Relazione per "Programmazione ad Oggetti"

Progetto: Marvel Snap (Java)

Studenti:

- [Del Moro Matteo]
- [Gennari Lorenzo]
- [Lumini Marco]
- [Spazzoli Paolo]

Data: [15/02/2026]

Sommario

Il presente documento descrive il processo di analisi, progettazione e sviluppo di un'applicazione software che riproduce le meccaniche principali del gioco di carte "Marvel Snap". L'applicazione è stata realizzata utilizzando il linguaggio Java e segue i principi della programmazione ad oggetti. La relazione illustra i requisiti del dominio, l'architettura software adottata (MVC), i pattern di progettazione utilizzati per risolvere specifici problemi e le soluzioni implementative principali.

Capitolo 1: Analisi

In questo capitolo viene descritto il dominio applicativo e i requisiti del software, astraendo dalle scelte implementative.

1.1 Descrizione e requisiti

Il software deve simulare una partita strategica a carte tra due giocatori ambientata nell'universo Marvel. Ogni carta ha un "Costo", una "Forza" e una abilità che si attiva quando la carta viene giocata. Ogni giocatore ha un certo livello di energia e può giocare carte finché la sua energia non arriva a zero. L'obiettivo è accumulare più "Forza" dell'avversario in tre campi distinti.

Requisiti Funzionali

Il sistema deve garantire i seguenti aspetti:

- **Gestione della Partita:** Avviare una nuova partita inizializzando i giocatori, i mazzi e le tre location casuali.
- **Gestione dei Turni:** Gestire una sequenza finita di turni (6 turni standard). In ogni turno, i giocatori possiedono un quantitativo di energia pari al numero del turno (Al primo turno si ha energia 1, al secondo energia 2 e così via fino all'ultimo).
- **Gestione Carte e Mazzi:** Ogni giocatore possiede un mazzo e una mano. Le carte hanno un "Costo" (energia richiesta per giocarla) e una "Forza". Ogni mano inizialmente è formata da 3 carte e, ad ogni turno, ogni giocatore deve pescare una carta che si aggiunge alla sua mano.
- **Giocata:** Permettere al giocatore di giocare carte dalla mano su uno dei tre campi, rispettando il vincolo di energia e il limite di spazio del campo (max 4 carte).

- **Risoluzione Effetti:** Le carte possono avere abilità che si attivano quando rivelate (OnReveal). I campi possono avere effetti passivi o attivi che modificano la forza o le regole.
- **Condizione di Vittoria:** Al termine della partita, vince chi ha più “Forza” su almeno 2 campi su 3. In caso di pareggio (quindi se su un campo i giocatori hanno la stessa forza), viene calcolata la somma delle “Forze” sui campi per ognuno dei giocatori e vince chi ha questa somma maggiore. Se anche questa dovesse risultare uguale, allora si ha il pareggio assoluto.

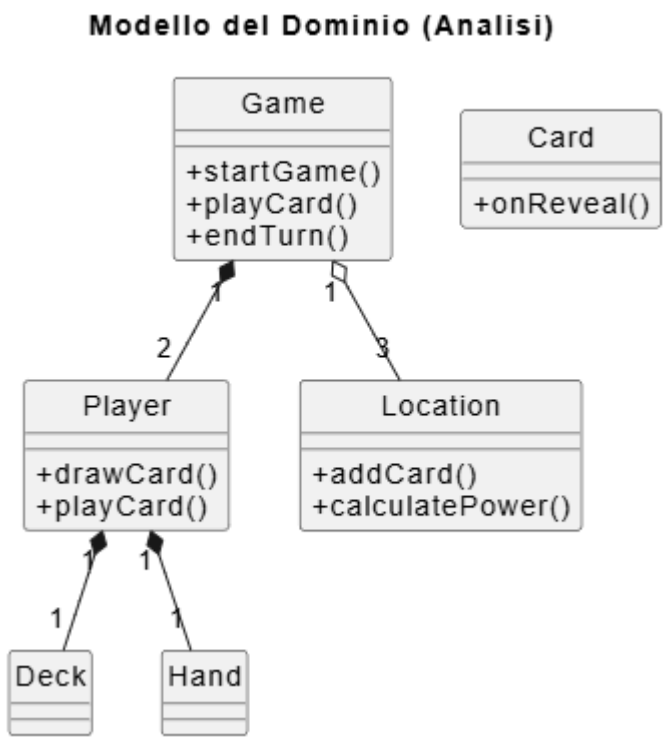
Requisiti Non Funzionali

- **Interfaccia Utente:** Il gioco deve presentare un'interfaccia grafica (GUI) che permetta l'interazione via mouse (click).
- **Estensibilità:** Il sistema deve essere progettato per facilitare l'aggiunta di nuove carte e nuove Location senza ristrutturare il core del gioco.

1.2 Modello del Dominio

Il dominio del gioco è composto dalle seguenti entità principali:

- **Game:** L'entità che raggruppa lo stato corrente della partita.
- **Player:** Rappresenta un giocatore. Ogni Player possiede un **Deck** (mazzo) da cui pesca e una **Hand** (mano) di carte attuali.
- **Location:** Rappresenta uno dei tre campi di gioco. Ogni Location ospita le carte giocate dai due Player e calcola la Forza locale.
- **Card:** L'entità base di gioco. Possiede attributi statici (Costo, Forza base) e abilità dinamiche.
- **TurnManager:** L’entità che regola il flusso dei turni e l'energia disponibile ad ogni turno.



Capitolo 2: Design

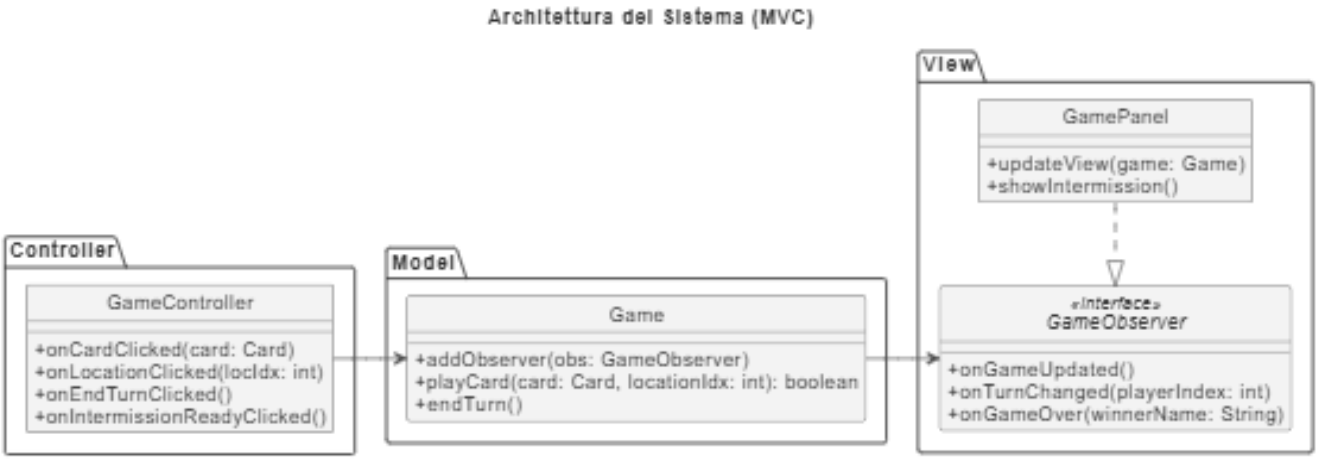
In questo capitolo si illustrano le strategie architetturali e i pattern di progettazione utilizzati.

