# 4 – Elaborazione – Iterazione 3

## 4.1 Introduzione

La terza iterazione si è concentrata sul completamento e l'affinamento di DigiGoose, portando l'applicazione da console a un'esperienza utente migliore. Gli obiettivi principali di questa fase sono stati l'introduzione della GUI, l'implementazione del sistema di persistenza per il salvataggio e caricamento delle partite, e l'irrobustimento della logica di gioco attraverso la risoluzione di bug e il perfezionamento delle meccaniche esistenti.

## 4.2 Implementazione dell’interfaccia utente con Swing

Per quanto riguarda la realizzazione dell’interfaccia grafica, è stata utilizzata la libreria Swing di Java. Questa evoluzione mira a migliorare significativamente l'esperienza dell'utente, rendendo il gioco più accessibile e intuitivo. Nei paragrafi seguenti, verranno analizzati i componenti chiave della nuova interfaccia grafica: la schermata di configurazione della partita, la rappresentazione interattiva del tabellone di gioco e le modalità con cui vengono comunicati visivamente gli effetti delle caselle speciali.

### 4.2.1 Configurazione della partita

La configurazione di una nuova partita è il primo passo interattivo che l'utente compie dopo aver selezionato "Nuova Partita" dal menu principale. Questa fase è gestita da un pannello dedicato, settingsPanel all'interno della classe InterfacciaUtente, che guida l'utente nella definizione dei parametri fondamentali del gioco.

#### 4.2.1.1 Selezione del numero di giocatori e configurazione individuale

Al centro del pannello di configurazione si trova un JSpinner che permette all'utente di selezionare il numero di giocatori desiderato. La modifica del valore in questo spinner scatena un evento che invoca il metodo updatePlayerConfigPanelsContainer. Quest'ultimo è responsabile della creazione o rimozione dinamica di istanze della inner class PlayerConfigPanel. Per ogni giocatore selezionato, viene visualizzato un PlayerConfigPanel dedicato, organizzato verticalmente all'interno di un JPanel con BoxLayout, a sua volta contenuto in un JScrollPane per gestire un numero variabile di giocatori senza compromettere il layout.

Ogni PlayerConfigPanel incapsula gli elementi UI necessari per configurare un singolo giocatore:

1. **Tipo Giocatore**: Una JComboBox permette di scegliere se il giocatore sarà UMANO o COMPUTER. La selezione di COMPUTER disabilita il campo nome e lo precompila con un identificativo standard (es. "CPU 1").
2. **Nome Giocatore**: Un JTextField per inserire il nome del giocatore. Questo campo è attivo solo se il tipo selezionato è UMANO.
3. **Colore Pedina**: Una JComboBox elenca tutti i colori disponibili definiti nell'enum Colore. Viene preselezionato un colore diverso per ogni pannello per facilitare la scelta di colori unici, sebbene la validazione finale avvenga al momento della conferma.

#### 4.2.1.2 Raccolta, validazione e conferma delle impostazioni

Una volta che l'utente ha definito le impostazioni per tutti i giocatori, preme il pulsante "Conferma Impostazioni". Questa azione scatena il metodo raccogliEConfermaImpostazioni in InterfacciaUtente.

Vengono eseguite delle validazioni cruciali:

* Il nome per i giocatori umani non può essere vuoto.
* Ogni giocatore deve aver selezionato un colore.
* Tutti i colori selezionati devono essere unici tra i giocatori.

Se una di queste validazioni fallisce, viene mostrato un messaggio d'errore specifico, e il processo di conferma si interrompe, permettendo all'utente di correggere gli input.

Se tutte le validazioni hanno successo, InterfacciaUtente invoca i metodi appropriati del GiocoController. Infine, viene chiamato giocoController.confermaImpostazioni(), che a sua volta invoca inizializzaNuovaPartita() per creare l'oggetto Partita con i dati raccolti.

Il design adottato permette una chiara separazione delle responsabilità: la InterfacciaUtente si occupa della presentazione e della raccolta dell'input, il GiocoController orchestra il processo di configurazione e valida i dati a un livello più alto, e ImpostazioniPartita agisce come un data-holder temporaneo prima della creazione effettiva dell'oggetto Partita.

### 4.2.2 Tabellone di gioco

Una volta completata la configurazione e avviata la partita, l'interfaccia utente presenta il tabellone di gioco. La sua visualizzazione e interazione sono gestite dalla classe GameBoardPanel, un JPanel personalizzato progettato specificamente per disegnare dinamicamente lo stato del gioco.

