Relazione

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il software mira a essere una riproduzione del classico gioco di carte UNO con alcune modifiche e aggiunte rispetto le regole classiche
In particolare la modifica più rilevante riguarda la composizione del tavolo da gioco: la partecipazione umana è riservata a un singolo utente, mentre i restanti tre posti sono affidati a giocatori automatici gestiti in tutto e per tutto dal computer, per un totale di **quattro partecipanti per ogni sessione**.

Intelligenza Artificiale e Personalità

Il sistema deve gestire avversari virtuali (Bot) caratterizzati da comportamenti eterogenei. Ogni Bot presenta una personalità che ne influenza lo stile di gioco:

- **Bot Fallax (Il Baro):** Rappresenta un'anomalia nel sistema, possedendo l'abilità di accedere a informazioni normalmente nascoste (le carte nella mano del giocatore umano) e di agire strategicamente per ostacolarlo.
- **Bot Implacabilis (L'Aggressivo):** La sua logica predilige l'utilizzo sistematico di carte sanzionatorie o di attacco per danneggiare gli avversari.
- **Bot Fortuitus (Il Casuale):** Agisce senza una strategia di lungo termine, selezionando casualmente una mossa valida tra quelle disponibili.

Configurazione e Svolgimento

Ogni sessione di gioco è influenzata all'avvio da un **Evento Modificatore**: un fattore casuale che altera una regola e che persiste fino al termine della partita.

La dotazione iniziale prevede la distribuzione di 7 carte per ciascun giocatore. Il campo di gioco è costituito dal **Mazzo di Pesca** (coperto) e dalla **Pila degli Scarti**, inizializzata con una prima carta casuale.

Il flusso di gioco si svolge seguendo una sequenza di turni. Il giocatore attivo deve scartare una carta dalla propria mano rispettando i **criteri di compatibilità** con la carta in cima alla Pila degli Scarti:

1. **Corrispondenza di Colore:** La carta giocata ha lo stesso colore.
2. **Corrispondenza di Valore/Simbolo:** La carta ha lo stesso numero o rappresenta la stessa azione.
3. **Jolly:** Le carte speciali nere (Wild) ignorano i vincoli di colore e sono sempre giocabili.

Se un giocatore non possiede carte compatibili, è obbligato a pescare una carta dal Mazzo di Pesca, terminando immediatamente il proprio turno. La condizione di vittoria è raggiunta dal primo giocatore che esaurisce tutte le carte della propria mano.

Meccanica Speciale: Stacking (Accumulo)

Questa regola avanzata denominata **Stacking** altera la gestione delle carte sanzionatorie (+2 e +4). Questa meccanica trasforma l'effetto di penalità da passivo a dinamico, permettendo la "difesa".

1. **Attivazione:** Quando un giocatore subisce un +2 o un +4, può evitare la pesca immediata giocando una carta compatibile.
2. **Gerarchia di Compatibilità:**
   - Contro un **+2**: È possibile rispondere con un altro +2 (qualsiasi colore).
   - Contro un **+4**: È possibile rispondere esclusivamente con un altro +4.

3. **Accumulo e Risoluzione:** Se la difesa ha successo, il valore della penalità si somma al precedente e il "debito" di carte passa al giocatore successivo. La catena si interrompe quando un giocatore non possiede una carta valida per rispondere: tale giocatore dovrà pescare la somma totale delle carte accumulate e perderà il turno.

> *Esempio di scenario Stacking:*
>
> Il Giocatore A gioca un +2. Il Giocatore B risponde con un +2 (Totale accumulato: 4). Il Giocatore C non possiede +2: subisce la penalità pescando 4 carte e salta il turno. Il gioco riprende dal Giocatore D.

Entità del Gioco: Le Carte Speciali

Oltre alle carte numeriche standard, il sistema presenta due categorie di carte funzionali che scatenano effetti specifici:

Carte Azione (Colorate)

- **Cambia Giro (Reverse):** Inverte l'ordine dei turni (da orario ad antiorario e viceversa).
- **Stop (Skip):** Priva il giocatore successivo del diritto di agire.
- **+2 (Draw Two):** Impone al successivo la pesca di 2 carte e la perdita del turno (salvo attivazione Stacking).

Carte Jolly (Nere)

- **Cambio Colore (Wild):** Giocabile sempre. Permette di ridefinire il colore attivo per il turno successivo.
- **+4 (Wild Draw Four):** Permette di scegliere il colore, impone la pesca di 4 carte e la perdita del turno al giocatore successivo. In regime di Stacking, può essere contrastata solo da un altro +4.

Eventi Creati

Il sistema introduce una serie di varianti alle regole standard, denominate "Eventi", che vengono selezionate all'avvio della partita. Queste modalità alterano il comportamento di specifiche carte o la composizione del mazzo.
Di seguito sono elencati gli eventi:

- **Partita Standard (Standard Game)**
  Rappresenta la modalità classica. Il mazzo è composto secondo le regole tradizionali e le carte mantengono i loro effetti originali senza alterazioni.
- **Raddoppio (Double Draws)**
  In questa modalità, il "potere" di pesca delle carte malus è drasticamente amplificato. Tutte le carte che impongono la pesca subiscono un raddoppio del loro valore nominale:
  - Il **+2** costringe il destinatario a pescare **4 carte**.
  - Il **Jolly +4** costringe il destinatario a pescare **8 carte**.
- **Zero Inverso (Reverse Zeros)**
  Questa variante attribuisce un valore funzionale alle carte numeriche di valore **0**. Oltre al loro valore di faccia, tutti gli "Zeri" ereditano l'effetto della carta azione "Cambia Giro" (Reverse). Giocare uno zero inverte immediatamente la direzione del flusso di gioco (da orario ad antiorario o viceversa).
- **Sette di Blocco (Block Sevens)**
  Similmente alla variante precedente, questa modalità potenzia le carte numeriche di valore **7**. Ogni volta che viene giocato un "Sette", esso attiva l'effetto della carta "Stop" (Skip), saltando il turno del giocatore successivo.
- **Caos Totale (Total Chaos)**
  Rappresenta la sfida definitiva per l'adattabilità dei giocatori. In questo scenario, **tutte le regole speciali sopra descritte sono attive simultaneamente**.
  - I +2 e +4 valgono doppio.
  - Gli Zeri invertono il giro.
  - I Sette saltano il turno.

Requisiti funzionali

- **Gestione della partita e delle regole:** Il sistema deve garantire la corretta applicazione delle regole del gioco UNO, inclusa la validazione delle mosse (corrispondenza per colore/simbolo), la gestione del mazzo di pesca/scarti e l'applicazione degli effetti delle carte speciali.
- **Intelligenza Artificiale eterogenea:** Il sistema deve supportare avversari controllati dal computer (BOT) caratterizzati da diverse strategie comportamentali.
- **Gestione eventi casuali:** l'applicazione deve essere in grado di iniettare e gestire modificatori globali alle regole che alterano il flusso della partita.
- **Sistema di vittoria:** L'applicazione deve essere in grado di determinare la condizione di fine partita.
- **Interfaccia utente interattiva:** Il sistema deve fornire una visualizzazione dello stato del gioco (mano del giocatore, ultima carta scartata, stato degli avversari) e permettere l'interazione per giocare carte o pescare.

Requisiti non funzionali

- **Reattività dell'interfaccia:** L'interfaccia grafica deve rispondere alle azioni dell'utente in tempo reale, senza blocchi percettibili durante il calcolo delle mosse dei BOT.
- **User Experience (Timing dei BOT):** Le mosse dei giocatori automatici non devono essere istantanee, ma devono prevedere un ritardo artificiale per permettere all'utente umano di seguire il flusso di gioco e comprendere cosa sta accadendo.
- **Portabilità:** Il software deve essere eseguibile su diverse piattaforme desktop che supportino l'ambiente di esecuzione scelto.
- **Robustezza:** Il gioco non deve raggiungere stati inconsistenti o bloccarsi a fronte di input errati dell'utente o sequenze di carte impreviste.