2.1 Architettura

L'applicazione adotta il pattern architetturale **Model-View-Controller (MVC)** per separare la logica del gioco dall'interfaccia utente.

- **Model (model package):** Contiene la logica del gioco (Game, Player, Card, Location…). È completamente all'oscuro dell'interfaccia grafica. Notifica i cambiamenti di stato tramite il pattern *Observer*.
- **View (view package):** Responsabile della visualizzazione (MainFrame, GamePanel, CardPanel…). Osserva il Model e si aggiorna di conseguenza. Intercetta gli input dell'utente e li delega al Controller.
- **Controller (controller package):** Gestisce il flusso e gli input (GameController, MainController). Riceve i comandi dalla View (es. "Carta Cliccata"), elabora la logica interagendo con il Model e, se necessario, aggiorna lo stato della View (es. cambio schermata).

La comunicazione tra Model e View avviene tramite l'interfaccia GameObserver, garantendo che il Model non dipenda da una specifica implementazione grafica.

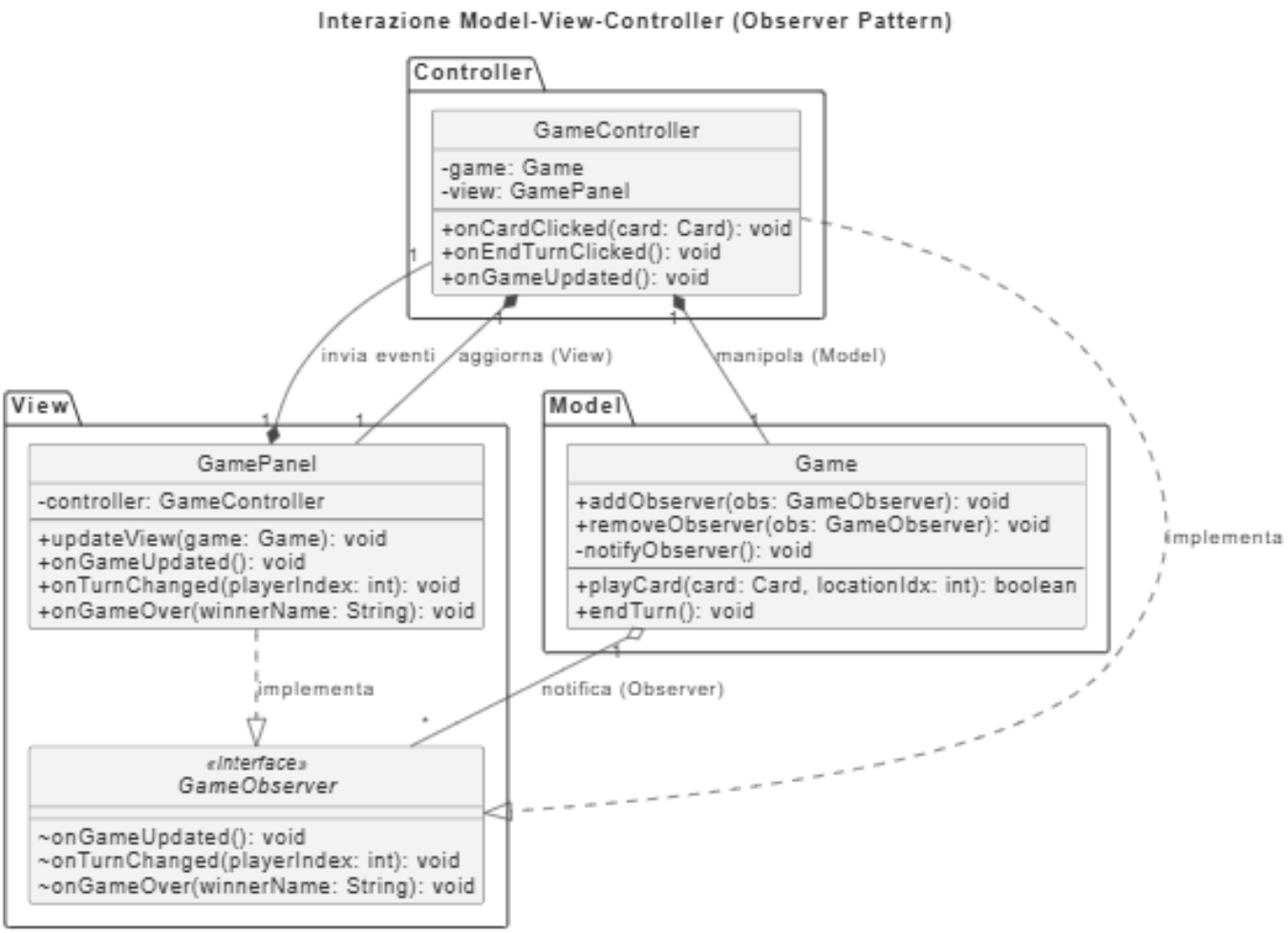


2.2 Design dettagliato

Di seguito vengono analizzate le soluzioni progettuali specifiche adottate dai membri del gruppo.

2.2.1 Gestione del Flusso di Gioco e Observer (a cura di Del Moro Matteo)

Problema: È necessario aggiornare l'interfaccia grafica in tempo reale quando lo stato del gioco cambia (es. fine turno, carta giocata), lasciando separati Model e View.
Soluzione: È stata utilizzata l'interfaccia GameObserver, la quale definisce i metodi di notifica (onGameUpdated, onTurnChanged) che osservano i cambiamenti del Model e cambiano la view di conseguenza. La classe Game agisce da *Subject*, ossia l'oggetto osservato, mantenendo una lista di osservatori e notificandoli.
Pattern: *Observer*:

