Indice

Capitolo 1

Analisi

1.1 Requisiti

Il software, realizzato come elaborato del corso di "Programmazione ad Oggetti" del corso di laurea in Ingegneria e Scienze Informatiche dell'università di Bologna, anno accademico 2020-2021, si pone come obiettivo la realizzazione di una copia del gioco di carte "Bang!", creato da Emiliano Sciarra e pubblicato dalla *daVinci Editore*.

La nostra versione del gioco presenta delle variazioni rispetto alla versione originale: ad esempio, non tutte le carte della versione base sono presenti. Sono inoltre assenti le carte introdotte dalle espansioni del gioco. Infine, alcune meccaniche di gioco sono state lievemente modificate al fine di ridurre la complessità del software.

Per vincere, i giocatori devono raggiungere un obiettivo specifico; che dipende dal ruolo assegnatogli.

Requisiti funzionali

- Come da regolamento, è possibile effettuare partite tra 4, 5, 6 o 7 giocatori. Il gioco avviene in locale e, per evitare di leggere le carte avversarie, i giocatori possono visualizzare lo schermo solo durante il proprio turno.
- Ad inizio partita ad ogni giocatore viene assegnato un ruolo, che determina l'obiettivo del giocatore stesso.
- Ogni giocatore, a turno, potrà giocare un qualsiasi numero di carte, e decidere in qualsiasi momento di passare il turno al giocatore successivo.
- Deve essere possibile eseguire un salvataggio dello stato della partita corrente, in modo che tale partita possa essere continuata in seguito qualora il gioco venisse interrotto.
- I giocatori devono essere in grado di consultare il regolamento del gioco prima dell'inizio di una partita.

Requisiti non funzionali

- Realizzazione di un'interfaccia grafica semplice ma efficace.
- Generale fluidità di gioco.

1.2 Analisi e modello del dominio

Il software dovrà contenere un tavolo di gioco (Table) nel quale saranno memorizzate le informazioni relative ai giocatori (Player), alle carte (Card) che questi hanno in mano o in gioco e al mazzo (Deck) da cui pescano le carte.
Ogni giocatore ha un ruolo (Role).
Le carte dovranno essere di due tipi diversi, identificate dai colori (Color) marrone e blu. Ogni carta ha un effetto (Effect) diverso, e gli effetti andranno divisi in base ai cambiamenti che provocano.
All'inizio del suo turno, ogni giocatore deve pescare due carte dal mazzo, e quando le carte del mazzo finiscono deve essere possibile costruire un nuovo mazzo composto dalle carte giocate o scartate da tutti i giocatori fino a quel momento.

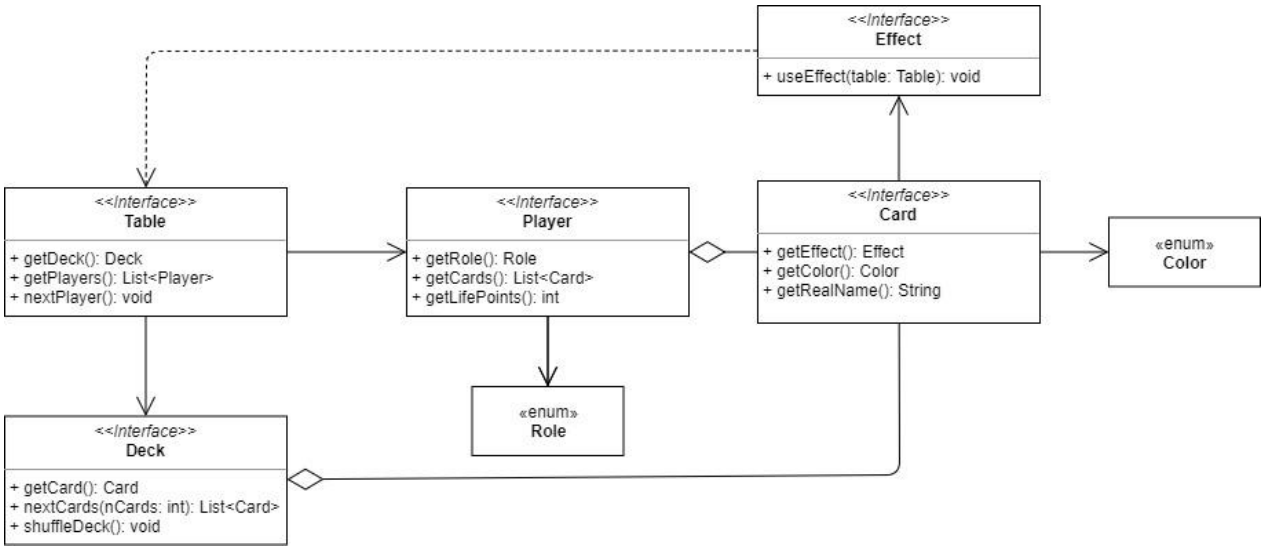


Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro.

Capitolo 2

Design

2.1 Architettura

L'architettura pensata per "BANG!" rispetta il pattern architetturale MVC, al fine di separare nettamente la parte di logica dell'applicazione dall'interfaccia utente. L'indipendenza tra le componenti di view e quelle relative al model fa sì che in futuro sia possibile ridurre al minimo il tempo necessario per apportare cambiamenti all'interno del codice. Al fine di abilitare la comunicazione fra i tre componenti, si è scelto di utilizzare il pattern *Observer*.