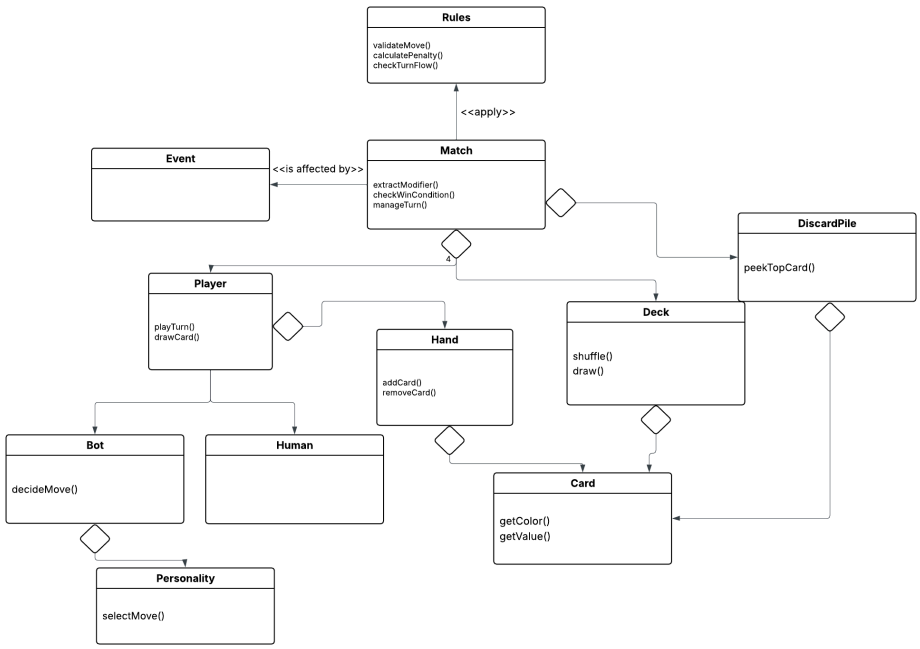
1.2 Modello del Dominio

Il sistema dovrà gestire l'entità **Partita**, intesa come sessione di gioco a turni. Ciascuna Partita aggrega un gruppo predefinito di quattro **Giocatori**. Tali Giocatori potranno essere di natura **Umana** oppure **Artificiale (Bot)**.

Ciascun Giocatore Artificiale è caratterizzato da una specifica **Personalità** (come *Fallax, Implacabilis* o *Fortuitus*), responsabile di determinare le scelte strategiche del soggetto. Ogni Giocatore dovrà possedere e gestire una **Mano**, composta da un insieme variabile di **Carte**.

Le Carte rappresentano l'elemento fondamentale di scambio: esse vengono estratte da un **Mazzo di Pesca** e, una volta giocate, vanno a costituire la **Pila degli Scarti**. Ciascuna Carta è caratterizzata da un colore Rosso, Giallo, Verde, Blu, oppure Nero e da un valore (numero o simbolo).

La Partita sarà inoltre influenzata da un **Evento Modificatore** determinato all'inizio della partita che altera le **Regole** per tutta la durata del gioco. La condizione di vittoria sarà determinata dall'esaurimento delle carte nella Mano del Giocatore.



Capitolo 2

2.1 Architettura

L'architettura di **Primus** adotta il pattern architetturale **Model-View-Controller (MVC)**, strutturato per gestire in modo efficace la natura sequenziale e a stati finiti di un gioco di carte a turni.
Il sistema è orchestrato da un **Game Loop** centrale, gestito dal Controller, che coordina l'interazione tra la logica di dominio (Model) e l'interfaccia utente (View).

Le componenti principali che definiscono i vertici del triangolo MVC sono identificate dalle seguenti interfacce chiave:

1. **Model (`GameManager`):** Rappresenta il nucleo della logica. Il Model è completamente agnostico rispetto all'interfaccia utente e non possiede riferimenti diretti alla View o al Controller.
2. **View (`GameView`):** Definisce il contratto per la visualizzazione dello stato di gioco e la cattura degli input utente. La View è passiva: non prende decisioni di gioco, ma si limita a renderizzare le informazioni ricevute e a notificare il Controller delle azioni dell'utente (click su carte o mazzi).
3. **Controller (`GameController`):** Agisce come regista del sistema. Implementa il ciclo di vita della partita e media il flusso dati.

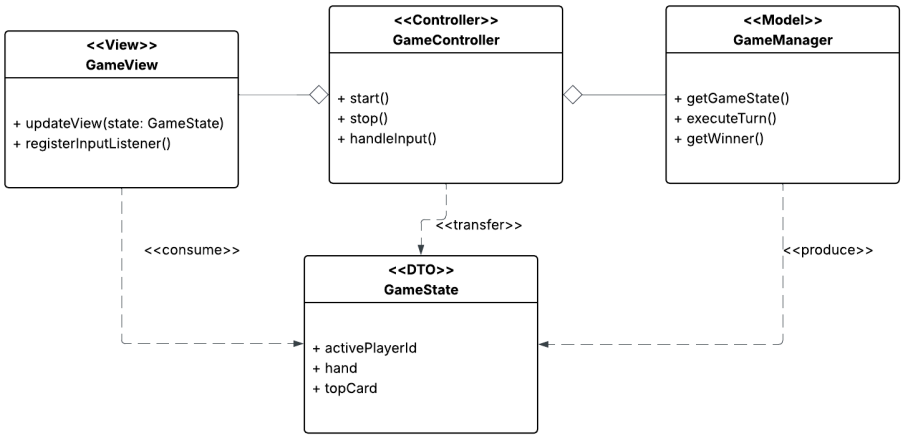
Interazione e Flusso Dati
La dinamica del sistema si basa su una chiara separazione delle responsabilità durante il ciclo di gioco:

- **Dal Model alla View (Aggiornamento):** Per garantire un disaccoppiamento forte, il Model non comunica direttamente con la View. È il Controller che, ad ogni iterazione del Game Loop o dopo ogni mossa, estrae lo stato corrente dal `GameManager` e lo inoltra alla `GameView`.
  Per il trasferimento dei dati è stato introdotto un oggetto specifico: il `GameState`. Questo agisce come un **DTO** immutabile. Passando alla View una "fotografia" dello stato invece che il riferimento al Model stesso, si garantisce che la View non possa inavvertitamente alterare la logica di gioco, favorendo la sicurezza e la manutenibilità.
- **Dalla View al Controller (Input):** L'interazione segue un approccio guidato dagli eventi. La `GameView` ha la responsabilità esclusiva di intercettare le azioni dell'utente (come la selezione di una carta o la richiesta di pescare) e di inoltrarle al Controller.

Questo meccanismo di delega garantisce che la View rimanga un componente puramente passivo: Quando l'utente umano agisce, la View notifica il Controller, senza conoscere i dettagli di come quell'input verrà processato.

- **Gestione dei Turni :**
  Il `GameController` ricopre un ruolo fondamentale nella gestione dell'eterogeneità dei giocatori. All'interno del loop principale, il Controller interroga il Model per ottenere il giocatore corrente.
  Il Controller distingue la strategia di attesa:
  - **Nel caso di Bot:** Il Controller invoca direttamente gli algoritmi decisionali del Bot e applica la mossa al Model.
  - **Nel caso del giocatore Umano:** Il Controller sospende il loop logico e attende l'evento di input proveniente dalla View, mantenendo l'interfaccia reattiva.

Con questa architettura, la sostituzione della View (ad esempio, passando da Swing a un'interfaccia testuale o Web) impatterebbe esclusivamente l'implementazione di `GameView`, lasciando inalterati il `GameManager` e la logica di gestione del `GameController`. Il Controller, infatti, interagisce con la View esclusivamente tramite l'interfaccia, ignorando i dettagli implementativi della libreria grafica sottostante.



2.2 Design di dettaglio

Luca Gianelli

Personalità dei bot

Descrizione del problema

Uno dei requisiti funzionali del sistema è la presenza di giocatori artificiali (Bot) in grado di partecipare alle partite in modo autonomo. Il problema principale risiede nella necessità di supportare **comportamenti eterogenei**: il gioco deve prevedere Bot con diverse personalità rispettando i principi di buona programmazione.

Implementare queste logiche all'interno di un'unica classe `Bot` avrebbe reso il codice monolitico, difficile da testare e chiuso all'estensione non rispettando nessun principio.

Inoltre, il processo decisionale di un giocatore in UNO si divide in due azioni distinte e non sempre sequenziali:

1. **Selezione della carta:** Scegliere quale carta giocare dalla mano.
2. **Selezione del colore:** Nel caso di carte Jolly (Wild), decidere il nuovo colore di gioco.

   Queste due logiche sono concettualmente distinte: un Bot potrebbe essere strategicamente abile nel giocare le carte, ma casuale nella scelta dei colori. Il sistema doveva quindi permettere di combinare queste abilità in modo flessibile.