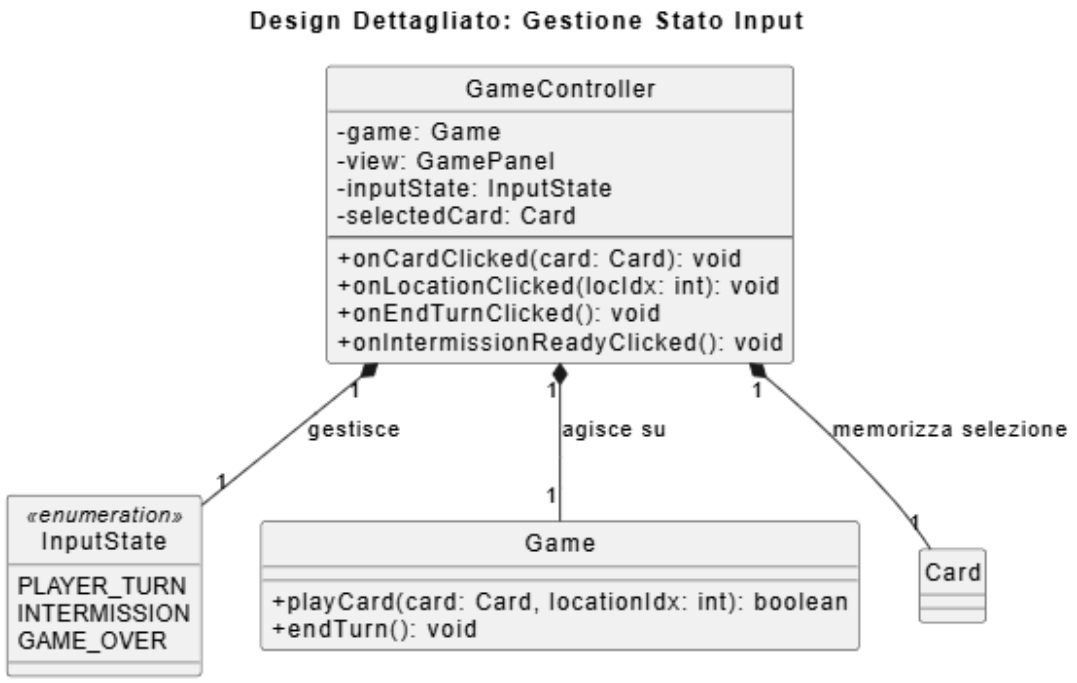


2.2.2 Gestione degli Stati di Input (a cura di Del Moro Matteo)

Problema: La gestione delle interazioni dell'utente può diventare caotica quando diverse azioni sono permesse o negate in base al momento del gioco (es. non si può giocare una carta se è il turno dell'avversario o se si è già passati). L'uso di variabili booleane sparse renderebbe il codice difficilmente leggibile e propenso a bug.

Soluzione: È stata implementata una macchina a stati semplificata basata sull'enumerativo InputState. Questo approccio permette di centralizzare il controllo sulle azioni dell'utente, filtrando quelle valide. Questo garantisce che la View reagisca in modo coerente e che l'utente possa interagire solo con gli elementi permessi dalla fase di gioco attuale.

Pattern: *State* (semplificato tramite Enum)**.**



___

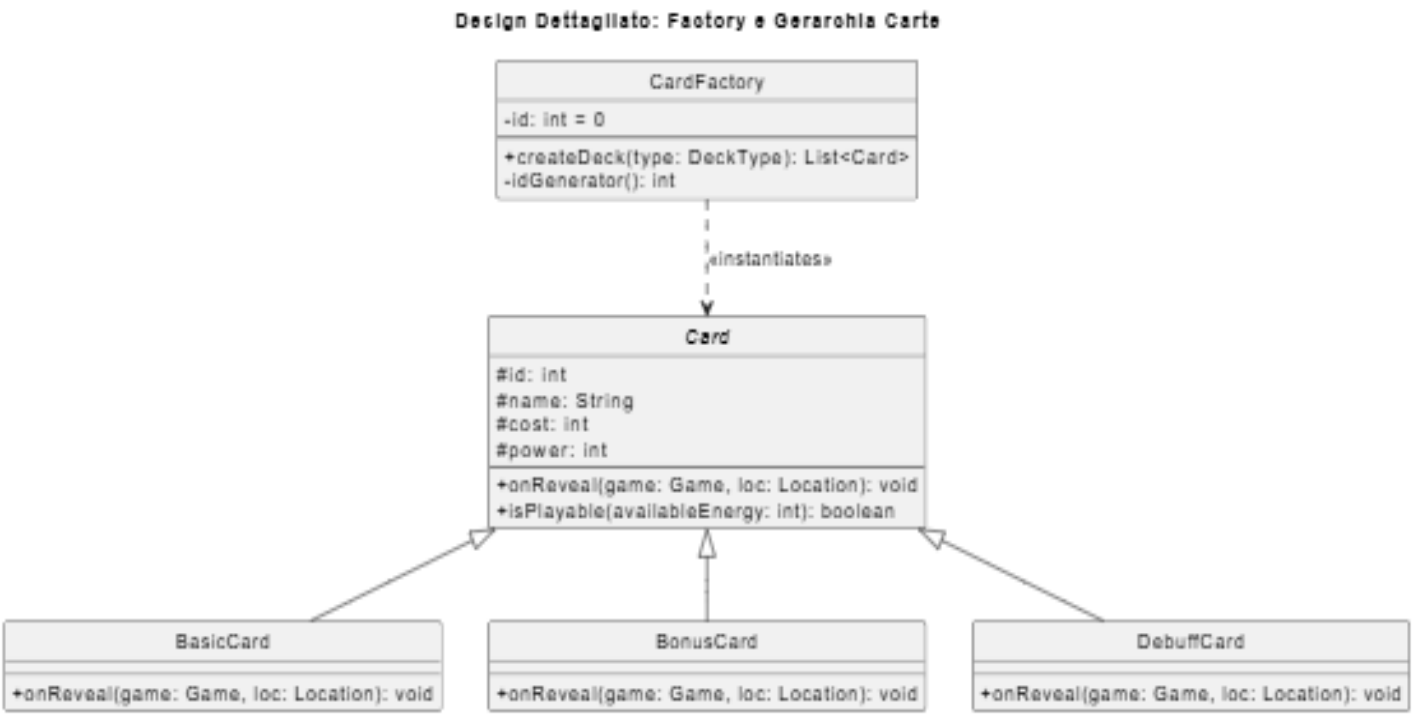
2.2.3 Gerarchia delle Carte e Factory (a cura di Gennari Lorenzo)

Problema: Il gioco prevede numerose carte con comportamenti diversi (Basic, Bonus, Debuff), ma il sistema deve trattarle in modo uniforme. Inoltre, la creazione dei mazzi richiede l'istanziazione flessibile di molte carte. Questo potrebbe portare a una class explosion.
Soluzione:

- Si è utilizzata una classe astratta Card che definisce il metodo astratto onReveal(). Le sottoclassi implementano le logiche specifiche. Questo configura un utilizzo del polimorfismo che richiama il pattern *Template Method* (dove il gioco chiama playCard che internamente invoca l'hook onReveal).
- Per la creazione delle carte è stata introdotta la classe CardFactory, che centralizza la logica di istanziazione separandola dall'utilizzo, come da classico Factory Pattern. In questa maniera, utilizzando classi anonime durante l'istanziazione delle varie carte, ho potuto evitare la class explosion.

Pattern:

- *Factory Pattern*: CardFactory nasconde la logica di istanziazione dal resto.
- *Template Method* (implicito): tutti i figli di Card devono implementare solo un metodo fondamentalmente il resto uguale, ma implicito perché non ho reso final i comportamenti in comune per rimanere flessibile nella scrittura del codice.



2.2.4 Architettura della Navigazione e Main Menu (a cura di Gennari Lorenzo)

Problema: L'applicazione deve gestire diverse fasi: il menu iniziale, la configurazione della partita (nomi e mazzi) e il gameplay vero e proprio. Gestire queste transizioni aprendo e chiudendo diverse finestre (JFrame) risulterebbe inefficiente, causerebbe sfarfallio e renderebbe difficile la gestione centralizzata dello stato del programma.