Il controller interagisce col model tramite la classe GameStateMachine, che utilizza lo *State pattern* per gestire le varie fasi di una partita. L'utilizzo di questo pattern si rende necessario a causa della particolare complessità delle regole che definiscono l'andamento del gioco.

GameStateMachine contiene inoltre un'istanza dell'interfaccia Table. Tale interfaccia agisce principalmente da "contenitore" dei vari elementi di gioco (giocatori, mazzo, etc…) e di dati utili per le interazioni tra il giocatore corrente e altri elementi di gioco; come ad es. quale giocatore è stato scelto come bersaglio di un attacco o quali carte sono già state giocate in un turno (in alcuni casi è necessario saperlo).

La view utilizza un *factory method pattern* per le varie schermate (menu principale, schermata delle regole, schermata di gioco, schermata di game-over). Per aggiornare la singola view e per notificare al controller di cambiare schermata quando necessario, viene utilizzato in modo estensivo il pattern Observer.

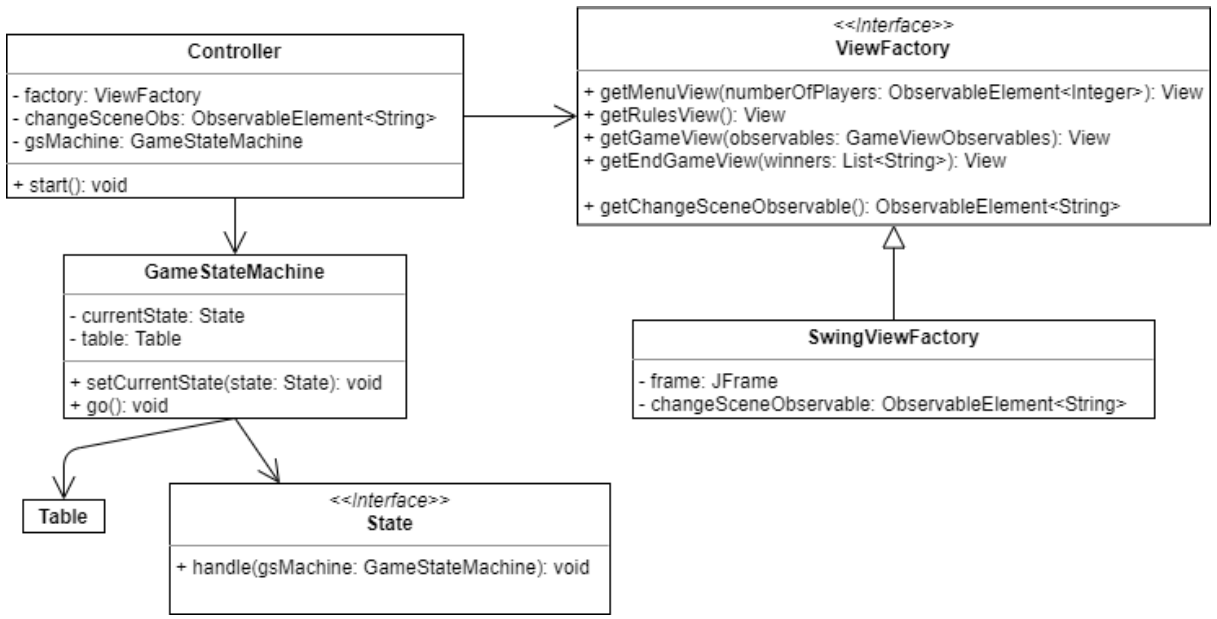


Figura 2.1: Schema UML del pattern architetturale MVC.