Soluzione proposta

La soluzione adottata segue il principio della **Composition over Inheritance**. Invece di utilizzare l'ereditarietà (creando sottoclassi come `RandomBot`, `AggressiveBot`), che avrebbe portato a una rigida gerarchia di classi, si è scelto di modellare la classe `Bot` come un contenitore ("Context") che delega le decisioni a componenti esterni intercambiabili.

Nello specifico, sono state definite due interfacce separate: `CardStrategy` per la logica di gioco e `ColorStrategy` per la gestione della scelta del colore in caso di Jolly. Il `Bot` concreto riceve le implementazioni di queste strategie al momento della costruzione.

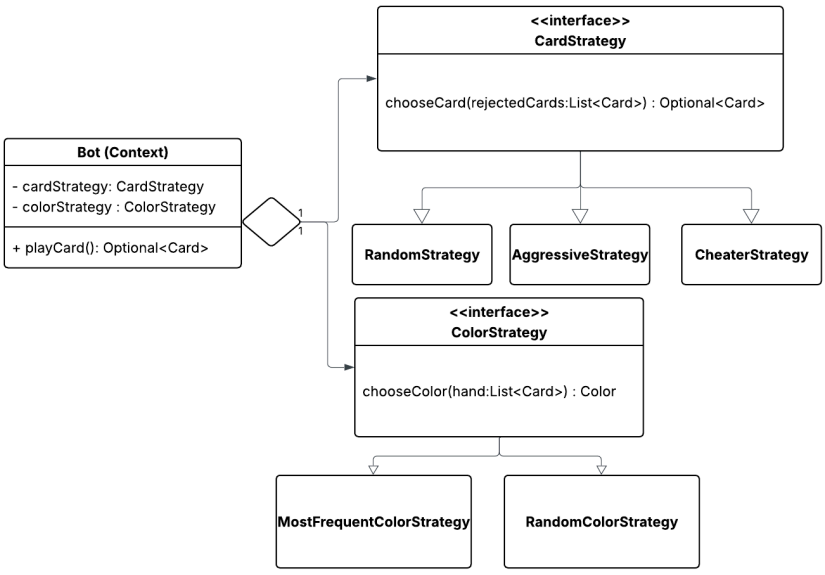
Analisi della soluzione proposta

La scelta di disaccoppiare completamente il "corpo" del Bot (la classe `Bot`) dal suo "cervello" (le strategie) garantisce la massima estensibilità (rispetto dell'**Open/Closed Principle**): è possibile aggiungere nuove personalità senza modificare la classe `Bot`. Inoltre, l'utilizzo di due strategie distinte permette un'elevata combinabilità (N strategie carte, M strategie colori generano N * M tipi di bot). L'unico svantaggio è un lieve aumento della complessità nella fase di istanziazione dell'oggetto `Bot`, che richiede il passaggio di più dipendenze (problema risolto tramite l'uso di una *Factory*, discussa successivamente).

Analisi delle alternative scartate:

- **Strategia Unificata:** Si è valutata l'idea di includere la scelta del colore all'interno di `CardStrategy`. Questa ipotesi è stata scartata perché violava il **Single Responsibility Principle (SRP)**. Unendo le due logiche, non sarebbe stato possibile riutilizzare l'algoritmo di scelta del colore su bot diversi, obbligando a duplicare codice o a creare classi con troppe responsabilità.
- **Template Method:** Si è considerato l'uso del pattern *Template Method* (definendo una classe base astratta con uno scheletro dell'algoritmo). L'alternativa è stata scartata a favore di *Strategy* per due motivi:
  1. **Staticità del comportamento:** Il *Template Method* si basa sull'ereditarietà, che definisce il comportamento a tempo di compilazione. Cambiare la strategia di un bot a *runtime* (es. un bot che diventa più aggressivo se sta perdendo) sarebbe complesso, richiedendo la sostituzione dell'intera istanza dell'oggetto Bot e la copia del suo stato. Il pattern *Strategy*, basandosi sulla composizione, permette invece di sostituire l'algoritmo al volo semplicemente assegnando una nuova implementazione alla variabile d'istanza.
  2. L'uso di interfacce permette di sfruttare le **Interfacce Funzionali** di Java, rendendo il codice più snello e permettendo la definizione di strategie tramite lambda expression.

Schema UML



Pattern utilizzati

La soluzione descritta è una reificazione del **Strategy Pattern**, applicato qui in una variante doppia per gestire le due diverse responsabilità decisionali.

Context: La classe `Bot` agisce come *Context*. Essa mantiene i riferimenti alle strategie e le invoca durante il metodo `playCard()`, rimanendo agnostica rispetto alla logica eseguita.

Strategy Interfaces:

- `CardStrategy`: È l'interfaccia funzionale che definisce il contratto per la selezione della carta (`chooseCard`).
- `ColorStrategy`: È l'interfaccia funzionale che definisce il contratto per la selezione del colore (`chooseColor`).

Concrete Strategies: Sono le implementazioni specifiche degli algoritmi.

- Per `CardStrategy`: `RandomStrategy` (scelta casuale), `AggressiveStrategy` (scelta basata su pesi offensivi), `CheaterStrategy` (scelta basata sulla mano dell'avversario).
- Per `ColorStrategy`: `RandomColorStrategy` e `MostFrequentColorStrategy` (sceglie il colore più presente nella mano)

Gestione dell'Accesso Privilegiato

Descrizione del problema

L'introduzione di una personalità "Baro" (`CheaterStrategy`) ha sollevato una sfida critica riguardante l'incapsulamento e l'integrità del gioco. Questa strategia necessita di ispezionare lo stato interno di un avversario (principalmente la mano di carte) per calcolare le proprie mosse.
Tuttavia, passare all'oggetto `CheaterStrategy` un riferimento diretto all'oggetto `Player` dell'avversario avrebbe costituito una grave falla di sicurezza difatti l'interfaccia `Player` espone metodi che cambiano lo stato interno (come `playCard()` o `addCards()`). Un errore nella logica del bot o un utilizzo improprio avrebbero potuto permettere al "Baro" non solo di guardare, ma di modificare lo stato dell'avversario, rendendo il sistema inconsistente.
Il problema consisteva nel garantire l'accesso in lettura alle informazioni sensibili impedendo tassativamente qualsiasi operazione di modifica.

Soluzione proposta

La soluzione adottata prevede l'interposizione di un livello di astrazione tra il Bot e la sua vittima. È stata definita un'interfaccia ristretta, denominata `OpponentInfo`, che espone esclusivamente i metodi di lettura necessari all'analisi strategica.
La classe `OpponentInfoImpl` implementa tale interfaccia e agisce come **Wrapper** attorno all'istanza reale del `Player`. Il `CheaterStrategy` riceve nel costruttore solo l'interfaccia `OpponentInfo`.

Analisi della soluzione proposta

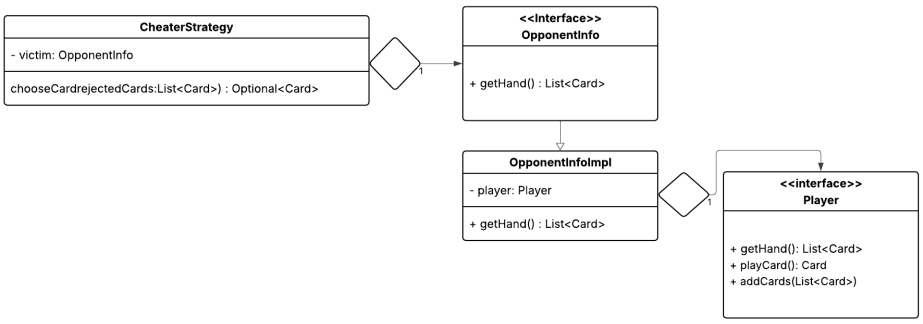
La soluzione adottata si distingue per la garanzia di sicurezza offerta a **tempo di compilazione**. A differenza di un approccio basato su controlli a runtime, l'utilizzo dell'interfaccia `OpponentInfo` rende strutturalmente impossibile per la `CheaterStrategy` invocare metodi mutatori, poiché questi non sono visibili nel contratto definito.

Questo approccio favorisce un netto **disaccoppiamento** architetturale: la strategia dipende esclusivamente da un sottoinsieme minimale e stabile di dati, rimanendo isolata dalla complessità e dall'evoluzione interna della logica del `Player`. Inoltre, l'esplicitazione delle operazioni consentite migliora drasticamente la **chiarezza d'intento** del codice, guidando lo sviluppatore nell'utilizzo corretto delle API ed evitando l'inquinamento dell'autocompletamento dell'IDE con metodi che non dovrebbero essere richiamati.