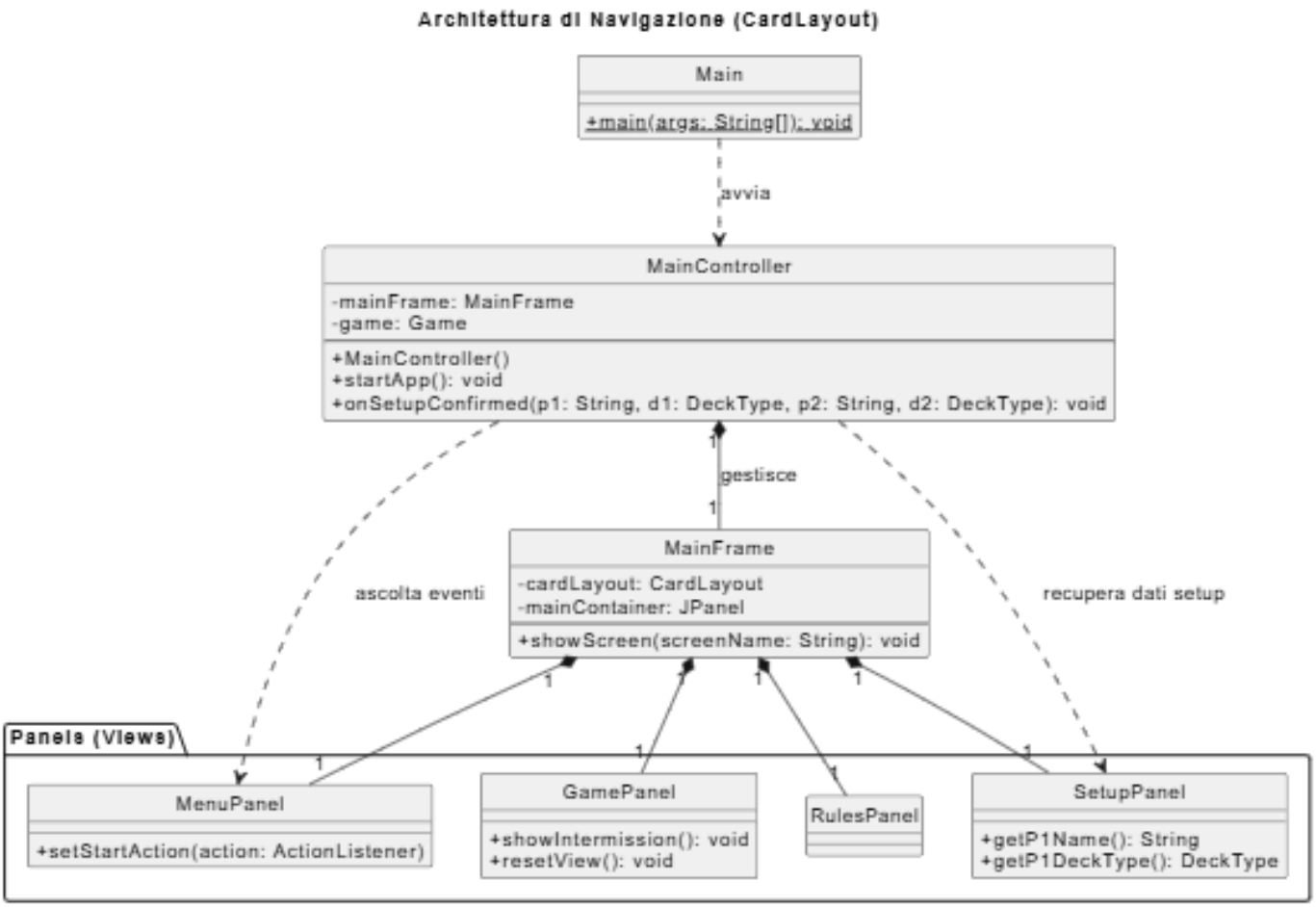
Soluzione: Si è adottata un'architettura basata su un unico contenitore principale, MainFrame, che delega lo scambio delle schermate a un CardLayout.

1. MainFrame: Agisce come cornice della finestra.
2. MainController: Gestisce la logica di navigazione. Implementa il passaggio tra i vari pannelli (MenuPanel, SetupPanel, GamePanel) rispondendo agli eventi dell'utente.
3. IntermissionPanel: Inserito per gestire la modalità "Hot Seat" (due giocatori sullo stesso PC), funge da "sipario" per nascondere le informazioni segrete tra un turno e l'altro, garantendo che un giocatore non veda la mano dell'altro.

Questa separazione permette di mantenere la logica della partita (GameController) isolata dalla logica di navigazione dell'interfaccia (MainController).

Pattern:

- *Mediator* (semplificato): Il MainController funge da mediatore tra i vari pannelli della View, decidendo quale mostrare in base all'input.
- *State Pattern* (implicito): La gestione dei pannelli tramite CardLayout riflette lo stato attuale dell'applicazione (In Menu, In Setup, In Game).



2.2.5 Gestione Dinamica della View (a cura di Gennari Lorenzo)

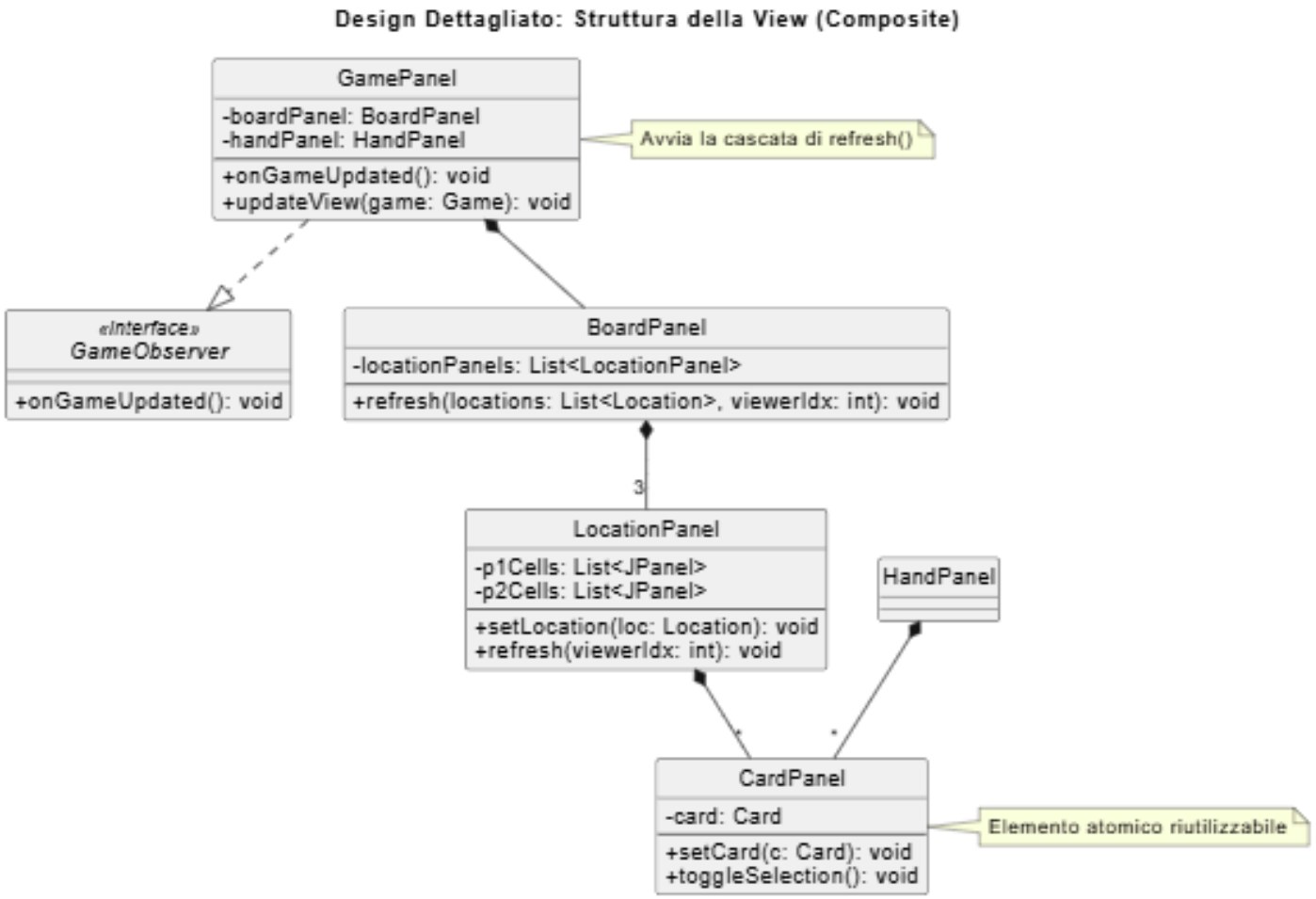
Problema: La schermata di gioco è complessa e composta da elementi ricorrenti (carte, location, mano). È necessario che la UI rifletta fedelmente e in tempo reale lo stato del modello (Pattern MVC), evitando però di ridisegnare l'intera interfaccia a ogni minima modifica, il che comprometterebbe le prestazioni.