#### 4.2.2.1 Struttura e layout del tabellone

Il GameBoardPanel è responsabile della rappresentazione grafica delle 64 caselle del gioco. Al momento della sua inizializzazione, o quando le dimensioni del pannello cambiano, il metodo calculateSquarePositions viene invocato. Questo metodo calcola le coordinate per ogni casella, disponendole in una griglia. La dimensione di ogni casella e il numero di caselle per riga sono calcolati dinamicamente per adattarsi allo spazio disponibile nel pannello contenitore, garantendo una visualizzazione ottimale su diverse risoluzioni, pur mantenendo una dimensione minima preferita per la leggibilità.

#### 4.2.2.2 Rendering delle caselle e delle informazioni

Il cuore del rendering avviene nel metodo paintComponent(Graphics g), che viene sovrascritto per disegnare tutti gli elementi del tabellone. Ogni casella è un rettangolo, dove il colore di sfondo varia per distinguere le caselle normali (bianco), la casella di partenza (grigio chiaro), la casella di arrivo (arancione) e le caselle speciali (giallo chiaro).

#### 4.2.2.3 Visualizzazione delle pedine dei giocatori

Le pedine dei giocatori sono rappresentate da ovali colorati disegnati sopra le caselle corrispondenti alla loro posizione attuale. Il GameBoardPanel itera sulla lista dei giocatori della partita corrente:

1. Per ogni giocatore, recupera la sua posizione e il colore della sua pedina. I colori definiti nell'enum it.digigoose.model.Colore sono mappati a istanze di java.awt.Color attraverso una HashMap per il rendering.
2. Per gestire la sovrapposizione di più pedine sulla stessa casella, viene calcolato un offset. Le pedine vengono disposte in una piccola griglia all'interno della casella (fino a 3 pedine per riga virtuale all'interno della casella), evitando che si coprano completamente. Una HashMap tiene traccia di quante pedine sono già state disegnate su una specifica casella per calcolare l'offset corretto.
3. Su ogni pedina viene disegnata l'iniziale del nome del giocatore, con un colore di testo (bianco o nero) scelto per garantire il contrasto con il colore della pedina.

#### 4.2.2.4 Aggiornamento del board durante la partita

Ogni volta che lo stato della partita cambia (es. un giocatore si muove), il metodo repaint() viene chiamato sul GameBoardPanel. Questo forza la riesecuzione di paintComponent, assicurando che il tabellone rifletta sempre lo stato più recente del gioco, incluse le posizioni delle pedine e le informazioni visualizzate.

### 4.2.3 Visualizzazione effetti caselle speciali

## Per aiutare i giocatori a capire cosa succede quando si finisce su una casella speciale, il gioco offre informazioni chiare in due modi: attraverso piccoli messaggi che appaiono quando si passa il mouse sopra una casella e tramite aggiornamenti testuali durante la partita.Quando il giocatore muove il mouse sul tabellone, il sistema è attento a dove si trova il cursore. Se il mouse si ferma per un secondo su una casella speciale (o su quella di partenza o di arrivo), appare una piccola finestra con una spiegazione. Questo piccolo ritardo evita che i messaggi appaiano e scompaiano troppo velocemente se si muove il mouse rapidamente. Se il mouse si sposta via dalla casella o esce dal tabellone, la finestra informativa scompare.

Dentro il tooltip è presente come titolo il nome della casella speciale preso dall’enum e una piccola descrizione dell’effetto.

In basso della finestra, rimane l’output testuale della console, potendo quindi seguire ogni passaggio del flusso di gioco.

## 4.3 Sistema di salvataggio e caricamento partita con JSON

Per migliorare l'esperienza di gioco e permettere agli utenti di interrompere e riprendere le proprie sessioni, è stato implementato un sistema di persistenza dei dati. Questa funzionalità consente di salvare lo stato corrente di una partita in un file e di ricaricarlo successivamente. È stato scelto il formato JSON per la sua leggibilità e la facile interoperabilità, avvalendosi della libreria org.json. Il sistema gestisce un singolo slot di salvataggio, sovrascrivendo automaticamente il file esistente a ogni nuovo salvataggio.

### 4.3.1 Creazione del file JSON per lo stato del gioco

Il processo di salvataggio è orchestrato dalla classe GestoreSalvataggio, specificamente attraverso il metodo salvaPartita(Partita partita). Quando l'utente sceglie di salvare la partita (tipicamente prima di tornare al menu principale da una partita in corso), questo metodo viene invocato. Le informazioni cruciali della partita vengono convertite in una struttura JSON.

Viene creato un oggetto JSON principale che rappresenta la partita. Al suo interno, vengono memorizzati attributi fondamentali come l'ID univoco della partita, la data di creazione, il turno e il giro correnti, e lo stato attuale del gioco (es. IN\_CORSO).

