione sullo stato della battaglia (ad esempio: di chi è il turno, chi sta cercando di giocare la carta, chi è il vincitore della battaglia, ecc.), ruolo che viene totalmente ricoperto dal model. Si è decisa questa suddivisione per delegare meno compiti possibili al presenter (escluso il reindirizzamento di informazioni) e poter gestire al meglio la logica della battaglia all'interno del model.

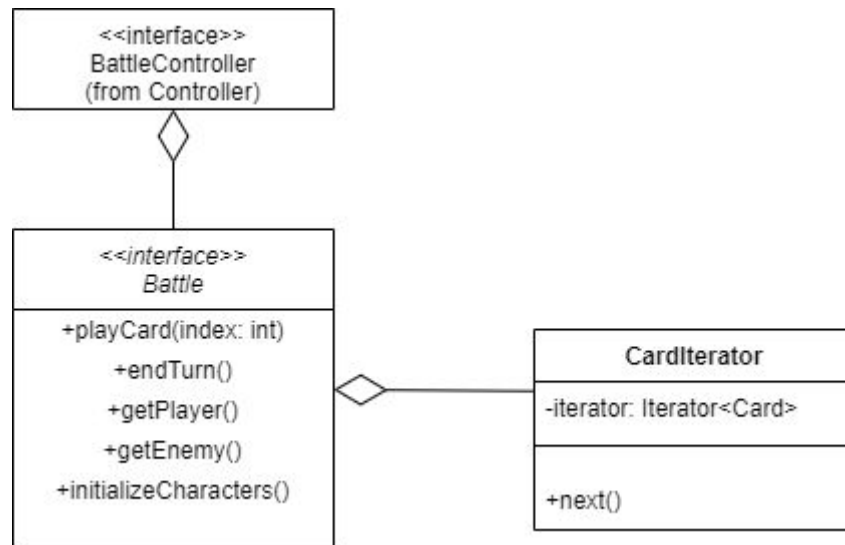


Nel model sono presenti tutti i metodi per interagire ed ottenere informazioni dai personaggi che stanno partecipando alla battaglia. È l'elemento che memorizza lo stato della battaglia, nonché ad eseguire tutte le azioni invocate dal giocatore sulla view.

Il metodo playCard serve per far giocare una carta ad un personaggio, e si è deciso di farle prendere in ingresso non la carta da giocare, ma un intero, rappresentante l'indice della carta nella lista di chi la sta giocando. Questa scelta è motivata dal fatto che, per dividere nettamente logica da view, quest'ultima non deve avere la definizione di carta. La soluzione più generica adottata è stata la presa in ingresso del suo indice della lista, ed in base al turno giocarla dalla lista corretta.

Per scorrere lungo i deck dei personaggi si è adottata una basilare implementazione del pattern IteratorPattern, che restituirà sempre una carta e, nel caso il deck fosse esaurito, viene automaticamente creato un nuovo iteratore dall'inizio del deck e restituita la prima carta. Per renderlo riutilizzabile, si è deciso di fargli prendere in ingresso una generica collezione di carte, piuttosto che una lista specifica. Il vantaggio è che, usando questo iteratore, si può lavorare sul deck del personaggio senza avere vincoli sulla sua effettiva implementazione. Nello schema seguente si è deciso di mostrare anche il campo privato Iterator per far

notare che l'iterazione non avviene tramite indice, ma tramite un altro iteratore interno, in quanto si è ritenuto più pulito farlo dato che la classe implementa Iterator.



La view non è altro che una componente passiva, che aggiorna la propria interfaccia con le informazioni passate dal controller. Non ha alcuna responsabilità particolare, se non l'assegnamento dell'evento da eseguire sulle carte. In questo caso l'evento è il click sull'immagine della carta ma, nel caso di cambiamenti futuri, dato che l'evento ottenuto in ingresso è generico, potrà essere riadattato in modo opportuno.

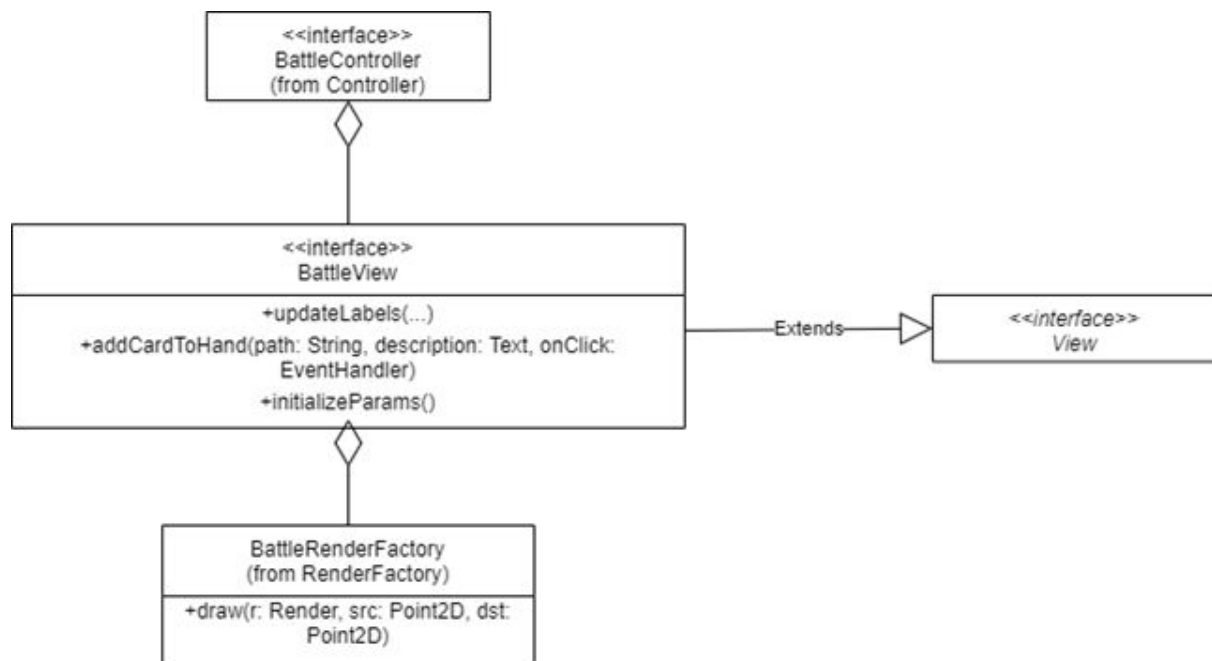
Per visualizzare le informazioni relative alla singola carta, si è deciso di modificare un campo testuale quando il puntatore del mouse è posizionato sulla stessa.

Dato che, al momento dell'istanziatura della view, lo stage non è ancora stato caricato, occorre impostare la scena ed eseguire un lookup dell'FXML per trovare i campi modificabili in un secondo momento, funzione ricoperta da `initializeCharacters`. Quest'ultima, infatti, viene eseguita dal controller quando lo stage è pronto per essere utilizzato.

Contiene tutti i metodi utilizzabili dal controller per aggiornare l'interfaccia. Per motivi di spazio e ripetizione, nello schema seguente tutti i metodi sono stati compressi in un generico `updateLabels`.