Soluzione: È stato applicato il Composite Pattern per la costruzione della UI: ogni elemento complesso (come il BoardPanel) è composto da elementi più piccoli e atomici (LocationPanel, CardPanel). In particolare:

1. CardPanel: Centralizza la logica di visualizzazione di una carta. Grazie al metodo setCard(Card c), il pannello è in grado di riconfigurarsi dinamicamente per mostrare i valori di forza, costo e descrizione, rendendo la carta altamente riutilizzabile sia nella mano che sul campo.
2. Sincronizzazione (MVC): GamePanel implementa GameObserver. Quando il modello notifica un cambiamento, il pannello avvia una cascata di refresh() che si propaga verso il basso nella gerarchia dei componenti, garantendo la coerenza visiva.

Pattern:

- *Composite Pattern*: Per la gerarchia dei componenti Swing.
- *Observer Pattern*: Per il collegamento tra il Game (Model) e il GamePanel (View).



___

2.2.6 Modellazione del Giocatore e del Mazzo (a cura di Lumini Marco)

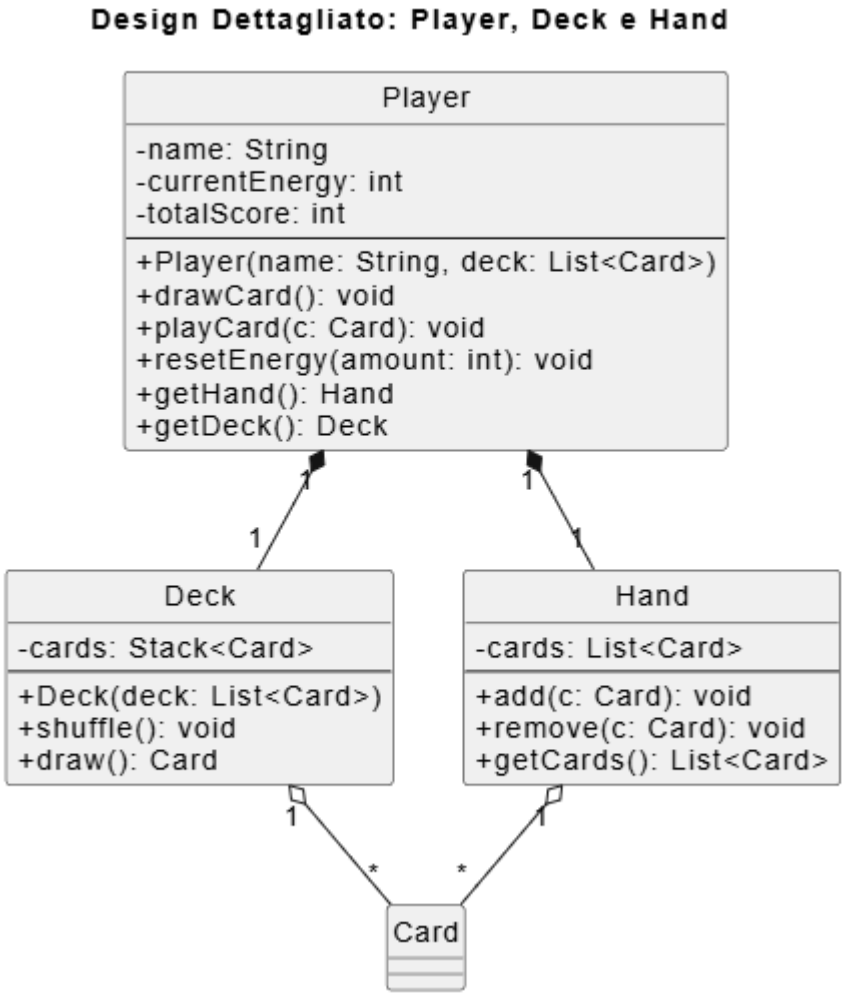
Problema: La difficoltà principale è stata la gestione dinamica delle carte all'interno del deck e della mano dei giocatori, ognuno dei quali doveva avere a disposizione una serie di dati da poter modellare per favorire la corretta progressione della partita. Era necessario, inoltre, che la gestione delle carte rispettasse i limiti impostati dalle regole del gioco.(es. non pescare da un mazzo vuoto o non pescare con già il numero massimo di carte in mano).

Soluzione: Per garantire la solidità del sistema, ho adottato un design basato sull'incapsulamento. Ho scelto strutture dati specifiche per ogni esigenza: uno Stack per il mazzo (ideale per la logica di pesca in cima) e una lista per la mano, resa accessibile all'esterno solo in modalità lettura. La classe Player è stata pensata invece per fungere da centro di controllo per coordinare il movimento delle carte tra deck mano e location tramite metodi come playCard() e drawCard(). Questo approccio ha permesso di gestire i casi limite e ha reso abbastanza semplice l'interazione con gli altri componenti i view e controller.

Problema: Il problema nella parte grafica è stato creare un'interfaccia intuitiva che fornisse al giocatore un feedback visivo immediato. Era necessario visualizzare chiaramente le informazioni di ogni carta (costo, forza, nome) e permettere un'interazione fluida con la mano del giocatore, rendendo evidente quale carta fosse selezionata o giocabile in ogni momento della partita.

Soluzione: Per la gestione della view, ho sviluppato un sistema di pannelli dedicati per le carte, capaci di gestire autonomamente il proprio stato grafico: attraverso l'uso di colori di sfondo e bordi dinamici, ho

garantito una distinzione tra lo stato normale e quello di selezione, rendendo intuitiva la fruizione del gioco per l'utente. Questi componenti sono stati poi integrati in un pannello contenitore per la mano, progettato per aggiornarsi dinamicamente in base alle variazioni riportate dal model. Infine per comunicare con il controller ho implementato dei listeners responsivi ai click da parte dell'utente affinché le azioni richieste potessero essere delegate.