Una parte significativa dell'oggetto JSON è dedicata alla lista dei giocatori. Per ogni giocatore, viene creato un sotto-oggetto JSON che include il suo ID, nome, tipo (umano o CPU), il numero di turni che deve eventualmente saltare e lo stato della sua richiesta di rilancio. Anche i dettagli della pedina associata a ciascun giocatore, come il colore e la posizione corrente sul tabellone, sono serializzati all'interno dell'oggetto del rispettivo giocatore.

L'ordine di gioco dei giocatori, che potrebbe essere diverso dall'ordine di creazione a causa del mescolamento iniziale, viene preservato memorizzando una lista degli ID dei giocatori nell'ordine corretto. Viene inoltre salvato l'ID del giocatore il cui turno è attualmente in corso.

Una volta che l'intero stato della partita è stato tradotto in questa struttura JSON gerarchica, l'oggetto JSON viene scritto su un file denominato digigoose\_save.json all'interno di una sottodirectory save/. Se un file di salvataggio esiste già, viene sovrascritto.

### 4.3.2 Gestione del caricamento della partita

La funzionalità di caricamento permette di ripristinare una partita precedentemente salvata. All'avvio dell'applicazione, InterfacciaUtente verifica, tramite gestoreSalvataggio.esisteSalvataggio(), la presenza del file digigoose\_save.json. Se il file esiste, il pulsante "Carica Partita" nel menu principale viene reso visibile e attivo.

Selezionando "Carica Partita", viene invocato il metodo caricaPartita() di GestoreSalvataggio. Questo metodo legge il contenuto del file JSON. Il testo JSON viene parsato e riconvertito in un oggetto.

Da questo oggetto principale, vengono estratte le varie informazioni per ricostruire l'oggetto Partita e i suoi componenti. Si ricreano le istanze dei Giocatore con i loro attributi (nome, tipo, colore pedina, posizione, turni saltati, richiesta rilancio), assicurandosi di ripristinare anche i loro ID originali. La ricostruzione dell'ordine dei giocatori e l'identificazione del giocatore corrente avvengono utilizzando gli ID salvati, mappandoli ai giocatori appena ricreati.

Una volta che l'oggetto Partita è stato completamente ripristinato in memoria con i dati del file di salvataggio, viene passato al GiocoController e al PartitaController. L'interfaccia utente passa quindi direttamente alla schermata di gioco (gamePlayPanel), inizializzando la logica di gioco (iniziaLogicaDiGioco()) con lo stato caricato. Viene visualizzato un messaggio di conferma del caricamento.

Se durante il processo di caricamento si verificano errori, ad esempio a causa di un file corrotto o mancante (nonostante il controllo iniziale), vengono gestite le eccezioni (IOException) e l'utente viene informato tramite un messaggio di errore, tipicamente rimanendo nel menu principale.

## 4.4 Irrobustimento della logica di gioco e correzioni bug

Oltre all'introduzione di nuove funzionalità come l'interfaccia grafica e il sistema di salvataggio, una parte significativa del lavoro in questa iterazione finale è stata dedicata al miglioramento della stabilità e della correttezza della logica di gioco. Questo ha comportato l'analisi e la risoluzione di alcuni bug emersi durante i test e il raffinamento di meccaniche di gioco complesse per garantire un'esperienza fluida e aderente alle regole.

### 4.4.1 Gestione bug rilancio dei dadi

Una delle meccaniche fondamentali del gioco dell'oca è la possibilità, offerta da alcune caselle speciali, di far rilanciare i dadi al giocatore. Durante le fasi di test precedenti, era emersa una problematica legata alla gestione di questa situazione, in particolare quando il giocatore che otteneva il diritto al rilancio era un computer (CPU) o quando il rilancio si verificava in contesti di effetti a catena.

La problematica principale risiedeva nel flusso di controllo del turno. In alcuni scenari, dopo che un giocatore (specialmente una CPU) aveva ottenuto un rilancio, il sistema poteva passare erroneamente il turno al giocatore successivo prima che il rilancio venisse effettivamente eseguito, oppure il pulsante per il rilancio per un giocatore umano non si riattivava correttamente.

La correzione ha implicato una revisione del metodo concludiTurnoEAvanza in InterfacciaUtente. Ora, prima di passare il turno, si verifica esplicitamente se il giocatoreCorrente ha l'attributo richiedeRelancio impostato a true.

* Se un giocatore umano ha diritto al rilancio, il pulsante rollDiceButton viene riabilitato, e il turno non passa. L'utente vedrà un messaggio che lo invita a lanciare di nuovo.
* Se un giocatore CPU ha diritto al rilancio, viene avviato un nuovo Timer (computerReRollTimer) che, dopo un breve intervallo per simulare la riflessione, invoca nuovamente gestisciTurnoGiocatore. Anche in questo caso, il turno non passa al giocatore successivo finché il rilancio non è stato completato.  
  L'attributo richiedeRelancio del giocatore viene resettato a false solo dopo che il rilancio è stato effettivamente gestito, assicurando che l'azione venga eseguita una sola volta come previsto.