Per disegnare i simboli di giocatore e nemico sulla view, anziché utilizzare un'immagine, si è implementata una semplice factory

utilizzando un pattern AbstractFactory che, con l'aiuto di un Drawer, ne disegna i simboli. Questo è stato fatto per separare il più possibile il personaggio dalla sua rappresentazione grafica, e variare questa visualizzazione rispetto alle altre view. Inoltre, è possibile riutilizzarla in altri contesti futuri dove sia necessario disegnare i personaggi senza alterarne l'implementazione.



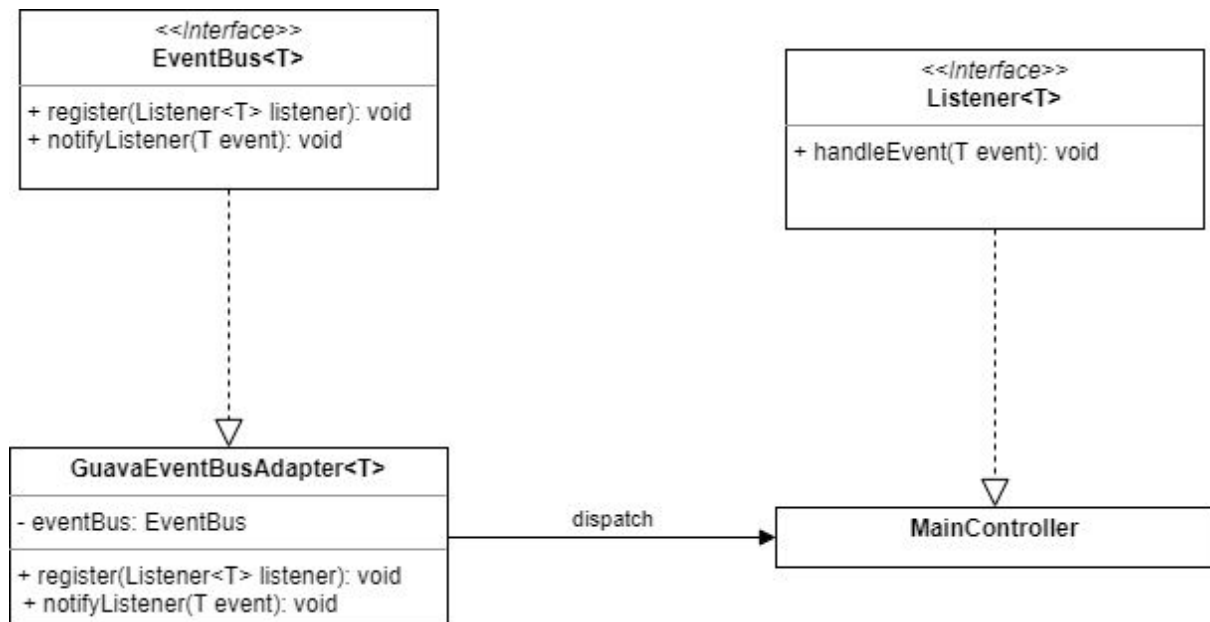
Stefano Guidi

Sistema di notifica

Il sistema di notifica è stato utilizzato all'interno del progetto corrente per informare il MainController(componente adibita a gestire lo switch di contesto tra controller) di un'eventuale richiesta da parte di un Controller.

Si è scelto di adottare un pattern **Observer** la cui implementazione è stata definita da 2 componenti: un notificatore(definito dall'interfaccia parametrizzata *EventBus<T>*) e da un listener(interfaccia *Listener<T>*); il notificatore è fornito in composizione ai controller mentre il listener deve essere implementato per con un handler per gestirne i messaggi(in questo caso il MainController implementa Listener). I ruoli individuabili sono pertanto quello di **Observer** per il MainController e di **Observables** per i Controller.

Per quanto riguarda l'implementazione di *Eventbus<T>* si è scelto di utilizzare l'implementazione di un EventBus di **Google Guava**, in particolare per facilitarne l'uso (e l'adattamento alla registrazione di soli *Listener<T>*) relativamente alle caratteristiche descritte precedentemente, se ne è svolta un'implementazione sfruttando il design pattern **Adapter** dove l'**Adaptee** è il Guava EventBus. (NOTA: Si è optato per l'utilizzo di tale pattern per alcune motivazioni relative all'API fornita da Google Guava ed il tentativo di aggirare problematiche relative alla **Type Erasure** di Java, si rimanda pertanto alla sezione personale del punto 3.3 sulle note di sviluppo relative all'uso di librerie di terze parti e costrutti avanzati).



Separazione entità logiche/visuali

Il design descritto di seguito è stato pensato per rispettare nel modo migliore possibile l'obiettivo delle architetture MVP.

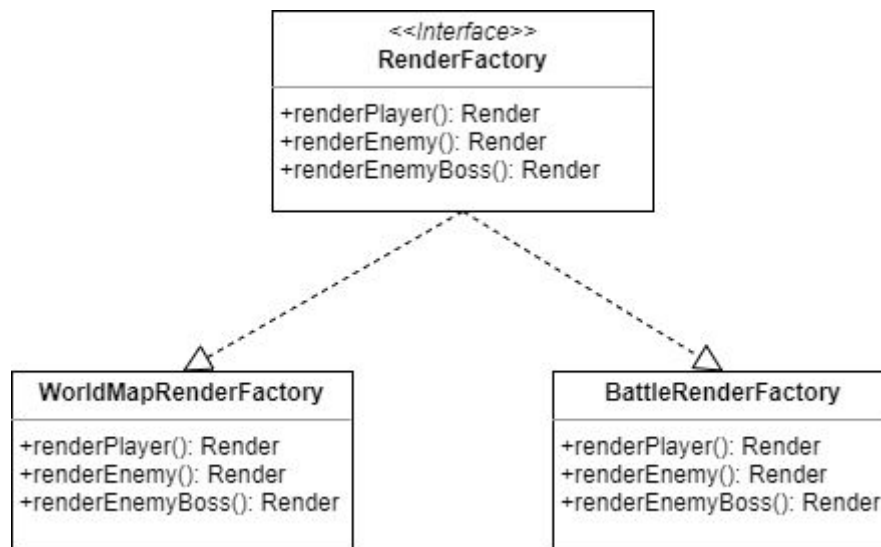
Si è voluto separare le entità logiche dalla loro visualizzazione da view, questo compito è svolto dal Controller(Presenter) che prende i dati del model e li "presenta" su view in un certo modo.

Per semplificare ciò si è creata un **Abstract Factory** che fornisce un oggetto **Render**(la cui creazione è facilitata dall'uso di un **Builder pattern**) relativo alle entità principali.

I vantaggi principali che questa scelta di design offre sono i seguenti:

- separazione entità logiche dalla loro logica di visualizzazione
- possibilità di implementare tale factory sia per la WorldMapView che per la BattleView e garantire il riuso.

(NOTA: nonostante il pattern sia stato definito personalmente, differenti membri hanno creato le proprie implementazioni di RenderFactory, si descrive la struttura completa delle implementazioni della factory a solo scopo di chiarezza).



Disegno sulla vista

Relativamente alla WorldMap si è scelto di visualizzare il player e gli enemy visualmente disegnandoli direttamente sul grafo di scena di JavaFx.