Analisi delle alternative:

- **Proxy di Protezione:** Si è valutata la creazione di un Proxy che implementasse l'interfaccia `Player` completa, lanciando eccezioni sui metodi di modifica. Questa strada è stata scartata per i seguenti motivi:
  1. Sicurezza a runtime vs compile-time: gli errori verrebbero rilevati solo durante l'esecuzione.
  2. Inquinamento dell'API: l'IDE suggerirebbe metodi (come `playCard`) che in realtà sono proibiti, creando confusione nello sviluppo.

Schema UML



Creazione dei bot

Descrizione del problema

L'adozione del pattern *Strategy* ha notevolmente aumentato la flessibilità del sistema, ma ha introdotto una complessità nella fase di creazione degli oggetti difatti il costruttore della classe `Bot` richiede ora l'iniezione esplicita di due dipendenze distinte (`CardStrategy` e `ColorStrategy`).

Per istanziare una specifica tipologia di giocatore (es. il bot "Implacabilis"), il client dovrebbe conoscere e assemblare manualmente le classi concrete corrette.

Replicare questa logica di assemblaggio in ogni punto del codice in cui è necessario creare una partita comporterebbe una massiccia duplicazione di codice e un forte accoppiamento tra i livelli alti dell'applicazione e le implementazioni concrete delle strategie.

Soluzione proposta

Per risolvere il problema, è stata introdotta un'interfaccia `BotFactory` con la responsabilità esclusiva di incapsulare la logica di creazione e configurazione dei Bot.

L'implementazione concreta, `BotFactoryImpl`, fornisce metodi (es. `createFortuitus`, `createImplacabilis`) che restituiscono un'istanza di `Player` già completamente configurata con le strategie appropriate.

In questo modo, il client delega alla Factory i dettagli della costruzione, se in futuro si decidesse di modificare il comportamento del bot "Implacabilis" (ad esempio cambiandone la strategia di colore), la modifica sarebbe localizzata esclusivamente all'interno della Factory, senza impattare il resto del sistema.

Analisi della soluzione proposta

La scelta di centralizzare la creazione dei bot tramite una **Factory** risponde primariamente all'esigenza di disaccoppiare la logica di gioco dalla configurazione delle strategie. L'approccio adottato garantisce che il `GameManager` (o qualsiasi altro client) dipenda esclusivamente dall'astrazione `Player`, ignorando completamente quali combinazioni di `CardStrategy` e `ColorStrategy` definiscano una specifica personalità di gioco. Questo eleva il livello di astrazione del codice, rendendo i metodi di creazione auto-esplicativi (es. `createImplacabilis`) e migliorando la leggibilità

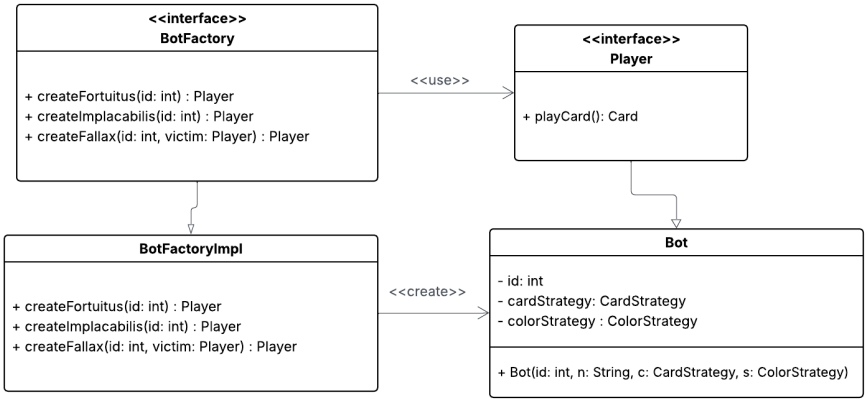
Analisi delle alternative:

- **Istanziazione diretta :** Si è valutato di permettere al client di chiamare direttamente il costruttore del `Bot`. Questa alternativa è stata scartata perché esponeva dettagli implementativi (le classi delle strategie) che non dovrebbero

competere al `GameManager`, rendendo il codice fragile e difficile da manutenere in caso di refactoring delle strategie.

- **Builder Pattern:** Si è considerato l'uso di un *Builder* per costruire il bot passo dopo passo. Tuttavia, dato che le combinazioni di strategie che definiscono le "personalità" del gioco sono fisse e note a priori una Factory con metodi specifici è risultata una soluzione più immediata e meno verbosa rispetto a un Builder generico.

Schema UML



Pattern utilizzati

La soluzione rappresenta una applicazione del pattern **Factory** (nello specifico una **Abstract Factory** che produce diverse varianti di giocatori).

- **Abstract Factory:** L'interfaccia `BotFactory` definisce il contratto per la creazione dei bot.
- **Concrete Factory:** La classe `BotFactoryImpl` implementa i metodi di creazione (`createFortuitus`, `createImplacabilis`). Essa agisce come un **Assemblatore**: incapsula la complessità dell'iniezione delle dipendenze, istanziando le strategie concrete (`CardStrategy`, `ColorStrategy`) e passandole al costruttore del `Bot`.
- **Product:** L'interfaccia `Player` rappresenta l'astrazione del prodotto restituito al client, garantendo che il resto dell'applicazione interagisca con i bot in modo uniforme.

Diego Cozzolino

Gestione Data-Driven e View Dinamica

In questa sezione si analizzano le scelte progettuali relative alla flessibilità della configurazione di gioco e alla logica di presentazione dell'interfaccia grafica.

Configurazione Dinamica del Mazzo (Data-Driven Design)

Problema

Uno dei requisiti di *Primus* era la possibilità di estendere le modalità di gioco (es. introdurre eventi come *Chaos* o *Double Trouble*) senza dover modificare il codice sorgente per ogni nuova variante. Un approccio "hardcoded", in cui le carte vengono istanziate manualmente nel codice (es. `new Card(...)`), avrebbe reso il sistema rigido e difficile da mantenere, violando il principio Open-Closed. Inoltre, era necessario gestire il caricamento di queste configurazioni in modo robusto.