2.2 Design dettagliato

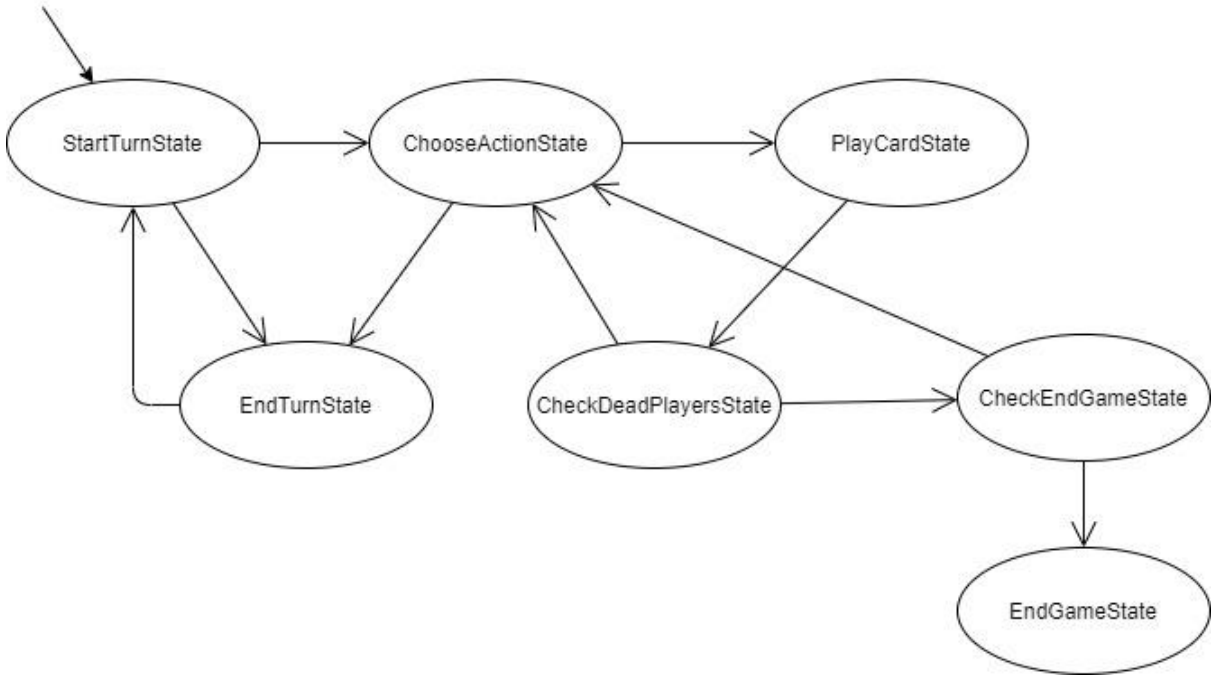


Figura 2.1: diagramma degli stati implementati con il pattern State.

Davide Merli

ViewFactory

Per la realizzazione della view si è scelto di utilizzare il pattern *AbstractFactory*. Tale pattern richiede l'utilizzo di una classe (SwingViewFactory) che implementi diversi metodi, ognuno dei quali ritorna un oggetto di una stessa classe. In questo specifico caso, si è implementata una factory di oggetti View. Tale classe permette la creazione delle varie schermate di gioco, implementate con la libreria Swing di Java. Qualora si volessero utilizzare altre librerie grafiche, sarebbe necessario creare una nuova classe che, come SwingViewFactory, implementi l'interfaccia ViewFactory, e modificare di conseguenza l'implementazione dei metodi.

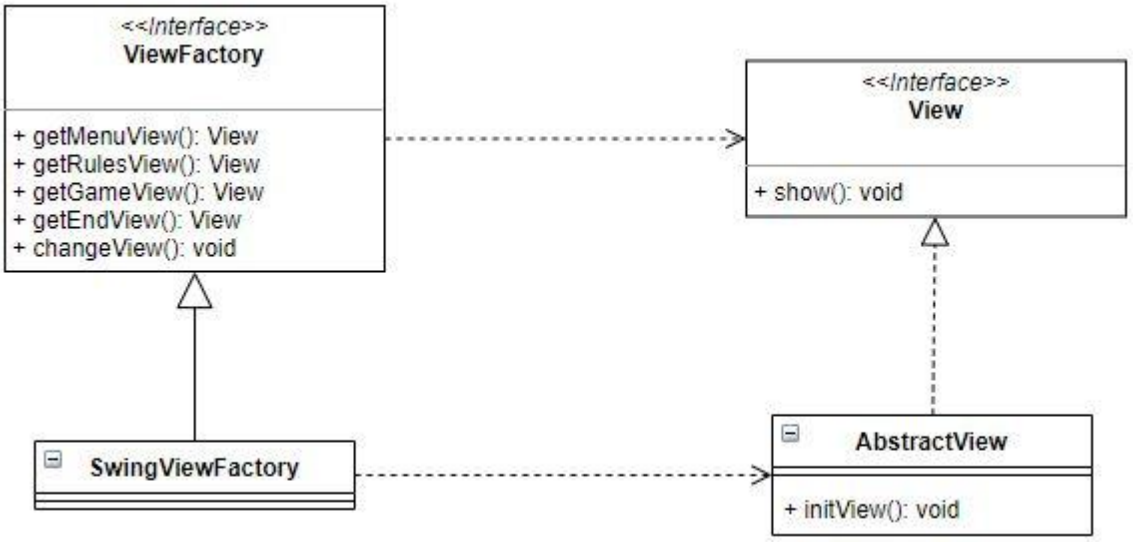


Figura 2.2: Rappresentazione UML del pattern AbstractFactory della view.

GameStateMachine

Per quanto riguarda la gestione della partita, si è deciso di suddividere il gioco (e i turni dei giocatori) in più fasi. Per implementare tale scelta si è deciso di applicare il pattern *State*. Una classe principale, la GameStateMachine, possiede uno stato. Ogni stato, che implementa l'interfaccia State, effettua diverse azioni e poi fa sì che si passi allo stato successivo. In questo modo, ogni fase di gioco è identificata da uno stato, e basta chiamare il metodo specifico di uno stato per gestire tutto ciò che deve avvenire nelle varie fasi di gioco. Inoltre, l'utilizzo di tale pattern rende semplice l'aggiunta di eventuali altri stati.

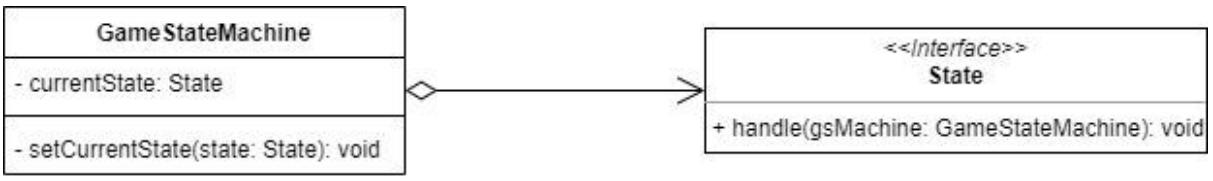


Figura 2.3: Rappresentazione UML del pattern State.

Table

Per il tavolo di gioco, contenente alcuni metodi utilitari all'andamento del gioco stesso, è stato utilizzato il pattern strategy, in modo che all'occorrenza si possano aggiungere altri tavoli che applichino regole di gioco diverse.