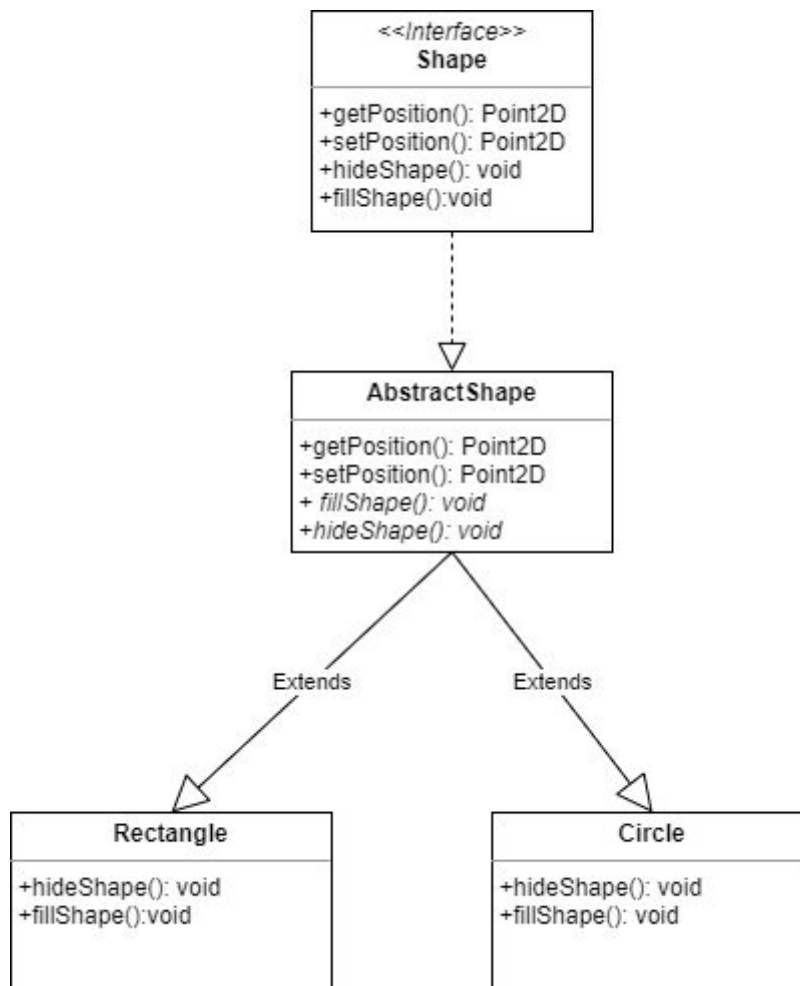
Per far ciò la View è dotata di un oggetto **Drawer** dove l'elemento parametrizzato deve essere un javafx.scene.Node, per fare in modo che l'oggetto su cui venga eseguita la visualizzazione(disegno) sia definibile in un grafo javaFx.

Questa interfaccia lavora sulla base di un oggetto **Render** e la sua posizione di destinazione relativa all'interno del **Node**; nell'implementazione CanvasDrawer(la quale wrappa un oggetto Canvas) si è scelto di svolgere il disegno su una canvas ma l'interfaccia è teoricamente estensibile ad un qualsiasi Node per poter applicare ad esso proprietà grafiche.

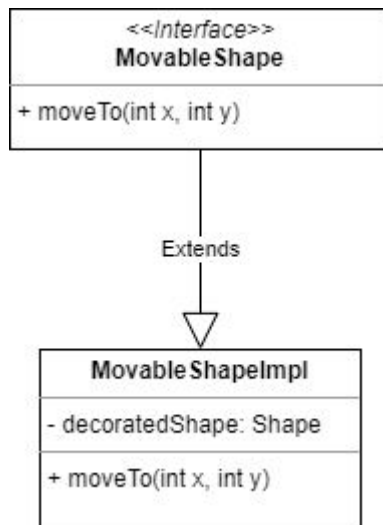
CanvasDrawer opera disegnando sulla Canvas degli oggetti definiti da un'interfaccia **Shape**, in particolare per fornire ad essi una maggiore estensibilità si è optato per l'uso di 2 design pattern

- **Template method:** è stato utilizzato per permettere l'interoperabilità di oggetti Shape a prescindere dalla loro

visualizzazione grafica, i template method utilizzati sono i metodi `fillShape` e `hideShape` della classe `AbstractShape`, sostanzialmente questi 2 metodi visualizzano/nascondono il contenuto grafico delle Shape, e, a differenza delle Render possono variare a seconda della forma utilizzata.



- **Decorator:** l'esigenza di questo pattern deriva dal fatto che alcune Shape hanno una responsabilità in più, cioè l'esigenza di spostarsi graficamente. Il player infatti potrà spostarsi mentre i nemici rimarranno sempre fermi. Pertanto la Shape del player viene definita come un `MovableShape`, un decorator di `Shape` che ne aggiunge la responsabilità di movimento, inoltre risulta un'operazione che riutilizza interamente le operazioni fornite da `Shape` concatenandole tra loro(in particolare `hideShape()` -> `setPosition()` -> `fillShape()`).



Integrazione Model/View in Controller

Nel costruttore del **WorldMapController** sono stati incluse in **dependency injection** un oggetto **WorldMap(model)** e **View(view)**, ciò descrive l'utilizzo di un pattern **Strategy**, nel quale la view ed il model possono variare a runtime previo il rispetto del contratto definito. In questo le implementazioni che ne rispettano i contratti sono **WorldMapImpl** per **WorldMap** e **WorldMapView** per **View**.

III Sviluppo

3.1 Testing automatizzato

Matteo Paolucci

Nei test mi sono concentrato sulla mia parte del model dal momento che era fondamentale per il progetto nel suo complesso. I test sono stati eseguiti tramite JUnit5 e il task (test) automatizzato di gradle

I test che ho effettuato sono:

- Aggiunta di carte nel mazzo, per verificare che effettivamente siano presenti nel mazzo
- Controllo di carte illegali: ho deciso di controllare la costruzione delle carte per far sì che non si abbiano problemi successivamente nello svolgimento del gioco
- Creazione del mazzo: il mazzo può essere creato in vari modi e mi sono assicurato che in ogni caso non ci siano problemi dopo la sua costruzione
- Controllo di mazzi illegali: ho verificato il fatto che il caricamento da file del deck funzionasse correttamente
- Controllo sul mazzo vuoto: ho voluto controllare il comportamento del mazzo quando il deck è vuoto per verificare il non ritorno di null

Francesco Marcaccini

Ho deciso di sottoporre a test automatizzato principalmente la classe di account, poiché era la più complessa, e con salvataggio su file.

In questa parte ho deciso di testare:

- La rimozione di una carta dal deck, per vedere se venissero effettivamente fatti i controlli necessari e se la carta fosse effettivamente aggiunta al deck e rimossa dalle carte rimanenti.
- L'aggiunta di una carta nel deck, che è essenzialmente simile a quella descritta precedentemente.
- Il salvataggio di nuove carte nel deck e nelle carte rimanenti, per vedere se queste carte venissero effettivamente salvate su file correttamente e poi caricate correttamente.
- La vittoria di una partita, per vedere se la logica di vittoria funzionasse correttamente, creando effettivamente una carta da aggiungere alle carte possedute e se le statistiche venissero effettivamente aggiornate e salvate.
- La perdita di una partita, essenzialmente simile a quella specificata precedentemente, ma per la logica di una sconfitta.

Ho anche effettuato dei test manuali:

- La rimozione dei file di default, per testare se il comportamento del programma fosse effettivamente quello desiderato, è stato necessario fare un test manuale qui' poiché non sarebbe stato possibile riottenere i file di default una volta cancellati, portando al non funzionamento di tutto il programma.