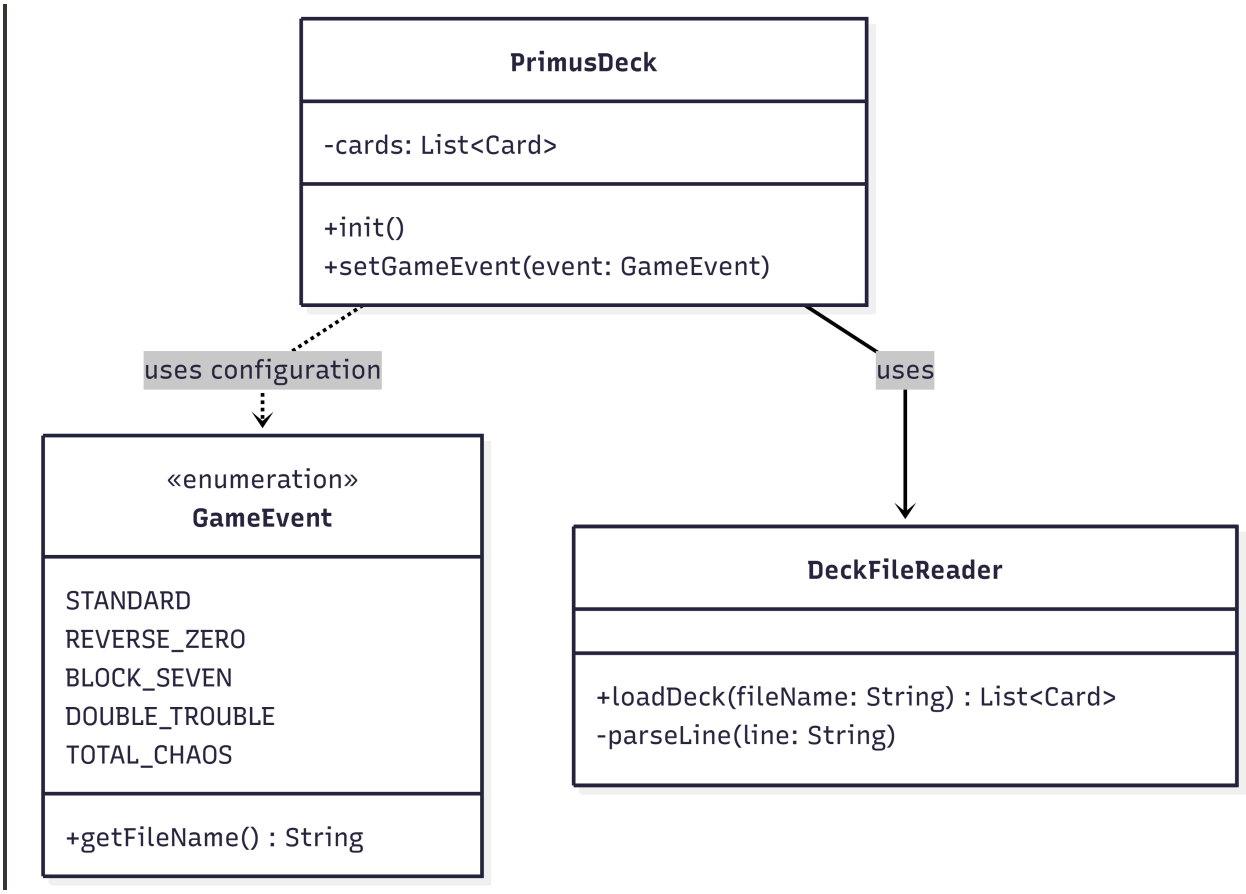
Soluzione

È stata adottata un'architettura **Data-Driven**. La definizione delle carte e delle loro proprietà (valore, colore, effetti speciali, quantità, “potere” di pesca) è stata disaccoppiata dalla logica Java e spostata su file di configurazione esterni in formato

CSV.
Il componente `PrimusDeck` delega la responsabilità del popolamento a una classe specializzata, `DeckFileReader`.

1. **Separazione delle Risorse:** Ogni modalità di gioco (`GameEvent`) è associata a uno specifico file CSV.
2. **Parsing Robusto:** Il `DeckFileReader` utilizza i `ResourceStream` del ClassLoader per accedere ai file. Questo design pattern è stato scelto specificamente per astrarre la posizione fisica del file.

Riferimento UML
Lo schema seguente mostra come `PrimusDeck` dipenda dall'astrazione del `DeckFileReader` e come l'Enum `GameEvent` funga da configuratore.



Algoritmo di Posizionamento Relativo nella GUI

Problema
Il motore di gioco (Model) gestisce i giocatori tramite un indice posizionale assoluto (0, 1, 2, 3) o ID univoci. Tuttavia, in un'interfaccia locale single-player, l'utente si aspetta sempre di visualizzare le proprie carte nella parte inferiore dello schermo ("Sud") per una questione di ergonomia e leggibilità.
Una mappatura diretta 1:1 tra l'ID del giocatore e la posizione a schermo (es. Player 0 sempre a Nord) avrebbe costretto l'utente umano a giocare in posizioni scomode qualora il suo ID non fosse stato quello previsto per il pannello sud.

Soluzione
All'interno della classe `PrimusGameView` (metodo `initGame`), è stato implementato un algoritmo di **rotazione visiva relativa**.
Il sistema non assegna i pannelli in ordine fisso, ma:

1. Identifica l'indice del giocatore umano nella lista fornita dal controller.
2. Assegna forzatamente il pannello `SOUTH` all'umano.

3. Calcola la posizione degli avversari (Bot) iterando la lista in senso orario a partire dall'umano. Questo garantisce che, indipendentemente dall'ID logico (che serve al Model), l'esperienza utente rimanga consistente.

Visualizzazione Reattiva: Eventi e Feedback

Problema
In un gioco con regole mutevoli (Eventi) e penalità improvvise (Malus), l'utente deve ricevere un feedback visivo immediato che spieghi "perché" le regole sono cambiate (es. perché deve pescare 4 carte).

Soluzione
La View è stata progettata per essere totalmente reattiva rispetto allo stato del gioco. Sfruttando i DTO (Data Transfer Objects) forniti dal controller, la View non mantiene alcuno stato interno sulla logica di gioco, ma si ridisegna ad ogni frame (`updateView`).
È stata implementata una logica condizionale nel pannello centrale (`TablePanel`) che:

- Verifica la presenza di flag di `Malus` nel DTO per attivare indicatori di allerta (bordi rossi sul mazzo) solo quando è il turno dell'umano.
- Estrae dinamicamente la descrizione dell'evento corrente (es. "Modalità: Chaos") per visualizzarla nell'HUD, garantendo che l'utente sia sempre consapevole delle regole attive senza bisogno di popup intrusivi.

Davide Zito

Gestione Asincrona dei Turni (Umano vs Bot)

Problema: Il flusso di gioco principale è orchestrato dal `GameController` tramite un Game Loop continuo. Mentre i Bot calcolano e risolvono il proprio turno in modo istantaneo sullo stesso thread del controller, il turno del giocatore umano richiede un'interazione asincrona tramite l'interfaccia grafica. Poiché la view opera sull'Event Dispatch Thread (EDT) tipico della libreria Java SWING, costringere il Game Loop ad attendere l'input umano tramite attese attive (busy waiting) o bloccando l'EDT avrebbe causato il congelamento totale dell'interfaccia rendendo il gioco inutilizzabile.

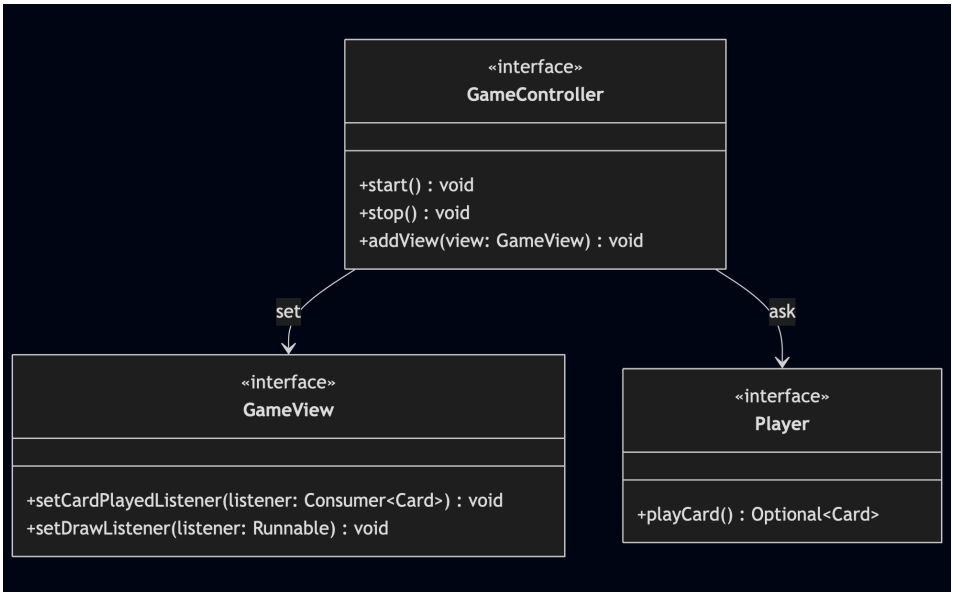
Soluzione: Si è deciso di risolvere la sincronizzazione tra il thread del Game Loop e l'EDT della view utilizzando il costrutto `CompletableFuture` nativo di Java. Quando è il turno del giocatore umano, il `GameController` crea un nuovo `CompletableFuture<Card>` e invoca un metodo bloccante (`.get()`). Questo sospende esclusivamente l'esecuzione del thread del Game Loop, lasciando l'EDT libero di continuare a renderizzare la grafica e processare gli eventi del mouse. Quando il giocatore clicca su una carta valida o sul mazzo, la view invoca una callback (iniettata in fase di setup quando viene aggiunta al controller) che chiama il metodo `.complete(card)` sul Future. Questa operazione risveglia istantaneamente il thread del Game Loop, che preleva la carta e riprende l'esecuzione del gioco passando la mossa al `GameManager`.

Alternativa valutata: All'inizio era stata valutata la possibilità di passare alla view un riferimento del controller e utilizzare in questo modo direttamente i metodi del controller per comunicare l'input del giocatore umano. Questa soluzione però avrebbe accoppiato in modo più stretto controller e view rispetto quella scelta che permette alla view di non conoscere neanche l'interfaccia del controller.

Pattern Utilizzati e Vantaggi: Questa soluzione reifica il pattern **Future / Promise** per la gestione della concorrenza. Questo ha permesso:

- Maggiore leggibilità e pulizia del codice.
- Disaccoppiamento netto: il controller sa che riceverà un dato futuro, ma non gli interessa sapere in che modo o da quale thread della view questo dato verrà generato.
- Assenza di deadlock tra la logica di dominio e il rendering grafico.