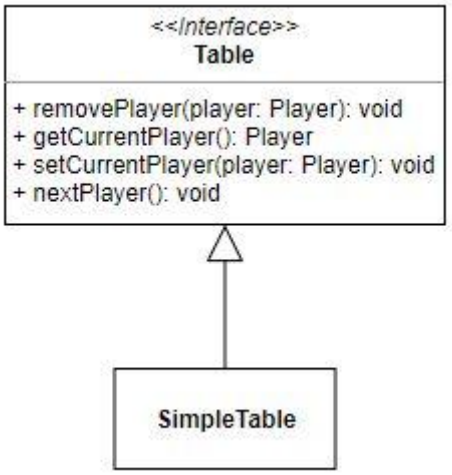


Figura 2.4: Rappresentazione UML del pattern Strategy per il Table.

CircularList

Al fine di gestire la relazione di distanza fra i giocatori, parte rilevante per molte meccaniche di gioco di "BANG!", si è pensato di creare una classe che implementi una lista circolare, con la quale è semplice ottenere l'elemento successivo e precedente di ogni elemento, nonché conservare informazioni riguardanti un elemento corrente.

Mattia Marchesini

Observable

Il pattern *observable* si è reso particolarmente necessario per le comunicazioni, di natura non bloccante, tra view e controller.
Questi due elementi devono poter reagire, eseguendo del codice specifico, quando vengono notificati. Ad esempio, il controller deve agire sul model quando l'utente decide di compiere un'azione premendo un pulsante dalla view e, viceversa, la view

deve contenere del codice per aggiornarsi quando certi valori vengono modificati dal controller.

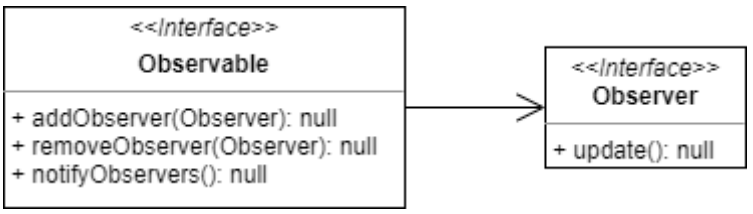


Figura 2.5: schema UML del pattern Observer (1)

Molto utile è stata la classe ObservableElement, implementazione di Observable, che aggiunge la funzionalità di avere un valore di qualsiasi tipo associato all'observable.

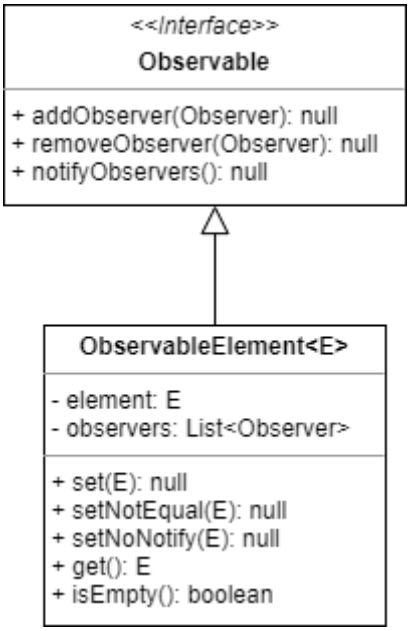


Figura 2.6: schema uml del pattern Observer (2)

In questo modo il controller può facilmente modificare qualsiasi aspetto della view semplicemente *settando* l'ObservableElement corrispondente.

Questi oggetti sono stati usati, in modo limitato, anche per far comunicare tra loro alcuni componenti facenti parte del model. Un esempio sono gli effetti di alcune carte che necessitano una "scelta" da parte del giocatore (es. a chi sparare quando si usa una carta "bang"), per cui il metodo useEffect() della classe Effect non può esaurire il suo compito immediatamente.

Deck e DeckReader

L'interfaccia Deck specifica le azioni tipiche di un mazzo come mescolare le carte, prendere la prossima carta, iniziare una nuova partita con un nuovo mazzo ecc.

La classe astratta AbstractDeck utilizza il pattern *strategy*: per creare un nuovo mazzo, implementato con una List<Card>, si passa al costruttore un oggetto di tipo DeckReader, che si occupa di "leggere" in qualche modo i dati relativi al deck e presentarli a Abstract deck come una lista di carte.

La classe Deck è un'implementazione di AbstractDeck, e si limita a passare al costruttore della sua classe padre un'istanza di JSONDeckReader, implementazione di DeckReader che legge i dati relativi al mazzo da un file json.