Claudio Verazza

Sono stati eseguiti dei test in JUnit 5 e Gradle per verificare il corretto funzionamento della battaglia, tutti completamente automatici. Si è testata la dimensione della mano, che non deve contenere più del numero consentito di carte nonostante i continui pescaggi, oltre che ad intrinsecamente testare che il deck non dia problemi se si arriva a pescare l'ultima carta. Importante è stato il test di inizializzazione della battaglia, che non può iniziare se prima tutti i relativi parametri e personaggi non siano stati inizializzati. Si è testato inoltre il corretto funzionamento delle carte, ovvero che applichino gli effetti corretti quando giocate.

Stefano Guidi

Nella porzione di test eseguita si è scelto di testare gli aspetti di movimento ed interazione relativi al model e la conversione da posizione logica a visuale relativa al controller, la motivazione principale di questa decisione è stato il fatto che queste features risultavano essere quelle più critiche per il corretto funzionamento della parte personale del software. Tra i test attuati:

- controllo corretta destinazione relativa ad una serie di movimenti
- controllo delle collisioni e della relativa interazione con esse se nemiche
- controllo della feature di movimento solo entro i limiti permessi della mappa
- controllo relativo alla corretta conversione tra posizione logica e posizione grafica visuale.

3.2 Metodologia di lavoro

Matteo Paolucci

Creazione del mazzo e delle carte. Creazione della view e del controller della costruzione del mazzo. Caricamento mazzo da file.

Francesco Marcaccini

Creazione dei personaggi di gioco, come il player e i nemici, creazione dell'account con i propri dati, salvataggio dei dati su file e caricamento di questi dati, creazione del menù principale.

Claudio Verazza

Implementazione del controller, model e view della battaglia. Creazione dell'interfaccia grafica della battaglia con metodi per giocare le carte. Creazione dell'iteratore sul deck del giocatore e nemico durante la battaglia.

Stefano Guidi

Creazione della mappa di gioco logica, creazione vista mappa con movimento e interazioni, integrazione di queste all'interno del Controller, sistema di notifica e MainController.

Lavoro di integrazione

Il lavoro di integrazione è stato facilitato dall'uso della libreria Google Guice che, oltre a semplificare l'iniezione delle dipendenze utilizzate, ha fornito al gruppo uno strumento per lavorare in maniera indipendente e modulare(ogni membro utilizzava inizialmente solo i propri moduli). In particolare il lavoro è stato gestito inizialmente sullo sviluppo indipendente dei membri del gruppo relativamente alle proprie feature assegnate e, solo in seguito, su una finale integrazione.

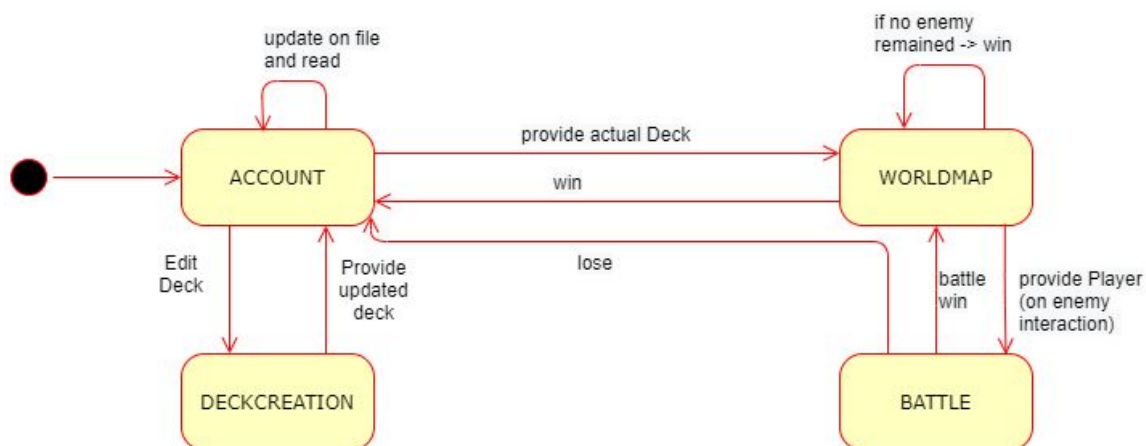


Figura: le feature principali sono relativamente indipendenti nelle loro funzioni

I primi lavori di integrazione sono stati svolti tra le funzionalità relative alla gestione dell'Account utente e quelle di generazione e modifica del deck in quanto esse erano le feature con più necessità di lavoro condiviso.

Le funzionalità di notifica (scambio messaggi tra MainController e controller) sono state sviluppate nelle prime parti di sviluppo, ma a solo scopo di corretto controllo del loro funzionamento (sono state integrate infatti nelle ultime fasi di lavoro).

La mappa di gioco e la battaglia sono state sviluppate per buona parte dello sviluppo in maniera indipendente (utilizzando delle variabili dummy per l'utilizzo del deck e del giocatore); sono stati prima integrati tra di loro (aggiungendo le funzionalità per scambiare l'oggetto player tra le 2 funzionalità), e, solo alla fine sono stati integrati con le parti rimanenti di progetto.

Uso del DVCS

L'uso del DVCS(Git) è stato organizzato per fare in modo che ogni utente possa lavorare su una propria branch personale. Il motivo che ha portato il gruppo su questa semplificazione nell'uso del DVCS(rispetto ad usare una serie di feature branch temporanee) è il fatto che è stato ritenuto che le funzionalità di lavoro fossero sufficientemente suddivise da permettere una tale indipendenza nell'uso.

NOTA SULL'USO DI GOOGLE GUICE

(Si è scelto di includere questa nota nella sezione corrente, siccome l'uso di Guice ha inciso un ruolo importante nell'architettura del sistema e sulle metodologie di lavoro. Per la specificità di uso è stata deciso di includerla in questa sezione).

Si è scelto di integrare la seguente suite: JavaFX + FXML + Guice per la gestione dei Model,Controller e View.

FXML di default ricerca e istanzia i controller associati ai layout FXML sulla base del classpath definito nel file, è possibile però fornire una factory di controller che viene richiamata in maniera preliminare per poter istanziare subito quest'ultimo.

In questo caso la factory utilizzata è una callback fornita dall'istanza del Guice injector creata(in particolare injector.getInstance(CLASS)). Per semplificare l'utilizzo si è creata una classe **SceneManager**, la quale, oltre ad incapsulare la logica di invocazione tra FXML e i moduli Guice, permette di accedere ad una determinata triade MVP tramite uno specifico layout da invocare.

Il vantaggio di ciò è che ogni oggetto SceneManager mantiene in memoria sia un oggetto scena che l'oggetto FXMLLoader correlato, evitando quindi di dover invocare ogni volta i moduli Guice(operazione più lenta in quando Guice si serve dell'uso di reflection).

3.3 Note di sviluppo

Paolucci Matteo

- *Stream*: utilizzata per manipolare le carte e componenti di JavaFx in maniera rapida, pulita ed efficace.
- *JavaFx*: utilizzata per creare e gestire la view
- *Gson*: utilizzata per caricare il mazzo da file
- *Guice*: utilizzata per iniettare dipendenze
- *Optional*: utilizzati per le carte
- *Lambdas Expressions*: utilizzate con gli Stream ed i test

Ho iniziato lo sviluppo a partire dal model, dove sono finito ad utilizzare la libreria Gson per de-serializzare le carte da file jsons.