2.2.7 Locations e Strategia (a cura di Spazzoli Paolo)

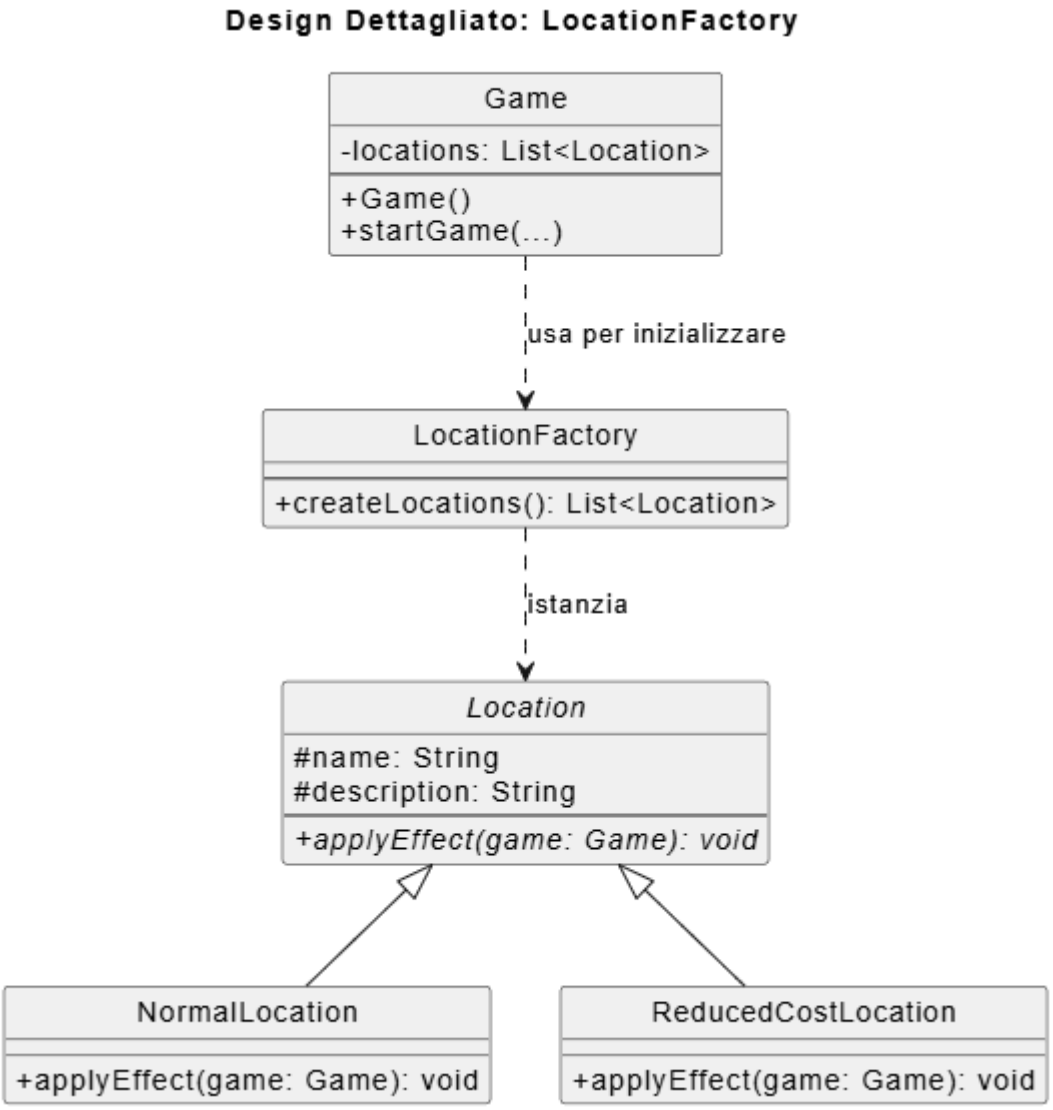
Problema: Le Location hanno effetti diversi che possono modificare alcuni aspetti della partita. (es. riduzione del costo delle carte in mano, aggiunta di forza alle carte posizionate).
Soluzione: La classe astratta Location definisce il contratto base, mentre le implementazioni concrete (ReducedCostLocation, NormalLocation) sovrascrivono il metodo astratto applyEffect(). Questo approccio permette di aggiungere nuove Location semplicemente estendendo la classe base, seguendo il principio Open/Closed.
Pattern: *Strategy* (l'effetto della location è la strategia variabile).

2.2.8 Creazione ed estrazione delle Location (a cura di Paolo Spazzoli)

Problema: il gioco deve presentare una pool varia di Location e dev'essere in grado, all'inizio di ogni nuova partita, di estrarne casualmente tre, diverse fra loro.

Soluzione: la classe LocationFactory si occupa della creazione della pool di Location, ognuna delle quali è rappresentata da una classe anonima che implementa in modo originale il metodo applyEffect(). Dopo la creazione, c'è un mescolamento delle location da parte del metodo Collection.shuffle() che permette in maniera semplice ed efficace di ottenere location diverse ed estratte in maniera aleatoria ad ogni partita.

Pattern: Simple Factory.



Capitolo 3: Sviluppo

3.1.0 Testing automatizzato

Sono stati realizzati JUnit test tramite la libreria **JUnit 5** per verificare la logica core del Model e il corretto funzionamento della GUI..

—

3.1.1 Testing automatizzato di Del Moro Matteo

I principali componenti testati sono i seguenti:

- **Game e TurnManager**: Sono stati realizzati test per verificare il corretto avanzamento del gioco. Si testano, quindi, l'avanzamento del numero del turno, lo switch dei giocatori all'interno dello stesso turno e la rivelazione delle Location e delle carte.
- **GameController**: Sono realizzati test per verificare che il controller aggiorni correttamente la view e gestisca correttamente la selezione delle carte.
- **WinCondition**: Si realizza un test per verificare che la classe WinCondition determini correttamente il vincitore nei tre casi possibili: vittoria normale(Forza maggiore su 2 Location su 3), Pareggio(Calcolo Forza totale), Pareggio assoluto.

3.1.2 Testing automatizzato di Gennari Lorenzo

I principali componenti testati sono i seguenti:

- **CardFactory e CardHierarchy**: Sono stati realizzati test per verificare che la creazione dei mazzi (Avengers, Villains, X-Men) produca il numero corretto di carte e che il polimorfismo tra BasicCard, BonusCard e DebuffCard funzioni correttamente.
- **SetupPanel**: Si è verificata la gestione dei componenti di input, assicurandosi che i nomi dei giocatori e le scelte dei mazzi vengano recepiti correttamente dalla View prima di essere passati al Controller.
- **MainNavigation** (GUI): Utilizzando la libreria AssertJ Swing, è stato automatizzato il flusso di navigazione. Il test simula il click sul pulsante "Nuova Partita", l'inserimento dei nomi nei campi di testo e la transizione al pannello di gioco.

3.1.3 Testing automatizzato di Lumini Marco

I principali componenti testati sono i seguenti:

- **Player, Hand e Deck:** I test verificano la corretta gestione delle collezioni che rappresentano la mano e il mazzo di ogni giocatore che prevedono il funzionamento di meccanismi di aggiunta e rimozione di carte. Inoltre per il giocatore i test si assicurano che gli aggiornamenti dell'energia durante la partita vengano eseguiti correttamente.
- **CardPanel, HandPanel:** Nei test realizzati per queste classi di view viene sostanzialmente controllata la correttezza della creazione dei componenti e dei loro contenuti. Si verifica anche che le funzionalità di aggiornamento visivo di selezione siano svolte con esattezza.

3.1.4 Testing automatizzato di Spazzoli Paolo

I principali componenti testati sono i seguenti:

- **LocationFactory**, **NormalLocation** e **ReducedCostLocation**: i test verificano la corretta creazione di tre diverse location per LocationFactory. Per quanto riguarda NormalLocation e ReducedCostLocation, i test verificano il funzionamento dei principali getters e setters, del polimorfismo tra le implementazioni delle due classi e della corretta gestione degli effetti specifici, assicurando che i buff (o debuff) di forza e costo vengano applicati nella maniera giusta.
- **BoardPanel** e **LocationPanel** (GUI): i test simulano, per mezzo della libreria AssertJ-Swing, la selezione di una carta dalla mano da parte dell'utente e la successiva giocata della carta nella prima location, verificando che la carta così selezionata venga realmente posizionata.