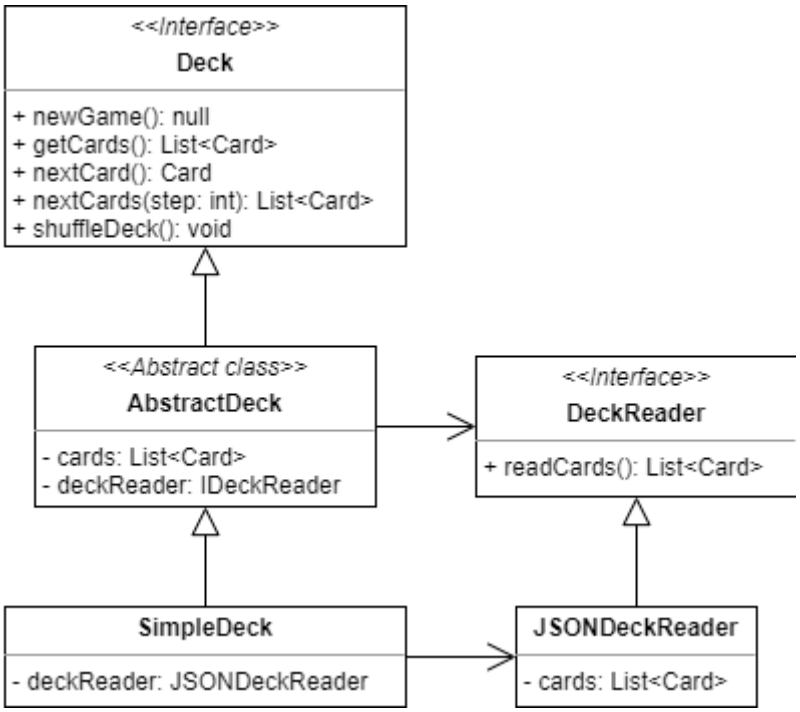


Figura 2.7: Schema UML del pattern Strategy per il deck.

Un'altra implementazione possibile sarebbe stata di non rendere astratta la classe AbstractDeck (che di per sé è completa già di suo), e utilizzarla passando ogni volta un oggetto DeckReader. Si è invece scelto, per rendere più semplice l'utilizzo all'interno del progetto, di usare una implementazione "fissa" e rendere la lettura di un file json il metodo standard per leggere i dati del mazzo.

Controller, ViewController e GameController

All'avvio, la classe contenente il metodo main chiama una nuova istanza di Controller, passando come parametro al costruttore un'implementazione specifica di ViewFactory da utilizzare. Per iniziare il gioco, basterà chiamare il metodo start().

Quando viene istanziata, la classe Controller mappa delle stringhe a un'implementazione dell'interfaccia funzionale ViewController. Ogni implementazione fungerà da controller per una view specifica. Poiché il controller per l'interfaccia di gioco vera e propria risulta particolarmente più complesso rispetto agli altri, si è utilizzato il pattern *strategy*, delegando il lavoro all'oggetto GameController (quindi nell'implementazione non si fa altro che creare una nuova istanza di GameController e chiamare il suo metodo setup())

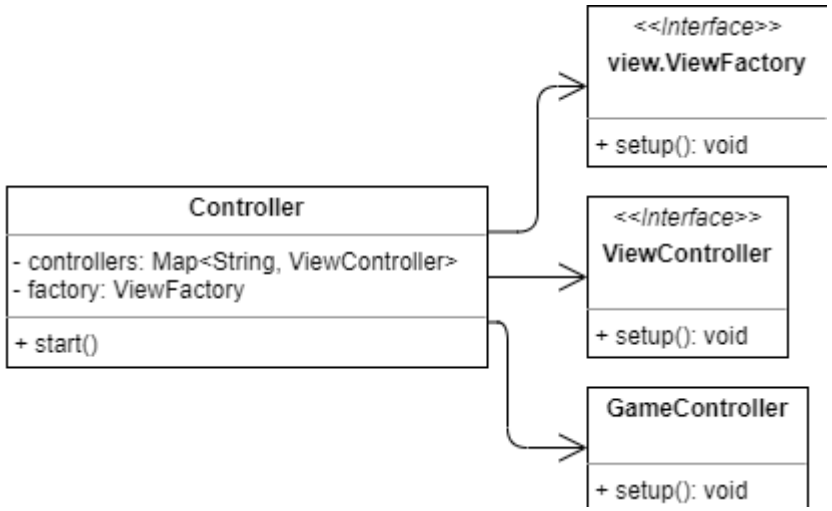


Figura 2.8: schema UML dei controller.

Infine, Controller aggiunge un Observer a un ObservableElement<String> presente nella classe GameStateMachine. Questo ObservableElement servirà a GameStateMachine per comunicare con il controller alla bisogna.

Resources

Classe contente metodi statici di utility per leggere file e immagini. Si è resa particolarmente utile in un progetto come questo, dove l'accesso a file avviene spesso (lettura di un file di testo per le regole, lettura di un file json per il mazzo, lettura delle immagini corrispondenti alle carte).

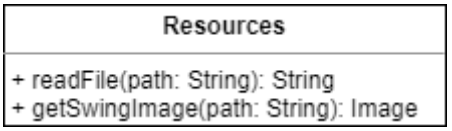


Figura 2.9: schema UML della classe Resources.

Ryan Perrina

Logics

La classe Logics è stata realizzata per rendere possibile l'uso della carta bang. All'interno di questa classe è un presente metodo che servono per restituire una lista di giocatori che posso decidere di colpire.

Card, Effects e Color

Per realizzare le carte al fine che ognuna avesse un diverso effetto, si è deciso di utilizzare il pattern Strategy. Tale pattern richiede un'interfaccia (Effects) che specifichi uno o più metodi che verranno implementati in modo diverso a seconda delle tipologie di carte. Per evitare ripetizioni di codice si è pensato di suddividere gli effetti in: Bang, CatBalou, Dodge, DrawCardsFromDeck, Emporium, Gatling, Indians, Jail, Missed, ModifyLifePoints, ModifyRetreat, Panic, Saloon, Scope, Weapond. Inoltre nell'ottica di facilitare nuove espansioni del gioco "Bang", si è deciso di usare un enum (Color) in cui viene espresso che prima degli effetti bisogna classificare le carte per colore.