Nello sviluppare la view ho utilizzato sia javafx che gli stream. JavaFx mi ha dato gli elementi di base per costruire la view, mentre gli stream mi sono stati molto utili per manipolare alcuni elementi della view e le carte. I generics mi sono stati utili per implementare il pattern Builder di Joshua Bloch che permette di creare dei builder estendibili per ogni classe che estende AbstractCard.

Uso della libreria Guice per iniettare le dipendenze nel controller del model e della view.

Per non ritornare dei null nelle carte e nel deck ho preferito utilizzare degli optional che migliorano anche la leggibilità del codice.

Dove ho utilizzato gli Stream, ho utilizzato anche le Lambda Expressions per migliorare la compattezza del codice e anche la sua funzionalità.

Francesco Marcaccini

Nella mia parte di lavoro ho fatto uso di generics, li ho usati per la classe FileManager, e ho deciso di adoperarli perché avevo la necessità di gestire il salvataggio e il caricamento su/da file di dati di tipo diverso. Ho anche fatto uso di una libreria esterna chiamata Gson, per il salvataggio e il caricamento su/da file di dati in formato json. Ho utilizzato la libreria JavaFx per la gestione e creazione delle interfacce e quindi i loro file fxml. Ho utilizzato internet principalmente solo per la documentazione di Gson e la JavaDoc, principalmente per lavorare con i File. Uso della libreria Guice per l'iniezione delle dipendenze del model, view e controller.

Claudio Verazza

È stato fatto un basilare utilizzo degli Optional quando il giocatore cerca di pescare una carta per capire se la carta estratta può essere aggiunta alla mano. Infatti, se la carta pescata è già nella mano, non può essere aggiunta alla stessa. Per evitare l'utilizzo di valori nulli durante il pescaggio, si è preferito utilizzare un costrutto più adatto per il compito. Per l'interfaccia grafica della battaglia è stato usato JavaFX in quanto è parso più adatto alle necessità di avere un'interfaccia semplice e snella, oltre che per adattarsi alle altre esigenze del progetto. Per rendere in alcune parti il codice più elegante, sono stati utilizzati frammenti di Google Guava, in particolare della classe Iterables, per ottenere una maggiore leggibilità. Utilizzo della libreria Guice per l'iniezione delle dipendenze nel presenter e nel modulo della battaglia.

Stefano Guidi

Uso della libreria Google Guava

L'uso personale relativo alla libreria di terze parti **Google Guava** è stato l'utilizzo dell'implementazione EventBus per la gestione messaggistica Publisher/Subscriber.

Le motivazioni che hanno portato ad utilizzare tale implementazione sono principalmente 2:

- i riceventi definiscono i loro metodi di handler tramite la sola annotazione **@Subscribe** rendendo il metodo di destinazione più leggibile rispetto ad altre librerie che supportano ciò tramite l'implementazione di interfacce di terze parti.
- Per il motivo precedente è teoricamente possibile definire più metodi di handler annotati; inoltre è possibile discriminare un messaggio sulla base dell'argomento che un metodo annotato si aspetta, definendo strategie di comunicazione più avanzate.

Come introdotto nella parte di **2.2 Design dettagliato** vi sono state delle motivazioni che hanno portato alla necessità di adattare la classe EventBus:

- L'API di Guava EventBus accetta un **Object** sia per il **register** che per il **notify**, pertanto l'Adapter restringe la registrazione al tipo Listener<T> ed il notify al solo generic T.
- La Type erasure di Java produce il problema di considerare a runtime i tipi Listener<T> -> Listener<Object> e <T> -> Object, pertanto 2 listener registrati come Listener<String> e Listener<Integer> potrebbero ricevere gli stessi messaggi rischiando di trattarli in maniera impropria. Per aggirare(in parte) ciò si è scelto pertanto di utilizzare l'Adapter creando un coupling solo tra EventBus<T> e Listener<T> ed evitando errori a runtime.

Uso della libreria Google Guice

Ci si è serviti (come gli altri membri) della libreria Google Guice (come descritto nella sezione precedente) per modularizzare il lavoro e facilitare l'iniezione delle dipendenze. Si è svolto il modulo per il WorldMapController e per la comunicazione.

Nella gestione del modulo di comunicazione si è scelto di trattare la dipendenza di MainController con l'annotazione @Singleton, si è fatto ciò per evitare che i vari controller ottenessero un'istanza diversa di MainController, questa scelta è stata fatta non per accentuare il ruolo di Singleton per l'oggetto, ma per evitare che in ogni controller venga allocata una nuova istanza senza alcun reale bisogno di ciò.

Cenni di sviluppo su generics/bounded generics/wildcards

- **Drawer<T extends Node>**: utilizzato per limitare l'interfaccia Drawer ai soli JavaFx Node, da una parte perde la possibilità di estensione ad altri framework grafici ma dall'altra restringe le implementazioni ad elementi su cui è possibile l'operazione.
- **Request<T, R>**: parametrizza una richiesta con T->direzione destinatario ed R->dati forniti con la richiesta.
- **Request<LAYOUT, ? extends Object>**: dichiarazione per il tipo di dati fornibili al MainController, utilizza come routing il layout e permette di fornire come dati un qualsiasi oggetto che estende Object.
- **EventBus<Request<LAYOUT, ? extends Object>>**: l'oggetto EventBus viene parametrizzato con l'oggetto precedentemente descritto permettendo di notificare soli Listener<Request<LAYOUT, ? extends Object>>.

Cenni di sviluppo su stream

Sono stati utilizzati in maniera intensiva nella parte di lavoro, alcuni esempi:

- **Stream per la rilevazione di nemici adiacenti**
(WorldMapImpl.getEnemyifAdjacent): genera le coordinate

circostanti a quella fornita in ingresso, in seguito controlla se in una di queste vi è un nemico, nel caso se ne rilevi uno si ritorna il primo nemico individuato.

- **Stream per la conversione da enemies logici a visuali(`WorldMapController.initializeView`):** questo stream prende la lista di enemy, li mappa nelle loro posizioni logiche, poi tramite il valore di `TransformPosition` le mappa nelle loro posizioni grafiche. Infine si applica una `reduction collect` ad una lista la quale viene consumata aggiornando sulla view.

NOTA IMPORTANTE:

L'unico aspetto riutilizzato da codice non personale è stato un frammento di Stream che, essendo stato ritenuto elegante nella sua logica è stato riutilizzato a partire da uno snippet fornito come soluzione nel compito del a02a.e2 del 2019 di OOP.

Il frammento in questione applicava una rotazione di coordinate di 90 o 180 gradi per generare delle nuove coordinate tramite un mapping stream partendo da punti specifici, è stato leggermente modificato per generare le posizioni dei nemici in forma di croce.

Cenni di sviluppo su lambda/interfacce funzionali

Utilizzare in 2 porzioni di software:

- `TransformPositionFunction<Point2D,Point2D>`: interfaccia funzionale utilizzata per fornire un mapping tra posizioni logiche/visuali. Si è scelto di definire una nuova interfaccia funzionale invece di utilizzare una `Function` o un `UnaryOperator` per aumentare la chiarezza relativa al suo uso.
- `MOVEMENT`: enum utilizzato per incapsulare le strategie di movimento, per ognuna delle 4 direzioni è definita un `UnaryOperator` applicato sulla posizione corrente.