- **Resilienza a Race Condition in scenari Multi-View:** L'uso del `CompletableFuture` gestisce in modo sicuro la concorrenza in presenza di viste multiple. Poiché l'invocazione del metodo `.complete(card)` ha effetto esclusivamente alla primissima chiamata, se più interfacce (o molteplici click accidentali dell'utente) tentassero di sbloccare il turno in contemporanea, solo il primo input verrebbe accettato. I tentativi successivi sullo stesso *Future* verrebbero scartati in modo silente, evitando race condition o crash dell'applicazione. Il Game Loop procederebbe immediatamente, calcolando il nuovo stato del tavolo e aggiornando a cascata tutte le viste in ascolto.



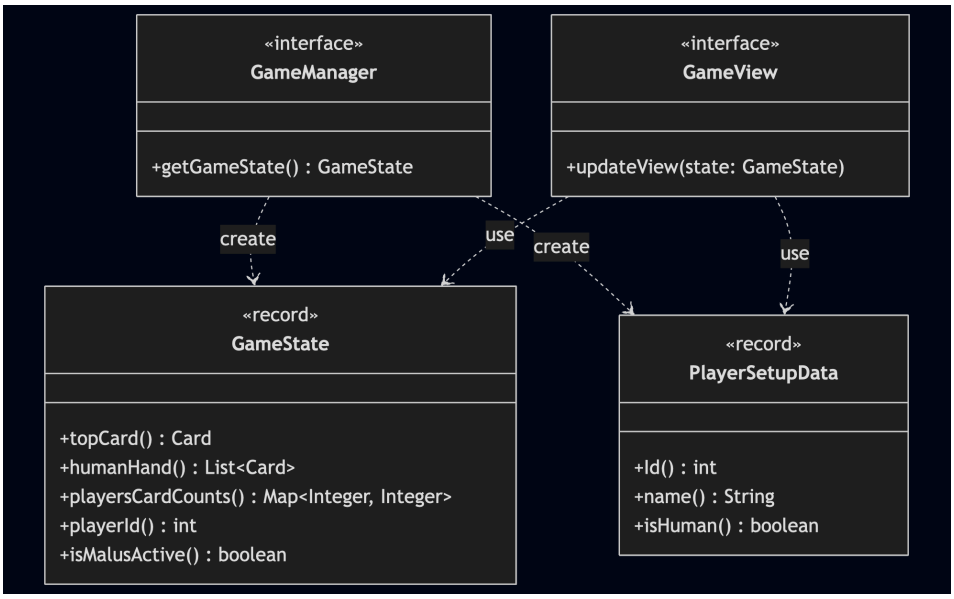
Unidirectional Data Flow e Stateless View

Problema: Durante lo sviluppo della logica di aggiornamento della view, è emerso che aggiornare singole componenti della UI in risposta a eventi specifici (es. rimuovere una carta dalla mano o aggiornare il conteggio di un malus) causava gravi problemi di desincronizzazione visiva (es. carte "fantasma" rimaste a schermo o conteggi errati per gli avversari).

Soluzione: Invece di far mantenere alla view uno "stato" interno da aggiornare in modo incrementale, si è optato per una riprogettazione basata su flussi di dati unidirezionali. Sono stati introdotti dei DTO immutabili, come `GameState` e `PlayerSetupData`. Ad ogni inizio e fine turno, il `GameManager` produce una “fotografia” completa e immutabile del tavolo. La view riceve questo pacchetto e ricostruisce passivamente la propria rappresentazione, senza dover eseguire alcuna logica del model. Mi sono occupato della strutturazione di questo flusso e della fornitura dei dati, garantendo che chiunque implementasse la parte visiva potesse farlo in totale sicurezza, senza rischiare desincronizzazioni.

Pattern Utilizzati e vantaggi: È stato utilizzato il pattern Data Transfer Object (DTO) che ha permesso:

- **Immutabilità**: Essendo il `GameState` e `playerSetupData` record contenenti copie difensive delle collezioni originali, si scongiurano problemi di concorrenza qualora la view leggesse i dati mentre il model li sta modificando. In aggiunta non viene permesso alla view di modificare dati di gioco che sono invece gestiti dal model.
- **Separazione delle responsabilità:** Chi implementa la grafica non deve preoccuparsi delle regole del gioco di UNO, ma solo di mostrare a schermo i dati ricevuti. Inoltre il model “non sa nulla” della o delle view attive che utilizzeranno i suoi dati.



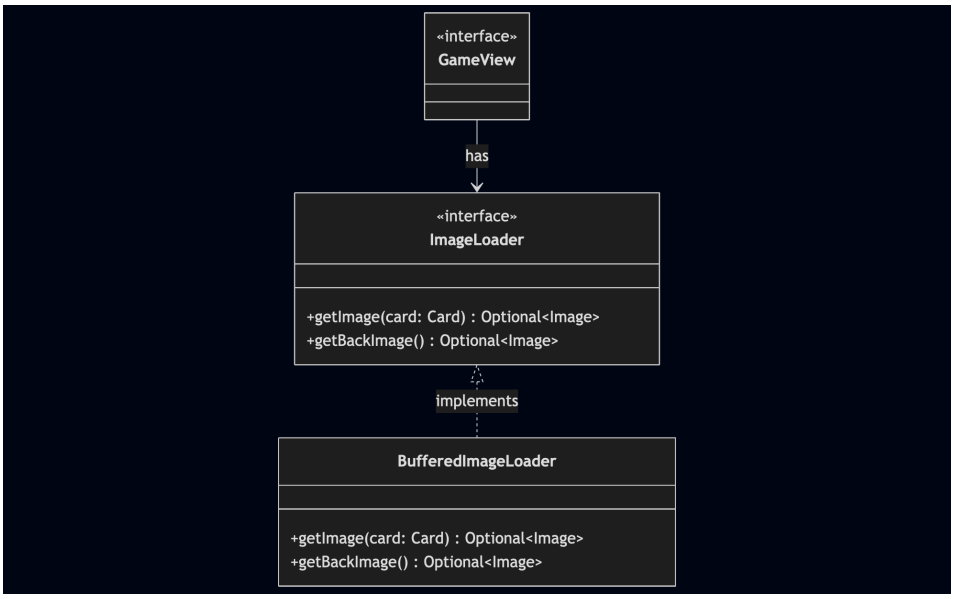
Caching Thread-Safe degli Asset Grafici

Problema: L'applicazione necessita del caricamento di numerosi asset grafici (le carte da gioco). Il caricamento da disco è un'operazione lenta. Considerando la possibilità di poter aggiungere un numero indefinito di view che avrebbero tutte o in parte richiesto gli stessi asset grafici si è optato per un `ImageLoader` unico per l'intero programma che gestisce il caricamento e degli assets.

Soluzione: Si è implementato un `BufferedImageLoader` basato su una `ConcurrentHashMap`. Questa implementazione di `ImageLoader` fornisce caching degli assets in modo thread-safe. Invece di verificare la presenza dell'immagine e successivamente inserirla (operazioni non atomiche), si è fatto ricorso ai metodi thread-safe di Java concorrente (come `computeIfAbsent`). Questo assicura che, nel caso in cui due thread richiedano contemporaneamente lo stesso asset il blocco specifico della mappa venga sincronizzato e l'operazione di I/O avvenga una sola volta, scartando i tentativi ridondanti ed evitando duplicazione.

Vantaggi:

- Garantisce che le istanze pesanti delle immagini vengano condivise e riutilizzate per tutta la durata del ciclo di vita dell'applicazione, minimizzando l'uso della RAM e i tempi di caricamento.
- Facilità l'accesso da parte di tutte le possibili view agli assets grafici



Capitolo 3

Sviluppo

3.1 Testing automatizzato

La suite di test copre:

- **Strategie:** Sono state testate le logiche decisionali dei Bot.
  - `CardStrategyTest` verifica che le strategie selezionino la carta ottimale in base agli scenari di gioco.
  - `ColorStrategyTest` valida gli algoritmi di scelta del colore
- **Ciclo di vita del Bot:** La classe `BotTest` verifica il corretto comportamento dell'entità `Bot`
- **Regole di Gioco:**
  - `ValidatorTest` assicura che il motore di validazione accetti solo mosse legali e gestisca correttamente le difese contro carte penalità.
  - `SanctionerTest` verifica la corretta accumulazione delle penalità e il reset dello stato.
  - `SchedulerImplTest` verifica la gestione e del cambio ordine turni.
- **Pattern Creazionali (Factory):**
  - `BotFactoryTest`: il test ispeziona i campi privati dei Bot creati per garantire che la Factory inietti le dipendenze corrette
- **Manager:** Sono state testate le principali situazioni di gioco che il manager deve gestire
  - `GameManagerImplTest`
- **Gestione del Mazzo e I/O:**
- `DeckFileReaderTest`: Verifica la robustezza del parser CSV. I test assicurano che le configurazioni di gioco vengano caricate correttamente, validando il formato delle righe (Colore, Valore, Quantità) e garantendo che file inesistenti o corrotti sollevino le eccezioni appropriate senza causare crash inattesi.
- `PrimusDeckTest`: Valida la logica core del mazzo di gioco. In particolare, verifica:

- La corretta inizializzazione del numero di carte in base all'Evento selezionato.
- L'integrità dell'operazione di mescolamento (`shuffle`).
- La meccanica di **"Infinite Deck"**: assicura che, qualora il mazzo di pesca si esaurisca, la pila degli scarti venga automaticamente rimescolata per formare un nuovo mazzo, garantendo la continuità della partita.