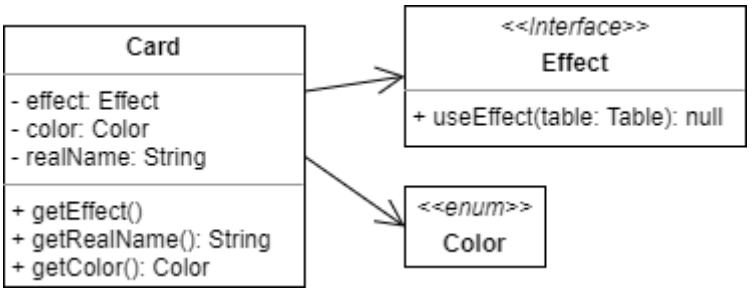


Figura 2.10: schema UML del pattern Strategy per le carte.

Player, SimplePlayer

Nelle regole del gioco da tavolo per fare in modo che la partita inizi abbiamo bisogno di giocatori in un numero che va dai 4 ai 7. L'idea originale per sviluppare i giocatori con punti vita,carte e molti altri attributi di cui tenere conto, era di creare dall'interfaccia Player ed aggiungere anche i player con delle caratteristiche speciali (i character) usando il pattern Stratey, purtroppo per i problemi sopracitati non ci è stato possibile aggiungere altro che il SimplePlayer. Non avendo a disposizione altri player che il SimplePlayer abbiamo assegnato una base di 4 vite (portate poi a 5 per lo Sceriffo).

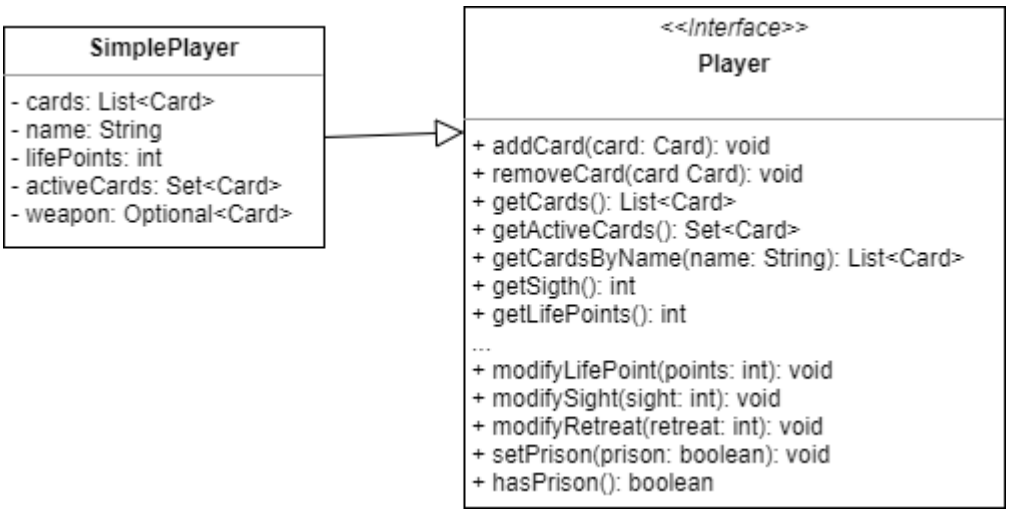


Figura 2.11: schema UML per Player e SimplePlayer.

GameViewObservable

L'obiettivo della classe GameViewObservable è di settare,. usando la classe ObservableElement,alcune informazioni necessarie alla view e che si potessero reperire in accordo con il pattern mvc.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

La realizzazione del testing automatizzato è avvenuta utilizzando la suite JUnit, nella versione 5.
Le classi sottoposte a testing automatizzato sono:

- CircularList: testing dei metodi getNext() e getPrev(), che scorrono la lista verso destra o verso sinistra, per controllare in particolare i casi in cui l'elemento corrente si trovi all'inizio o alla fine della lista
- Logics: testing sul metodo getTargets(), per verificare che la distanza fra giocatori venga calcolata nel modo corretto, anche nel caso in cui i giocatori abbiano attivato carte che modificano suddetta distanza
- State: testing generale sui passaggi di stato operati dalla GameStateMachine

3.2 Metodologia di lavoro

Il programma della divisione del lavoro non è stato rispettato completamente. Ciò è dovuto principalmente che uno dei quattro componenti iniziali del gruppo ha dovuto lasciare inaspettatamente il lavoro a causa di problemi personali. I tre componenti rimasti si sono dovuti dividere quindi una parte di lavoro aggiuntivo. Inoltre, alcuni cambiamenti sono dovuti al fatto che certe meccaniche non erano state inserite nel progetto originale, ma solo in seguito, in fase di sviluppo.
Alcune parti sono state sviluppate in gruppo da più studenti, in particolare il pattern State (Davide Merli e Mattia Marchesini) e il pattern Strategy per gli effetti delle carte (Ryan Perrina e Mattia Marchesini).
Per la realizzazione di questo progetto condiviso si è deciso di adottare il DVCS Git, in quanto di semplice utilizzo e conosciuto da tutti i membri del gruppo. Il software finale si trova sul branch "main", ma durante lo sviluppo dell'applicazione sono stati creati svariati branch temporanei per testare separatamente l'aggiunta di nuove feature al programma.

Mattia Marchesini

Realizzazione di tutto il codice dentro i package controller, libs.observe, model.deck.
Realizzazione di alcuni stati (CheckGameOverState, ChooseActionState, EndGameState, PlayCardState, StartTurnState), TurnObservable, Pair.
Effetti delle carte: Bang, Dodge, Emporium, Jail, Gatling Indians, Saloon, Panic.