IV Commenti finali

4.1 Autovalutazione e lavori futuri

Paolucci Matteo

Il mio ruolo all'interno di pogeshi è stato per quanto fondamentale, non troppo difficile. Premetto che prima di affrontare questo progetto non avevo conoscenze pregresse sia sulla libreria JavaFx che sullo strumento di automazione dello sviluppo software Gradle. A livello di progettazione penso di aver fatto poco, ma bene. In particolare il lavoro sulle carte, in cui pur non avendo re-inventato la ruota, penso di aver saputo applicare correttamente i Design Patterns utili a migliorare sia la leggibilità che l'incapsulazione del software. Lo sviluppo in Linux è stato essenziale per la portabilità del codice, poiché ho individuato vari errori che hanno permesso a tutto il team di migliorare il software dal punto di vista del supporto multiplatforma. La coordinazione con i miei compagni di lavoro è stata buona e molto utile, in quanto mi ha dato uno specchio del com'è lavorare in un team di sviluppo software e di quanto possa essere bello e allo stesso tempo difficile collaborare in un team di persone con idee e modi di pensare diversi.

Francesco Marcaccini

Penso che i punti di forza della mia parte siano il salvataggio e caricamento su file dei vari dati, penso di aver gestito in modo abbastanza accettabile la classe generica e le varie eccezioni. Mentre i punti più deboli penso siano quelli che riguardano il Character, infatti penso avrei potuto farlo in modo migliore, magari con un pattern. Per quanto riguarda la mia parte di lavoro all'interno del gruppo penso sinceramente di aver suddiviso male i compiti, poiché mentre lavoravamo mi è sembrato che alcune persone avessero meno lavoro di altre, o meglio, avessero parti più semplici di altri. Ad esempio penso che la mia parte fosse più semplice di quella degli altri, nonostante avessi molte classi su cui lavorare, alla fine erano molto banali e non avevano molto di complicato, infatti l'unica cosa che ritengo mi abbia portato più challenge è quella riguardante la libreria esterna Gson, per lavorare su File. Penso quindi che sarebbe stato meglio se avessi valutato meglio a priori una suddivisione più corretta.

Claudio Verazza

Tutto sommato, nonostante sia consapevole delle imperfezioni nella stesura del programma, posso dirmi moderatamente soddisfatto del risultato finale: sono certo di avere imparato molto, e molto ho da imparare per padroneggiare il linguaggio. Per quanto la mia parte non sia stata particolarmente corposa, ho comunque speso molto tempo a revisionare e migliorare il mio lavoro, e, alla fine, non penso sia stato affatto facile, dato che la mia è la parte che maggiormente integra le altre. Penso che il mio punto di forza sia stata la netta separazione tra presenter e model, gradino su cui ho prestato particolare attenzione, oltre che a creare dei metodi abbastanza generali da poter essere utilizzati senza doverli modificare nel caso di cambiamenti esterni. Il punto di debolezza penso sia stato il moderato utilizzo delle feature avanzate del linguaggio: per quanto potenti esse siano, non sono riuscito ad integrarle in un modo particolarmente efficiente nella mia parte, anche se sono consapevole che possano essere utilizzate.

Stefano Guidi

La mia parte di lavoro consisteva nella gestione della mappa di gioco e delle funzionalità di notifica col MainController. In termini di risultato mi ritengo parzialmente insoddisfatto del risultato del mio lavoro: in particolare ho speso una buona parte di esso a cercare soluzioni corrette ed estensibili per le implementazioni, finendo per complicare il problema molto più di quanto fosse necessario e, ottenendo di conseguenza soluzioni non proprio ottime. D'altra parte mi ritengo soddisfatto di alcune parti specifiche del mio lavoro e di come sia riuscito ad utilizzare i costrutti avanzati di java ed alcuni design pattern in maniera tutto sommato accettabile. Oltre alle difficoltà di sviluppo precedentemente espresse ho avuto problemi con l'integrazione del lavoro la quale si è rivelata non proprio semplice(nonostante gli sforzi sin dalle prime fasi di lavoro per evitare ciò). Ritengo, però, che avere conoscenze pregresse nello sviluppo con Guice mi abbia aiutato a risparmiare tempo sul mio sviluppo personale consentendomi, almeno su quell'aspetto, di procedere senza particolari problemi.

4.2 Difficoltà incontrate e commenti per docenti

Paolucci Matteo

Ho avuto svariate difficoltà nello sviluppare la mia parte di lavoro al progetto. I miei problemi più grossi sono stati due da un lato non avevo la minima idea di come utilizzare Gradle e dall'altro sono stato l'unico a sviluppare su Linux.

Il problema di Gradle, e il motivo per cui ho dovuto perderci svariate ore, è stato il fatto che ai miei compagni le dipendenze di Gradle scaricate da Gradle non davano problemi, mentre io ho dovuto perdere svariato tempo per capire quali erano i problemi che colpivano me e non i miei compagni.

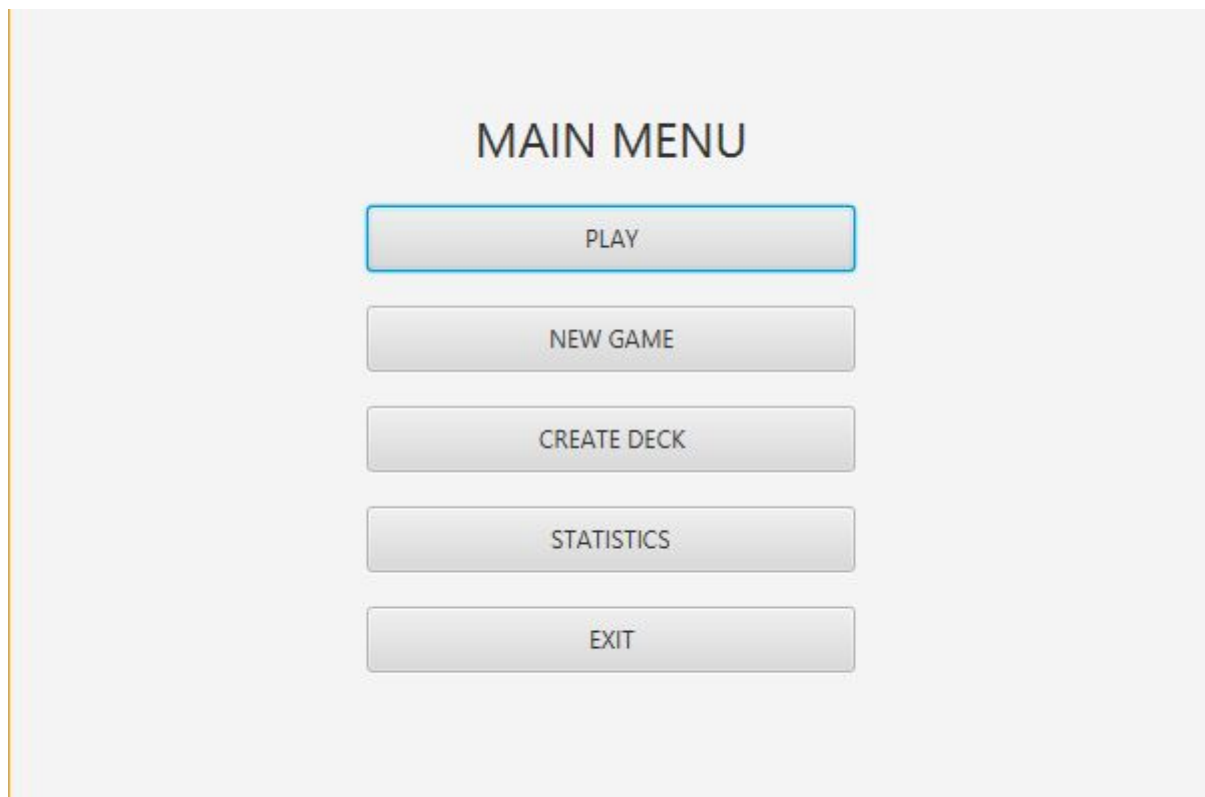
Il problema di Linux che è stato sia un PRO che un CONTRO, poiché pur avendo migliorato il progetto a livello di supporto multiplatforma, mi ha messo a confronto con vari problemi che mi sono dovuto risolvere da solo in quanto unico sviluppatore con sistema operativo Linux.