3.2 Note di Sviluppo

Luca Gianelli

- **Utilizzo di Java Reflection**

  Permalink:
  https://github.com/UniboPrimus/Primus/blob/25c035b726a90ff28383c7b35c7251f4dc6afcbc/src/test/java/com/primus/

- **Utilizzo di Stream e Lambda expressions**:

  Usate pervasivamente. Un esempio Permalink:
  https://github.com/UniboPrimus/Primus/blob/25c035b726a90ff28383c7b35c7251f4dc6afcbc/src/main/java/com/primus
  L37C89

- **Utilizzo di Optional**

  Permalink:
  https://github.com/UniboPrimus/Primus/blob/25c035b726a90ff28383c7b35c7251f4dc6afcbc/src/main/java/com/primus

- **Utilizzo di librerie di terze parti (SLF4J)**

  Usata pervasivamente. Un esempio Permalink:
  https://github.com/UniboPrimus/Primus/blob/25c035b726a90ff28383c7b35c7251f4dc6afcbc/src/main/java/com/primus

Diego Cozzolino

- **Lambda Expressions e Functional Interfaces**
  Utilizzate pervasivamente nella gestione degli eventi dell'interfaccia grafica (Swing) e per l'esecuzione di task nel *Event Dispatch Thread*.
  Permalink:
  https://github.com/UniboPrimus/Primus/blob/25c035b726a90ff28383c7b35c7251f4dc6afcbc/src/main/java/com/primus
- **Utilizzo di librerie di terze parti (SLF4J)**
  Usate pervasivamente *Permalink:*
  https://github.com/UniboPrimus/Primus/blob/25c035b726a90ff28383c7b35c7251f4dc6afcbc/src/main/java/com/primus
- **Utilizzo degli asset grafici da**: https://github.com/sdcirri/Project-JUno

Davide zito

- **Stream** (uso pervasivo):

  https://github.com/UniboPrimus/Primus/blob/25c035b726a90ff28383c7b35c7251f4dc6afcbc/src/main/java/com/primus

- **Lambda Expression** (uso pervasivo):
  https://github.com/UniboPrimus/Primus/blob/25c035b726a90ff28383c7b35c7251f4dc6afcbc/src/main/java/com/primus
- **Optional** (uso pervasivo):

  https://github.com/UniboPrimus/Primus/blob/25c035b726a90ff28383c7b35c7251f4dc6afcbc/src/main/java/com/primus

- **ConcurrentMap e thread-safe**:
  https://github.com/UniboPrimus/Primus/blob/25c035b726a90ff28383c7b35c7251f4dc6afcbc/src/main/java/com/primus
- **CompletableFuture**:
  https://github.com/UniboPrimus/Primus/blob/25c035b726a90ff28383c7b35c7251f4dc6afcbc/src/main/java/com/primus

- **SLF4J** (uso pervasivo): https://github.com/UniboPrimus/Primus/blob/25c035b726a90ff28383c7b35c7251f4dc6afcbc/src/main/java/com/primus

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Luca Gianelli

Il mio contributo all'interno del progetto si è focalizzato principalmente sul **Model**, con responsabilità specifica sullo sviluppo dell'intera architettura dei **Giocatori Automatici (Bot)**, del **Validatore** delle mosse e del **Sanzionatore**.

Ritengo che il punto di forza del mio lavoro risieda nella gestione architetturale dei Bot: l'approccio utilizzato ha permesso di creare avversari con personalità distinte, mantenendo al contempo il codice pulito ed estensibile. Sono inoltre molto soddisfatto della robustezza raggiunta nella validazione delle regole e dell'integrazione della meccanica di difesa **Stacking**, inizialmente prevista come opzionale.

Un aspetto che avrei voluto approfondire maggiormente riguarda l'euristica del Bot *Fallax* (il Baro). Sebbene il suo comportamento sia funzionale, non sono riuscito a perfezionarne l'intelligenza quanto avrei desiderato: la sua capacità di sfruttare la conoscenza delle mani avversarie è sicuramente migliorabile per renderlo strategicamente più letale.

Considerando la solidità della base attuale, intendo proseguire lo sviluppo del progetto lavorando sul raffinamento della logica delle AI esistenti, l'introduzione della modalità torneo e l'aggiunta di nuovi Bot basati su archetipi di onniscienza quali:

- **Caelestis (L'Alleato):** Un bot angelo custode programmato per giocare favorendo l'utente umano.
- **Diabolicus (La Nemesi):** Un bot progettato per massimizzare le sconfitte di qualunque altro giocatore.

Diego Cozzolino

Mi sono occupato dalla parte più “strutturale” fornendo l’implementazione delle carte e delle altre componenti base a cui i miei compagni hanno dovuto fare riferimento sia per costruire la logica di gioco che per il passaggio di informazioni attraverso le varie componenti del progetto, ritengo che la mia parte di progetto sia stata la più semplice “concettualmente” ma con necessità di rimanere scalabile e semplice per permetterne la facile comprensione e utilizzo da parte degli altri. Alcune delle soluzione scelte per implementare le carte sono state di successo come quella di spostare per intero le specifiche delle carte su file csv.

Alcune delle mie scelte però soprattutto sull’implementazione delle carte si sono rivelate, con il senno di poi, poco scalabili inizialmente anche durante la fase di progettazione avevamo immaginato le carte in due gruppi separati (carte normali, carte speciali) con comportamenti diversi, con l’aggiunta degli eventi è stato necessario riadattare il loro funzionamento. Infatti sarebbe stato meglio immaginare le carte come “bianche” per poi venire composte di tutte le loro caratteristiche utilizzando un Builder Pattern, questo avrebbe reso più facile l’implementazione e la scalabilità delle carte.

Ci siamo accorti di una casistica molto limite (anche nel gioco reale) che con le modalità ad eventi (soprattutto quelle che raddoppiano le pesche) c’è la possibilità che se i giocatori hanno tutte le carte in mano ma qualcuno ha bisogno di pescare il mazzo non potrà aggiornarsi perché non ci sono carte disponibile (Il codice solleva solo un eccezione ma non è completamente gestito).

Mi sono anche occupato di una parte della view che nella mia opinione poteva essere meglio ottimizzata e segmentata, ma sono comunque contento del risultato finale, data la facilità di comprensione da parte dell’utente giocante, avrei solo voluto abbellire di più la parte estetica.

Vorrei provare a portare avanti lo sviluppo di questo progetto per poter applicare tutte quelle soluzioni che ne avrebbero migliorato la qualità e la semplicità

Davide Zito

Il mio ruolo è stato mettere insieme parti separate del software aggiungendo tutto quello che serve per farle funzionare. Ho provveduto a creare il gameLoop e a fornire tutti i dati necessari alla view per poter essere sempre aggiornata. Ho reso concrete le regole del gioco utilizzando le parti sviluppate dai miei compagni e unendo le mie.

Ritengo essere valido l'uso dei DTO per il passaggio dei dati alla view mantenendo però solo accesso in lettura senza possibilità di modifiche esterne al manager del gioco.

Sono soddisfatto dell'ottimizzazione del caricamento delle risorse per tutte le possibili view.

Credo di aver separato bene la view dal controller e dal model permettendole di essere sostituita con facilità.

Non sono pienamente soddisfatto di aver permesso al controller di accedere alla reference del Player attualmente giocante siccome è comunque qualcosa che ritengo sarebbe dovuto restare interno al manager, questa scelta è stata fatta per mantenere in un unico posto (il controller) la gestione della richiesta della carta sia ai Bot e al giocatore umano.