___

3.2.0 Note di sviluppo

___

3.2.1 Note di sviluppo di Del Moro Matteo

- **Utilizzo di LambdaExpression:**

  Utilizzata in GameController per gestire il collegamento con il bottone “Sono pronto”.

  Permalink:

  https://github.com/genna-12/marvel-snap-java/blob/main/src/main/java/com/marvelsnap/controller/GameController.java#L33

3.2.2 Note di sviluppo di Gennari Lorenzo

- **Uso di Lambda Expressions:**

  Utilizzate nel MainController per collegare gli eventi dei bottoni alla logica di navigazione. Questo approccio riduce il "boilerplate code" eliminando la necessità di creare classi ActionListener separate per ogni transizione. Utilizzate anche in diverse altre parti del codice.

  - **Permalink:**
    https://github.com/genna-12/marvel-snap-java/blob/0578ed03acd817f91374e547a4f85b72a439076a/src/main/java/com/marvelsnap/controller/MainController.java#L33-L35

- **Interfacce Funzionali e Callback (Runnable):**

  Impiego dell'interfaccia funzionale Runnable per gestire il ritorno al menu principale dal GamePanel. Questa tecnica permette un disaccoppiamento totale: il pannello non sa come tornare al menu, si limita a eseguire la "ricetta" (callback) fornita dal controller.

  - **Permalink:**
    https://github.com/genna-12/marvel-snap-java/blob/0578ed03acd817f91374e547a4f85b72a439076a/src/main/java/com/marvelsnap/view/GamePanel.java#L32

    https://github.com/genna-12/marvel-snap-java/blob/0578ed03acd817f91374e547a4f85b72a439076a/src/main/java/com/marvelsnap/controller/MainController.java#L66-L72

- **Polimorfismo tramite Classi Anonime:**

  Nella CardFactory, ho utilizzato classi anonime per definire il comportamento specifico del metodo onReveal per carte complesse (es. Professor X o Iron Man). Questo permette di iniettare logica comportamentale diversa in oggetti dello stesso tipo senza dover creare decine di file .java superflui.

  - **Permalink:**
    https://github.com/genna-12/marvel-snap-java/blob/0578ed03acd817f91374e547a4f85b72a439076a/src/main/java/com/marvelsnap/util/CardFactory.java#L117-L124

- **Integrazione di Librerie di terze parti (FlatLaf):**

  Utilizzo di FlatLaf per modernizzare l'interfaccia Swing. L'integrazione ha richiesto la gestione delle dipendenze tramite Gradle e l'inizializzazione del LookAndFeel (riportato nel permalink) prima del caricamento dell'EDT (Event Dispatch Thread).

- **Permalink:**
  https://github.com/genna-12/marvel-snap-java/blob/0578ed03acd817f91374e547a4f85b72a439076a/src/main/java/com/marvelsnap/main/Main.java#L20

- **Testing della GUI con Librerie Esterne (AssertJ Swing):**

  Utilizzo di AssertJ Swing per automatizzare i test d'intefaccia. Questa libreria permette di simulare l'interazione umana e verificare che il MainController aggiorni correttamente il CardLayout del MainFrame.

  - **Permalink:**
    https://github.com/genna-12/marvel-snap-java/blob/9b096ad29ff02e6bb4993e49641b3efd58c57a9e/src/test/java/MainNavigationTest.java

- **Utilizzo di API JDK avanzate (CardLayout)**

  Uso strategico del CardLayout all'interno del MainFrame per gestire la macchina a stati dell'applicazione. Questa soluzione è preferibile alla gestione di multipli JFrame poiché mantiene la consistenza della memoria e delle risorse grafiche durante l'intera esecuzione.

  - **Permalink:**
    https://github.com/genna-12/marvel-snap-java/blob/9b096ad29ff02e6bb4993e49641b3efd58c57a9e/src/main/java/com/marvelsnap/view/MainFrame.java#L27

3.2.3 Note di sviluppo di Lumini Marco

- **Interfaccia Funzionale Runnable**

  Ho utilizzato l'interfaccia funzionale Runnable per passare al CardPanel il compito da eseguire al momento dei click invece di forzarlo a conoscere i dettagli della logica di gioco. Questa feature ha permesso di disaccoppiare completamente la view dal controller, rendendo il componente grafico riutilizzabile.

  - **Permalink:**
    https://github.com/genna-12/marvel-snap-java/blob/feature/p3-player/src/main/java/com/marvelsnap/view/CardPanel.java#L115

3.2.4 Note di sviluppo di Spazzoli Paolo

- **Uso di Lambda Expression**

  Utilizzata in BoardPanelTest sfruttando l’interfaccia funzionale Callable.

  - **Permalink:**

    https://github.com/genna-12/marvel-snap-java/blob/1cdf89b59e1c5abfd731f34900dac714e39a01a1/src/test/java/BoardPanelTest.java#L35

- **Testing della GUI con AssertJ-Swing:**

  Ho utilizzato questa libreria in BoardPanelTest per simulare l'interazione di un utente con l'interfaccia grafica, rendendo il test completamente automatico.

  - **Permalink:**

    https://github.com/genna-12/marvel-snap-java/blob/1cdf89b59e1c5abfd731f34900dac714e39a01a1/src/test/java/BoardPanelTest.java

___

Capitolo 4: Commenti finali

4.1 Autovalutazione e lavori futuri

Del Moro Matteo

- **Ruolo nel gruppo:** Mi sono occupato principalmente di gestire il core del gioco, ovvero il Model. Ho gestito il flusso della partita, l'avanzamento dei turni e l'assegnamento dell'energia ad ogni turno. Ho anche implementato una parte del Controller, ovvero il GameController, che osserva l'avanzamento del gioco e comunica con la View. Essendo un utente con background di programmazione non molto vasto, ho dedicato molto tempo all'implementazione di una logica corretta e rigorosa.
- **Punti di forza:**
  - **Utilizzo di una macchina a stati:** L'utilizzo di una macchina a stati (InputState) all'interno del GameController semplifica molto i controlli sulle azioni dell'utente.
  - **Gestione del reset della partita:** Un risultato ottimale è stato il raggiungimento della persistenza del software tramite reset dinamico. Il reset delle variabili e l'eliminazione dinamica degli Observer permettono di effettuare più sessioni consecutive senza dover riavviare ogni volta l'intera applicazione.
- **Punti di debolezza:**
  - **Poco utilizzo di feature avanzate:** Per garantire stabilità e facilità di lettura del codice, ho preferito utilizzare costrutti semplici, limitando l'utilizzo di strutture complesse a una LambdaExpression nel GameController. Un approfondimento futuro sulle feature avanzate del JDK potrebbe ottimizzare ulteriormente il codice.
- **Lavori futuri:**
  - **Implementazione del Multiplayer:** Per ora si può giocare sullo stesso calcolatore (Modalità Hot-Seat). Si potrebbe cercare di rendere le partite multiplayer online.
  - **Implementazione Agente Bot contro cui giocare:** Si potrebbe anche implementare un bot che giochi contro l'utente. Data la separazione di logica e View, questo bot potrebbe interagire con la logica e giocare la sua partita.