Io e Francesco abbiamo avuto nella fase iniziale del progetto qualche difficoltà nel coordinare il nostro lavoro, in quanto molte parti del nostro codice erano fuse l'una con l'altra.

Appendice A


Guida utente

Main Menu



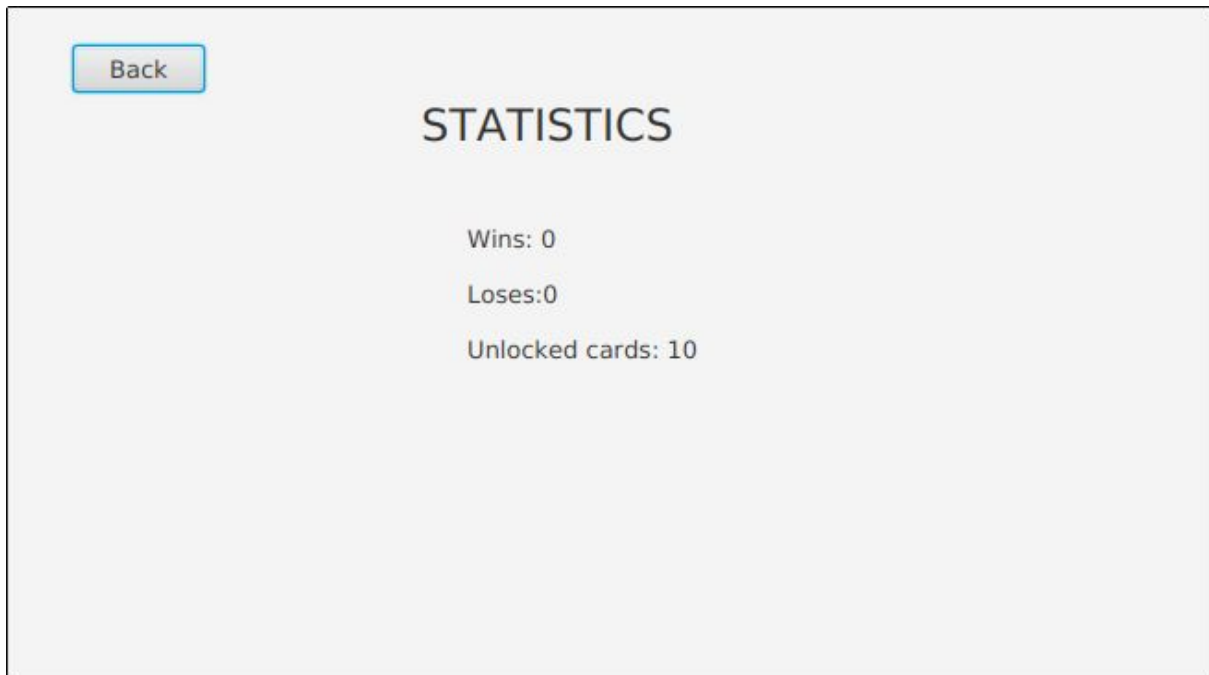
- Play: Incomincia la partita con il deck corrente.
- New Game: Resetta tutti i salvataggi e comincia la partita con il deck di default.
- Create Deck: Porta ad una schermata di creazione del proprio mazzo, in cui e' possibile modificare le carte presenti nel proprio mazzo con quelle sbloccate.
- Statistics: Porta ad una schermata in cui e' possibile visualizzare le proprie statistiche riguardanti partite precedenti.
- Esci: Chiude l'applicazione.

Deck Creation

Deck cards	10/10	Card description	Owned cards
<div> <div>Card1</div> <div>Card2</div> <div>Card3</div> <div>Card4</div> <div>Card5</div> <div>Card6</div> <div>Card9</div> <div>Card10</div> <div>Card2</div> <div>Card7</div> </div>		<div>  <div>Card2</div> <div>DESC</div> <div> <div>5</div> <div>3</div> <div>2</div> </div> </div>	<div> <div>Card3</div> <div>Card9</div> <div>Card10</div> <div>Card8</div> </div>
Remove		Save	Add

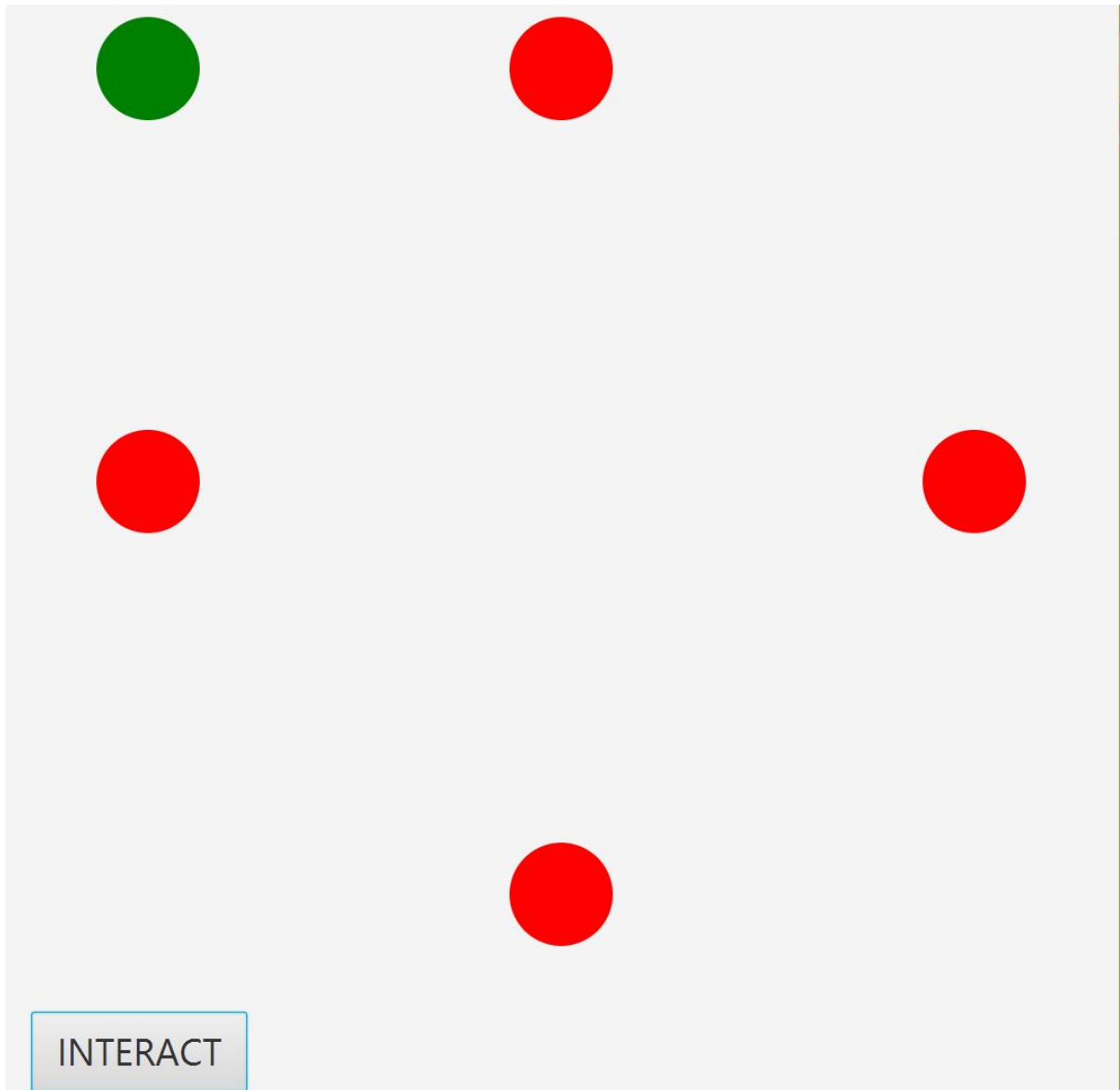
- Selezionare la carta del deck che si vuole rimuovere, e poi premere Remove per toglierla dal proprio deck.
- Se il deck non e' pieno il deck non può essere salvato, e quindi non potrà uscire dalla pagina finché il numero di carte nel deck non e' uguale a quello riportato in alto a sinistra (es. 10/10).
- Per aggiungere una carta al deck (il deck non deve essere pieno), bisogna selezionare una carta fra le owned cards che si vuole aggiungere, e poi premere add.
- Card description: Vi e' il nome della carta, una immagine, la sua descrizione e le sue statistiche. Il numero rosso rappresenta il danno che la carta farà al nemico; il numero bianco rappresenta il costo in mana per giocare la carta; il numero azzurro rappresenta la quantità di scudo che la carta darà al giocatore.