Davide Merli

Realizzazione della grafica di gioco (schermata di menu principale, schermata del regolamento, schermata di gioco, schermata di game-over), realizzazione (congiunta) della GameStateMachine, alcuni stati (ChooseActionState, CheckDeadPlayersState, EndTurnState, CheckEndGameState), tavolo di gioco. Le classi implementate si trovano principalmente nei package view e states.

Ryan Perrina

Realizzazione del Player,SimplePlayer, Card,sviluppo congiunto della view , sviluppo congiunto degli effetti delle carte,realizazzione dei test in particolare TestLogics e TestState.Le classi si trovano in model.effect, model, view, interfaccia Effect.
Effetti delle carte: Weapon, DrawCardFromDeck, ModifyRetreat, CatBalou, Missed, Scope.

3.3 Note di sviluppo

Mattia Marchesini

- Utilizzo delle lamba expressions in molteplici classi, in particolare all'interno del controller e per aggiungere Observer agli oggetti di tipo Observable
- Utilizzo degli stream negli stati e negli effetti di mia competenza
- Utilizzo dei generici in ObservableElement e in Pair
- Utilizzo degli Optional usando stream in PlayCardState e in alcuni effetti
- Utilizzo della liberia GSon per la deserializzazione da json alla classe JSONCard
- Uso della reflection per capire la classe di appartenenza di un effetto

Davide Merli

- Utilizzo della libreria grafica Swing
- Utilizzo di lambda expressions, specie con stream di liste
- Utilizzo limitato di Optional
- Definizione di nuove annotazione per la creazione della documentazione (javadoc)
- Creazione di una classe generica (CircularList) come estensione della classe List

Ryan Perrina

- Utilizzo di lambda expressions,e stream
- Utilizzo della classe Optional
- Realizzazione dalla javadoc
- Utilizzo di junit

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Possibili miglioramenti al progetto:

- Possibilità di aggiunta delle carte personaggio e delle relative abilità speciali
- Possibilità di aggiunta dei semi delle carte, utili per certi effetti aggiuntivi
- Possibilità di aggiunta della carte contenute nelle espansioni di gioco, che introducono nuove meccaniche di gioco
- Realizzazione di un gameplay online
- Possibilità di avere una versione localizzata del nome dei personaggi e delle carte
- Capire come riprodurre i file audio delle carte quando il software è dentro un file jar

Mattia Marchesini

Date le circostanze, sono abbastanza soddisfatto del risultato ottenuto con questo progetto. Ritengo che concettualmente i design progettuali usati per il realizzarlo siano validi, ma avendo più tempo l'implementazione sarebbe potuta essere migliore.
Nel caso si volesse continuare a sviluppare questo progetto, la base è abbastanza solida per effettuare un refactoring e aggiungere funzionalità senza particolare difficoltà.

Davide Merli

Ritengo personalmente di aver svolto una parte abbastanza consistente del progetto, in particolare per quanto riguarda la parte di interfaccia utente. Tuttavia, se avessimo avuto più tempo a disposizione, avrei preferito studiare librerie grafiche più complesse, come JavaFX, per la realizzazione di una grafica migliore. Sono inoltre piuttosto soddisfatto della classe generica CircularList, in quanto ritengo possa tornare utili in svariate occasioni, nonostante la sua realizzazione non sia stata eccessivamente complessa.
Penso che il problema principale del gruppo sia stato partire: abbiamo impiegato troppo tempo a decidere che progetto portare e a come realizzarlo. Inoltre abbiamo sottovalutato la complessità dell'applicazione.
Nel complesso, considerando che questo era il mio primo progetto di dimensioni considerevoli, posso ritenermi abbastanza soddisfatto del lavoro mio e del mio gruppo.

Ryan Perrina

Nel complesso mi ritengo soddisfatto del mio lavoro anche se sono dispiaciuto di non essere riuscito a sopperire completamente alla parte mancante: seppur avendo l'idea, come specificato in precedenza, il tempo non era sufficiente. Avrei voluto aggiungere suoni anche per altri eventi come ad esempio l'inizio del gioco, e magari aggiungere una musica di sottofondo.

4.2 Difficoltà incontrate e commenti per i docenti

Mattia Marchesini

Ovviamente la mancanza di un componente del gruppo ha portato a ritardi e la mancanza di alcune funzionalità che ci eravamo prefissati.
Pur avendo già affrontato il progetto l'anno scorso ho ancora qualche difficoltà a capire in che misura utilizzare i diversi pattern senza ricadere nell'*over-engineering* o, all'incontrario, nello *spaghetti-code*.
Mi sarebbe piaciuto avere una visione più completa del progetto fin dall'inizio.
Nonostante ciò ritengo di aver prodotto molto più codice dell'anno scorso e ne sono adeguatamente soddisfatto.
Per quanto riguarda il progetto in sé, quest'anno ho avuto nuovamente qualche difficoltà e ho aiutato i miei compagni in problemi che già avevo avuto col progetto precedente.

Davide Merli

Penso che il corso non abbia trattato a sufficienza l'uso di gradle/maven. Tale argomento è stato trattato solo una volta in laboratorio, e a mio parere in modo poco approfondito. La creazione di un progetto maven e di un file jar tramite maven ha causato ben più problemi si quanto non ci si aspettasse.
Inoltre dedicherei più tempo alla parte riguardante i pattern, non tanto dal punto di vista teorico quanto alla loro applicazione pratica (in laboratorio): una corretta organizzazione del pattern MVC è stata problematica, specie nel capire bene il confine tra model e controller. Anche il pattern observer, dal nostro gruppo molto utilizzato, è stato un punto critico, e solo lievemente esposto in classe.