4.2 Difficoltà incontrate

Luca Gianelli

Lo sviluppo di **Primus** ha rappresentato per me un banco di prova fondamentale, costituendo la prima vera esperienza significativa nella gestione di un progetto software di media complessità. L'impatto con strumenti professionali come **Git** per il versionamento e **Gradle** per l'automazione della build è stato inizialmente ripido, ma estremamente formativo, essere stato introdotto alle dinamiche del ciclo di vita del software e alla risoluzione dei conflitti pratici rappresenta un bagaglio di competenze che ritengo prezioso per il mio futuro professionale, facendomi apprezzare molto l'impostazione pratica del corso.

Tuttavia, ho riscontrato alcuni problemi riguardo alla stesura della relazione tecnica. Sebbene riconosca l'importanza della documentazione, ho percepito una notevole sproporzione tra l'impegno richiesto per lo sviluppo del codice e quello necessario per soddisfare i requisiti formali del documento finale.
La difficoltà principale che ho riscontrato non è risieduta tanto nella struttura (ben definita dalle specifiche), quanto nel comprendere con precisione **quale livello di dettaglio e quali informazioni specifiche** inserire in ogni sezione per evitare ridondanze o omissioni. Spesso mi sono trovato in difficoltà nel discernere tra ciò che era considerato "dettaglio implementativo" e ciò che meritava di essere elevato a "dettaglio di design", con il timore costante di risultare troppo prolisso o troppo superficiale.

Avrei quindi preferito un approfondimento su questo aspetto durante le lezioni frontali. Ritengo che sarebbe stato estremamente utile integrare delle **simulazioni pratiche** in aula, magari analizzando esempi di stesura corretta partendo da un pezzo di codice o da un diagramma. Avere dei riferimenti concreti su come trasformare l'analisi in documentazione formale avrebbe ridotto drasticamente il disorientamento in fase di stesura, permettendomi di concentrarmi maggiormente sulla qualità dell'analisi architettonica piuttosto che sulla forma.

Diego Cozzolino

Questo progetto è stata la mia prima esperienza nella creazione di un progetto di questa entità da zero, come lo è stato l'utilizzo di Git per la gestione del codice condiviso con gli altri membri del gruppo, sicuramente ho imparato molto nell'utilizzo di queste piattaforme e ho imparato anche quali sono le mie difficoltà e punti di forza nel lavoro di gruppo svolto.

Il progetto assegnato però necessita di alcune modifiche e migliorie durante il periodo di lezione.

- **La relazione** richiesta è complessa e difficile da redigere considerato che richiede quanto meno l'approvazione o preferibilmente la presenza di tutti i partecipanti, già di per se una cosa complicata, inoltre la relazione durante il periodo di lezione è stata spiegata brevemente e in maniera "superficiale" non mostrando realmente la complessità di individuare le varie componenti da elencare al suo interno.
  Nella mia opinione sarebbe più utile e formativo ridurre il carico di esercizi forniti durante le ore di laboratorio (per quanto utili) e utilizzare quel tempo per portare avanti una sorta di progetto di prova in cui viene mostrato come dovrebbe essere fatta una relazione modello permettendo agli studenti di interagire e comprendere meglio la mentalità con cui approcciarsi al lavoro.
  Dato che saper impostare una relazione di questo tipo è fondamentale per un futuro ingegnere

- **La suddivisone dei compiti** è stata una delle parti più complesse del progetto, questo perché, come da indicazioni, va fornita insieme all'idea del progetto momento in cui non è ancora stata stabilità la progettazione dell'applicazione, questo oltre a essere difficile si adatta male al flusso di lavoro di un gruppo giovane e con poca esperienza in questo tipo di attività, nel nostro caso infatti oltre all'abbandono di uno dei membri è capitato spesso che venissero mescolati i compiti preassegnati a causa di vicinanza della scadenza di consegna o semplicemente per difficoltà di tempi dati dalla presenza di appelli o impegni in generale, per quanto comprenda la necessità di suddividere in maniera equa il lavoro sia a scopo didattico che per la valutazione del lavoro svolto secondo me sarebbe più ottimale un approccio più flessibile e adattabile alle necessità del gruppo e del progetto, infatti anche i progetti spesso non si adattano ad una suddivisione così stretta, a causa della struttura degli stessi.

In conclusione trovo che il lavoro assegnato sia stato utile non che formativo per future esperienze ma a mio parere necessità di alcune modifiche per renderlo meglio comprensibile e più gestibile per il gruppo di lavoro.

Davide Zito

La realizzazione di questo progetto si è rivelata un'esperienza estremamente formativa, permettendomi di comprendere le reali dinamiche dello sviluppo software: la collaborazione in team, la gestione del codice condiviso tramite Git/GitHub, l'importanza del testing e cosa significhi architettare e documentare un sistema da zero. Tuttavia, ritengo costruttivo segnalare alcune criticità riscontrate durante il percorso:

- **Carico di lavoro e curva di apprendimento:** La totale libertà concessa in fase di progettazione, seppur stimolante, si è rivelata a tratti spaventosa. Essendo per quasi tutti noi la prima esperienza di sviluppo di questa entità, il rischio di "smarrirsi" è concreto. Considerando che il progetto rappresenta solo metà dell'esame (affiancato da una prova pratica che richiede già notevole rapidità di programmazione), sarebbe stato molto utile un approccio più guidato durante le lezioni, con step intermedi o esempi pratici che accompagnassero lo sviluppo da zero.
- **Vincoli sul Model e Teamwork:** Coordinare i lavori in team durante la sessione, con la sovrapposizione di altri appelli, è stato arduo. In questo contesto operativo, l'obbligo di dover dividere le componenti del *Model* in modo rigorosamente equo fin dall'inizio è risultato limitante. Genera ansia nella scelta del progetto e mal si adatta alle dinamiche di sviluppo reali, dove ritengo sia fisiologico scambiarsi task o riassegnare il lavoro in base al tempo rimasto e alle attitudini emerse in corso d'opera.
- **Stesura della Relazione (critico):** Il documento richiesto risulta a mio avviso eccessivamente rigido e complesso da redigere. Nonostante il template fornito, non è stato immediato decifrare cosa fosse effettivamente richiesto, complice una spiegazione in aula forse troppo rapida rispetto alla complessità del documento stesso. Una semplificazione o uno snellimento delle richieste lato relazione/documentazione aiuterebbe gli studenti a focalizzare maggiormente le energie sulla scrittura e progettazione del codice.

In sintesi, pur riconoscendo il valore didattico dell'esperienza che è stata per me, spero che questi spunti possano essere utili per calibrare ulteriormente questo esame del corso di OOP.

Guida utente

All'avvio del software viene mostrata immediatamente la schermata di gioco nella quale è possibile individuare:

- Un bordo giallo intorno all'utente che deve giocare il turno attuale
- Una carta coperta al centro del tavolo che indica il mazzo pescate interagibile
- Una carta scoperta al centro del tavolo che indica la carta attualmente in gioco
- Le proprie carte mostrate nella parte bassa della schermata, queste sono quelle che saranno giocabili nel proprio turno

In caso di impossibilità di difendersi dai malus avversari sarà necessario cliccare sul mazzo pescate (che sarà contornato da un bordo rosso) per accettare il malus.
Nella parte superiore dell'area di gioco centrale è ben visibile l'indicatore dell'**Evento in corso** (es. "Standard Game", "Total Chaos", ecc...). Questa etichetta testuale permette di identificare a colpo d'occhio quale variante delle regole è attiva per la partita corrente, influenzando le strategie di gioco.

Appena sopra le proprie carte è presente una piccola descrizione delle azioni dei giocatori o avvisi sullo stato di malus attivo.

Al termine della partita sarà mostrata una piccola schermata dove verrà indicato il vincitore e sarà possibile scegliere se iniziare una nuova partita o terminare il gioco.

Esercitazioni di laboratorio

B.0.1 luca.gianelli@studio.unibo.it

Laboratorio 07 : virtuale.unibo.it

Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=207921#p286077

Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=209589#p288463

laboratorio 12: https://virtuale.unibo.it/mod/forum/discuss.php?d=211539#p290665

B.0.2 davide.zito2@studio.unibo.it

Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=207193#p285019

Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=207921#p286103

Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=208718#p287238

Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=209589#p288529

Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=210617#p289672

Laboratorio 12: https://virtuale.unibo.it/mod/forum/discuss.php?d=211539#p290741

Bibliografia