Statistics



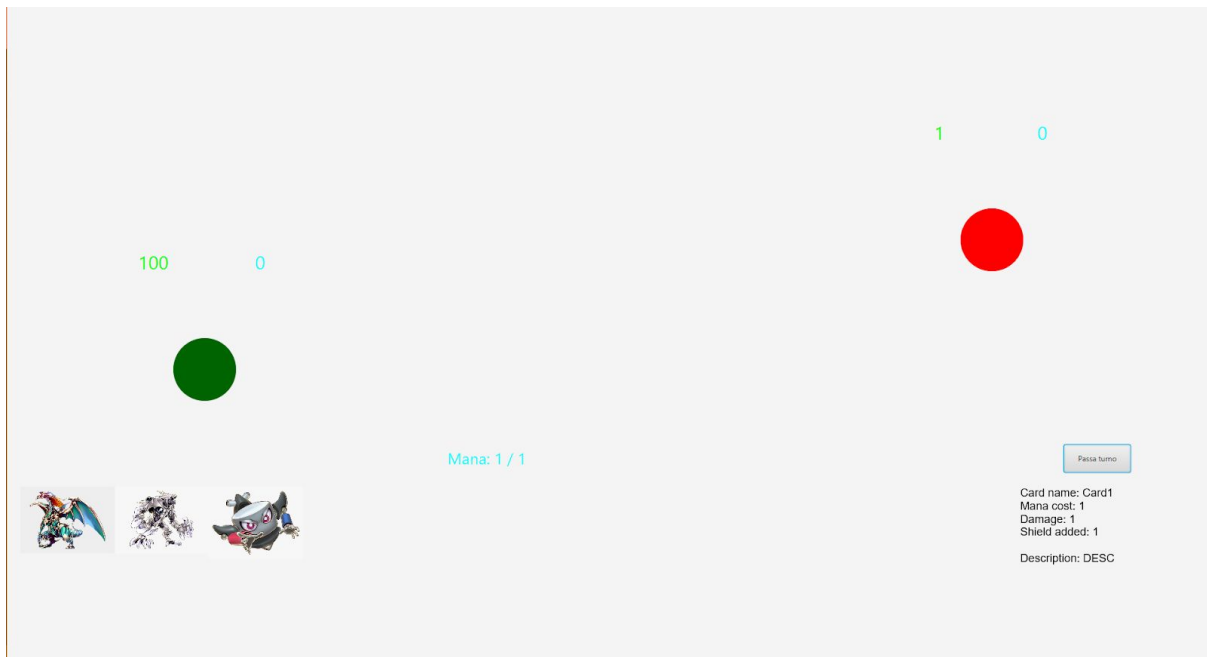
- Wins: il numero di vittorie
- Loses: il numero di sconfitte
- Unlocked Cards: il numero di carte uniche (diverse da quelle già possedute) sbloccate.

Game Map



- Il cerchio verde raffigura il player, ossia la persona che sta giocando.
- I cerchi rossi raffigurano i nemici.
- Il cerchio blu (che qui non si vede) raffigura il boss, per combatterlo vanno sconfitti prima tutti i nemici presenti.
- Per muoversi il player deve premere:
 - w : verso l'alto.
 - a : verso sinistra.
 - s : verso il basso.
 - d : verso destra.
- Una volta che il player raggiunge un nemico e vi è a fianco può premere il tasto Interact per entrare nella fase di combattimento.

Battle



- Durante questa fase il player può scegliere che carte vuole usare, tra quelle che ha pescato (massimo di 5 carte in mano), e premendoci sopra le giocherà, ricevendo i vari effetti in cui la carta consiste (es. farà 1 di danno al nemico, si scuderà di 1 e utilizzerà 1 di mana)
- Mettendo il cursore sulle varie carte sarà possibile vedere le statistiche di quella carta sulla destra, ossia il suo nome, costo in mana, danno, scudo, e la sua descrizione.
- Una volta che il giocatore avrà giocato le carte che vuole sarà necessario passare, una volta premuto il bottone il nemico farà la propria mossa e poi sarà nuovamente il turno del giocatore.
- Alla fine di ogni turno il Mana Massimo aumenterà di 1 sino ad un massimo di 10, e il mana corrente verrà riempito nuovamente.
- Sopra al giocatore e al nemico vi sono due valori: Il valore verde raffigura la sua vita, mentre il valore azzurro raffigura il suo scudo.
- La vita del giocatore rimarrà la stessa fra i vari combattimenti che verranno fatti (es. il giocatore perde 10 di vita nella prima battaglia, quando entrerà nella seconda avrà ancora 10 di vita in meno), verrà resettata solamente a fine partita.
- Lo scudo del giocatore verrà resettato alla fine di ogni battaglia, a differenza della vita non verrà quindi portato nelle varie battaglie successive.

Bibliografia

1. *Effective Java* by Joshua Bloch. Released December 2017.
Publisher(s): Addison-Wesley Professional.
ISBN: 9780134686097.
2. *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Released October 1994. ISBN: 887192150X.
3. *MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java:*
<http://www.wildcrest.com/Potel/Portfolio/mvp.pdf>
4. Google Guice repository: <https://github.com/google/guice>
5. Google Guava repository: <https://github.com/google/guava>
6. Google Gson repository: <https://github.com/google/gson>
7. JavaFX repository: <https://github.com/openjdk/jfx>