Ryan Perrina
Ho incontrato difficoltà nel scegliere alcuni pattern piuttosto che altri, a mio parere, sono stati trattati in modo troppo rapido e con pochi esempi a cui fare riferimento. Inoltre nella parte di laboratorio non sono stati tratti alcuni aspetti come l'uso di una libreria grafica esterna.

Appendice A

Guida Utente

Menu principale

Il menu principale, molto semplice, presenta tre pulsanti:

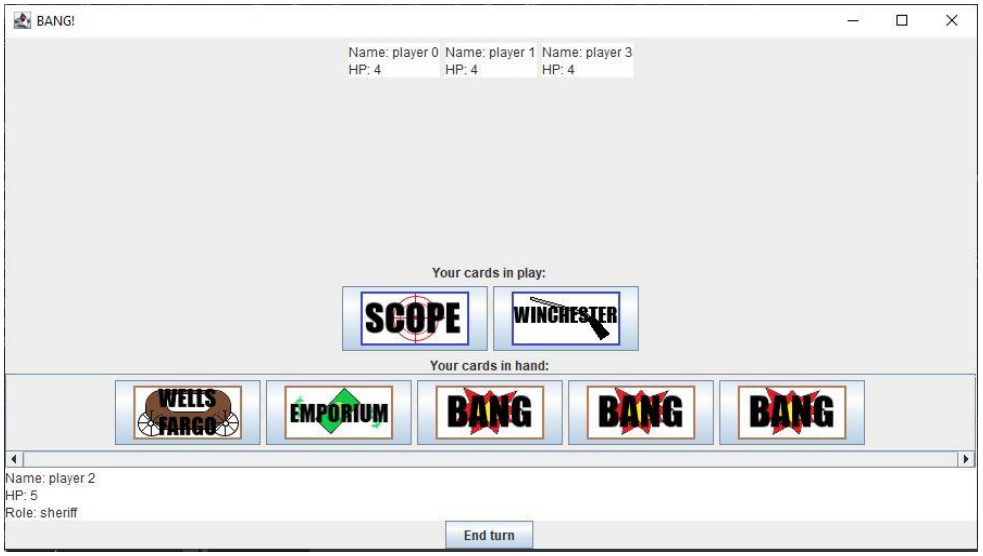
- "Play": fa comparire una finestra che permette all'utente di scegliere il numero di giocatori (da 4 a 7). Una volta scelto tale numero e cliccato sul pulsante "Ok" si apre la schermata di gioco.
- "How to play": fa comparire una schermata che spiega molto brevemente le principali regole del gioco.

- “Quit”: esce dal gioco.

Schermata delle regole di gioco

- Nella parte bassa di questa schermata sono presenti tre pulsanti, “Roles”, “Brown cards” e “Blue cards”, che permettono di visualizzare, rispettivamente, le regole riguardanti i ruoli dei personaggi, il funzionamento delle carte marroni e il funzionamento delle carte blu. Infine, cliccando il pulsante “Back”, si torna al menù principale.

Schermata di gioco



Nella parte inferiore dello schermo sono visualizzate le informazioni riguardanti il giocatore corrente. Dall’alto verso il basso si trovano:

- le carte blu (attive) del giocatore di turno
- le carte in mano del giocatore di turno
- le informazioni riguardanti il nome, i punti vita e il ruolo del giocatore di turno
- un pulsante da cliccare quando si vuole passare il turno

Cliccando su una delle carte in mano (non delle carte blu in gioco) viene mostrata una finestra che permette di scegliere se si vuole giocare (“Play”) o scartare (“Discard”) la suddetta carta. Alcune carte devono essere giocate contro specifici giocatori scelti dall’utente: per tali carte, in seguito alla pressione del tasto “Play”, viene mostrata un’ulteriore schermata che permette all’utente di scegliere il bersaglio. In tale schermata non vengono visualizzati tutti i giocatori, ma solo i possibili bersagli di una certa azione (che dipendono da vari fattori). Qualora non fosse possibile giocare una carta, cliccando sul tasto “Play” la carta selezionata viene scartata.

Cliccando sulle carte blu in gioco, invece, non accade nulla, poiché tali carte sono passive: il loro effetto continua a valere fintanto che tali carte restano in gioco.

Nella parte superiore dello schermo sono visualizzati invece dei riquadri contenenti le informazioni riguardanti gli avversari del giocatore corrente: nome e punti vita. Non viene mostrato il ruolo degli avversari, poiché tale ruolo deve restare segreto. L'unica eccezione è lo sceriffo ("sheriff"), poiché tutti i giocatori sono a conoscenza di chi sia. Se questi giocatori hanno attivato delle carte blu, tali carte vengono visualizzate sotto il riquadro con le informazioni sul giocatore.

Schermata di game-over

Al termine della partita viene visualizzata un'ulteriore schermata, nella quale si leggono i nomi (uno o più) dei giocatori che hanno vinto la partita. Cliccando sul pulsante "Quit" l'applicazione si chiude.

Appendice B

Esercitazioni di laboratorio

Bibliografia

Per il risolvere vari dubbi riguardo utilizzo delle librerie swing,json,per il sound,jar abbiamo usato il sito StackOverflow.