___

Gennari Lorenzo

- **Ruolo nel gruppo:** In questo gruppo mi sono occupato della View e del Controller, ma dato il mio background precedente (trasferimento da altro ateneo e diploma tecnico), ho finito per fare da architetto/coordinatore tecnico e "mantainer". Oltre a scrivere la mia parte, ho impostato la

struttura del progetto su git con Gradle e mi sono assicurato che i pezzi di Model scritti dai miei compagni si integrassero bene con l'interfaccia. Fondamentalmente ho fatto da ponte tra il codice puro e l'interfaccia utente, risolvendo i bug che nascevano quando mettevamo insieme i vari moduli.

- **Punti di forza:**
    - **Architettura e Debug:** Essendo già abituato a lavorare su progetti Java, sono riuscito a risolvere velocemente problemi ostici come i glitch grafici e i blocchi del thread principale (EDT).
    - **Automazione:** L'aver introdotto AssertJ Swing ha alzato la qualità del progetto. Ci ha permesso di lavorare più velocemente evitando l'utilizzo della classe Robot per il test delle GUI che sarebbe stato di gran lunga più verboso.
    - **Coordinamento e Design:** Ho promosso all'interno del team l'adozione del pattern MVC (Model-View-Controller) per garantire una netta separazione tra la logica di gioco e l'interfaccia Swing. Sin dalle prime fasi, mi sono occupato della definizione dello scaffolding del progetto e della creazione di stub per le classi non ancora implementate, partendo da un diagramma UML preliminare ma esaustivo. Questo approccio ha permesso ai membri del gruppo di lavorare in parallelo senza blocchi. Abbiamo prediletto una metodologia di sviluppo iterativa (più vicina all'Agile) rispetto al classico modello Waterfall, permettendoci di integrare e testare le feature man mano che venivano completate, riducendo drasticamente i tempi di debugging finale.

Punti di debolezza:

Delega: A volte per fare prima e rispettare le scadenze, ho preferito correggere io stesso i bug nel codice degli altri invece di aspettare che lo facessero loro. Questo ha velocizzato i tempi ma a scapito di un confronto più approfondito.

Estetica: Mi sono concentrato talmente tanto sul far funzionare bene il tutto che ho trascurato un po' l'estetica. L'app è solida e gira bene, ma graficamente avrei potuto aggiungere più effetti visivi se non avessi passato così tanto tempo a fare fix cross-team.

Lavori futuri: Il progetto è un'ottima base. Se dovessi continuare, la prima cosa sarebbe passare a un'architettura Client-Server con i Socket, per rendere il gioco davvero multiplayer.

___

Lumini Marco

- **Ruolo nel gruppo:** Mi sono dedicato alla parte del Model riguardante la gestione del giocatore curando le meccaniche di gioco ad esso legate come la pescata di una carta dal proprio deck o la giocata di una carta dalla propria mano e la gestione dei dati utili ad esse durante il corso della partita. Relativamente alla view mi sono occupato della realizzazione della grafica delle carte con l'aiuto del collega Gennari e del pannello della mano per contenere le carte. Inoltre ho creato una breve e semplice guida per l'utente che si interfaccia per le prime volte con marvel snap per facilitare la comprensione del gioco.
- **Punti di forza:** ho privilegiato la semplicità e la solidità del codice e grazie a questo non sono stati necessari eccessivi debug e in generale gli errori sono stati gestiti dal team senza troppe difficoltà. Questo approccio dunque ha permesso un lavoro efficiente e la produzione di un codice robusto e manutenibile.
- **Punti di debolezza:** Nella realizzazione della componente grafica avrei potuto concentrarmi un po' di più sulla parte visiva per provare ad implementare delle animazioni o aggiungere delle immagini

per contraddistinguere le carte dei vari personaggi. Inoltre se da una parte ho tenuto il codice ad un livello più semplice e sicuramente più nelle mie corde dall'altra potrei aver peccato in termini di prestazioni e con funzionalità più avanzate avrei potuto ottimizzare maggiormente il codice.

- **Lavori futuri:**
    - **Implementazione meccanica dello snap e ranking:** sarebbe stimolante provare ad aggiungere una classifica di giocatori. Essa si baserebbe sull'acquisizione di cubi che verrebbero assegnati al termine di ogni partita. Tramite uno snap ogni giocatore potrebbe raddoppiare l'ammontare di cubi in palio durante ogni partita in base ad uno svolgimento favorevole o se si sente in vantaggio per portare a casa più punti e scalare più in fretta la classifica. Anche se questo probabilmente comporterebbe anche implementare una versione multiplayer del programma.
    - **Aggiunta di nuove carte:** si potrebbero aggiungere altre carte presenti effettivamente nel gioco o addirittura crearne di nuove da zero con effetti più complicati per creare anche nuovi archetipi per rendere ancora più godibile l'esperienza di gioco e raggiungere una versione ancora più fedele all'originale.

___

Spazzoli Paolo

- **Ruolo nel gruppo:** in questo lavoro mi sono dedicato alla parte del Model che riguarda la creazione e gestione delle Location e all'implementazione della rispettiva parte grafica. Essendo il componente dotato di conoscenza più approfondita delle regole e delle meccaniche di Marvel Snap, ho cercato di guidare il gruppo verso una resa il più possibile coerente con il gioco delle varie funzionalità implementate.
- **Punti di forza:** credo di aver ottenuto un'implementazione abbastanza semplice e ordinata, la mia esperienza con il gioco mi ha inoltre permesso di trovare velocemente bug ed elementi poco funzionali. Il lavoro fatto sulle Location inoltre, credo che ne faciliti molto l'aggiunta di nuove.
- **Punti di debolezza:** avendo prediletto una programmazione semplice, ho fatto poco ricorso a elementi di programmazione più avanzati, che avrebbero reso il codice ancora più conciso ed efficiente. Riconosco inoltre che avrei potuto implementare un numero maggiore di test, per esempio sulle Location più "particolari".
- **Lavori futuri:** la direzione verso cui sarebbe interessante portare il progetto sarebbe, a mio parere, lo sviluppo dell'interfaccia grafica, aggiungendo immagini diverse per ciascuna carta e per ciascuna Location e animazioni. SI può poi pensare di ridurre ancora la differenza tra questo gioco e l'originale, implementando elementi che per mancanza di tempo non abbiamo aggiunto, come la meccanica dello Snap, la presenza di un timer per ogni turno, o la meccanica della priorità (che permette alle carte del giocatore che il turno precedente stava vincendo più location di essere scoperte per prime nel turno corrente, aggiungendo un ulteriore livello di complessità e interazione).

___

Appendice A: Guida utente

1. **Menu:** Cliccare su "Nuova Partita" per iniziare. Inserire i nomi dei giocatori e scegliere il mazzo.
2. **Partita:**
    - Il Giocatore 1 vede la propria mano in basso.

- Cliccare su una carta (diventa verde) e poi cliccare su una delle tre Location per giocarla. Prima di fare ciò, verificare se la propria energia è sufficiente per giocare la carta(La propria energia è in alto a destra, mentre l'energia richiesta dalla carta è nell'angolo in alto a sinistra di essa).
- Premere "Termina Turno" quando finito.
- Appare una schermata per nascondere le carte mentre i giocatori si scambiano il posto.

3. **Fine Partita:** Al termine del 6° turno, appare un popup con il nome del vincitore.

Appendice B: Esercitazioni di laboratorio

P1 - matteo.delmoro@studio.unibo.it

- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=206731
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=207193
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=207921
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=208718
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=209589
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=210617
- Laboratorio 12: https://virtuale.unibo.it/mod/forum/discuss.php?d=211539