Inoltre, è stata migliorata la gestione della logica di gioco quando un effetto speciale come RILANCIA\_TORNA\_INDIETRO viene attivato. In questo caso, l'interfaccia utente ora gestisce un secondo lancio di dadi dedicato (invocando partitaController.tiraDadi()) il cui risultato determina di quanti passi il giocatore deve tornare indietro. Questo isola il "rilancio per tornare indietro" dal normale flusso di un "rilancio per avanzare", prevenendo confusioni e garantendo che l'effetto venga applicato correttamente e in modo trasparente per l'utente, con messaggi appropriati nell'area di testo.

### 4.4.2 Gestione turni giocatori bloccati

Diverse caselle speciali nel Gioco dell'Oca impongono restrizioni ai giocatori, bloccandoli per un certo numero di turni o fino al verificarsi di una specifica condizione (ad esempio, ottenere un particolare risultato con i dadi). La gestione di questi stati di "blocco" è stata significativamente irrobustita per coprire tutti gli scenari e garantire che i giocatori vengano sbloccati correttamente.

La logica di gestione dei giocatori bloccati è centralizzata principalmente nel metodo prepareNextTurn di InterfacciaUtente e nel metodo verificaELiberaGiocatoriBloccati di PartitaController.  
Quando è il turno di un giocatore (prepareNextTurn):

1. Si controlla l'attributo turniSaltati del giocatore corrente.
   * Se turniSaltati > 0: il giocatore deve saltare un numero fisso di turni. Viene visualizzato un messaggio, turniSaltati viene decrementato, e il turno passa immediatamente al giocatore successivo (tramite una chiamata ricorsiva a prepareNextTurn dopo un breve Timer per dare tempo all'utente di leggere il messaggio).
   * Se turniSaltati < 0: il giocatore è bloccato da una condizione speciale (es. in attesa di un 6 per uscire dalla Prigione, o di un 3/6 sulla casella 26, o di un 4/5 sulla casella 53). Viene visualizzato un messaggio specifico che descrive la condizione di blocco. Il giocatore tenterà comunque di lanciare i dadi.
   * Se turniSaltati == 0: il giocatore è libero di agire.

Il metodo gestisciTurnoGiocatore in InterfacciaUtente coordina il lancio dei dadi. Subito dopo il lancio, viene invocato partitaController.verificaELiberaGiocatoriBloccati(currentDiceRoll). Questo metodo analizza il risultato dei dadi appena lanciati:

* Controlla se nei dadi è presente un 6. Se sì, tutti i giocatori che erano bloccati con turniSaltati == -1 (effetto "ATTENDI\_DADO" generico, che prima non era chiaramente implementato e ora è implicitamente coperto da caselle specifiche come la Prigione se si volesse generalizzare, anche se nel codice attuale -1 è usato per la casella 19 originale, ora FERMA\_UN\_TURNO) vengono sbloccati (il loro turniSaltati è posto a 0).
* Se il giocatore *corrente* era bloccato in Prigione (turniSaltati == -2, ora associato a FERMA\_DUE\_TURNI o specifici turniSaltati positivi dalla casella Prigione 52) e ha lanciato un 6, viene sbloccato. *Nota: La casella 52 (Prigione) ora imposta turniSaltati = 2, quindi lo sblocco per un 6 non è più la meccanica primaria per quella casella, ma il codice per -2 e un 6 rimane in verificaELiberaGiocatoriBloccati.*
* Se il giocatore *corrente* era sulla casella 26 (turniSaltati == -3) e ha lanciato un 3 o un 6, viene sbloccato.
* Se il giocatore *corrente* era sulla casella 53 (turniSaltati == -4) e ha lanciato un 4 o un 5, viene sbloccato.

Se, dopo il lancio dei dadi e la verifica, il giocatore corrente risulta ancora bloccato (cioè turniSaltati < 0), il suo movimento viene impedito, un messaggio informa l'utente che non ha ottenuto il risultato necessario, e il turno passa al giocatore successivo. Se invece viene sbloccato o non era bloccato, procede con il movimento.

È stata inoltre chiarita la comunicazione all'utente dello stato di blocco. La funzione getBlockMessage in InterfacciaUtente fornisce stringhe di testo appropriate per ogni tipo di blocco condizionale, che vengono visualizzate nell'area messaggi.

Questo approccio più strutturato assicura che le condizioni di blocco e sblocco siano gestite correttamente, che il flusso del gioco non si interrompa inaspettatamente e che l'utente riceva un feedback chiaro sullo stato dei giocatori bloccati e sui requisiti per il loro sblocco.