Presentazione UML

Domenico Parlato, Andrea Santaniello, Juan Marco Patagoc, Eliseo Paoletti

GC53

Il diagramma UML del gioco adotta il seguente pattern architetturale Model-View-Controller (MVC):

* **Modello**: Il Modello rappresenta i dati e le regole del gioco. Nel nostro caso, le classi **Table**, **Player** e **PlayerArea** costituiscono il Modello. Queste classi contengono i metodi e gli attributi che definiscono lo stato del gioco e le azioni che i giocatori possono intraprendere.
* **Vista**: Nel nostro diagramma UML, la Vista non è esplicitamente rappresentata. La Vista è responsabile della presentazione dei dati del Modello all’utente.
* **Controller**: Il Controller gestisce l’interazione tra il Modello e la Vista. Nel nostro caso, la classe **GameController** funge da Controller. Questa classe chiama i metodi di modifica sul Modello, gestisce il flusso di gioco e controlla le azioni dei giocatori.

In questo modo, il Controller **GameController** può modificare lo stato del Modello (le classi **Table**, **Player**, **PlayerArea** e le relative classi a loro associate) in risposta alle azioni dell’utente sulla Vista.

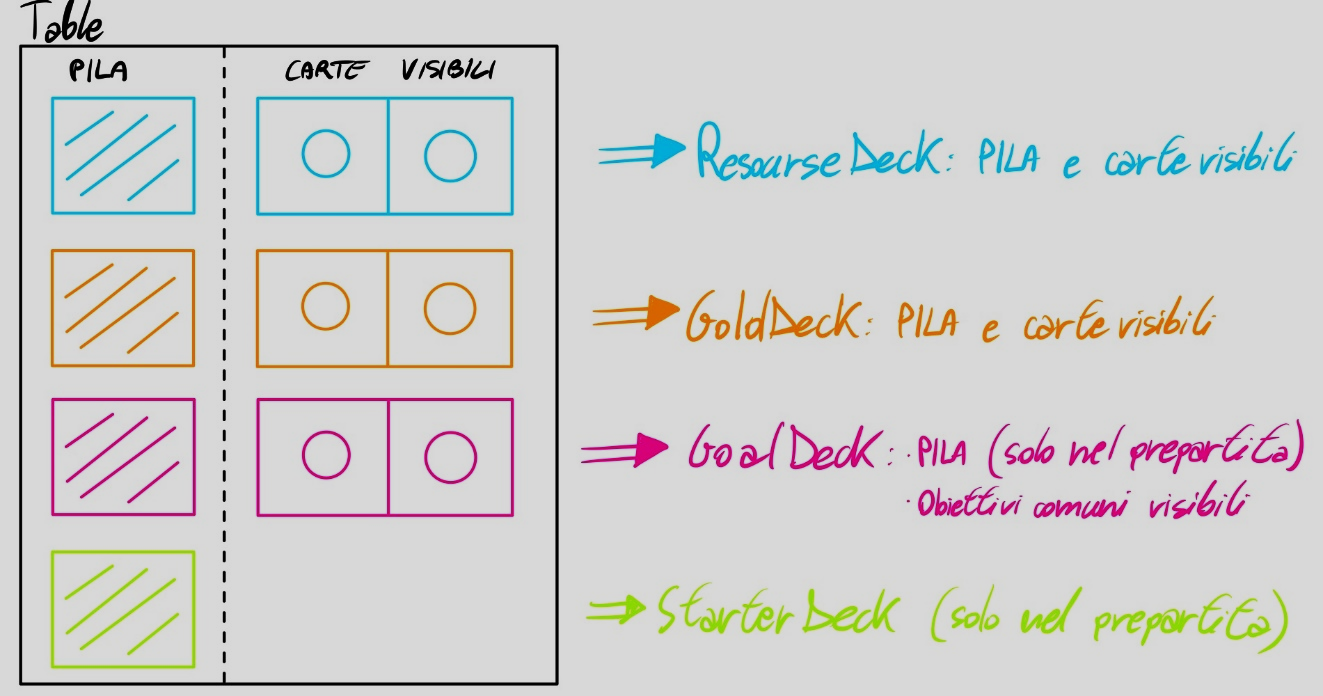
Per una visione più chiara cercherò adesso di descrive il flusso del gioco, escludendo la parte di inizializzazione della partita. Il flusso del gioco è gestito principalmente dalla classe **GameController**, che determina l’ordine dei turni dei giocatori (da 2 a 4) tramite il metodo **getNextPlayer**. Ogni giocatore è rappresentato dalla classe **Player**, che ha vari attributi:

* **Token**: Il colore della pedina del giocatore.
* **Username**: Il nome utente del giocatore.
* **Points**: Il contatore dei punti del giocatore.
* **PersonalGoalCard**: La carta obiettivo personale del giocatore.
* **personalHand**: La mano del giocatore, rappresentata da un array di dimensione 3 di tipo **Res&Gold**.

Dalla **VIEW** il giocatore seleziona una carta da **personalHand** che desidera giocare tramite il metodo **getCard**. Il giocatore ora selezione il modo in cui vuole posizionare la carta, se sul fronte o sul retro, e il controller chiama il metodo **turnCard** che prende in ingresso la suddetta carta dalla mano del giocatore.

Fatto ciò al giocatore viene richiesto di selezionare la posizione dove intende giocare la carta sulla plancia di gioco (nel nostro UML si tratta della classe **PlayerArea**), rappresentata da noi come una matrice quadrata di dimensione 59x59 (dallo studio del caso pessimo di posizionamento carte tutte in diagonale in una partita a 2 giocatori): la selezione avviene da un array di stringe in cui collezioniamo le coordinate della matrice di posizioni disponibili e adiacenti a carte già presenti, quindi posizioni in cui il giocatore può effettivamente piazzare la carta, senza che ci sia la possibilità di errori di piazzamento dovute a posizioni non adiacenti a carte già piazzate. Questo array di stringhe (in cui usiamo la formattazione delle coordinate su stringa in questo modo: "***coordinataX.coordinataY***") è un attributo della classe PlayerArea chiamato **adjacentPosition** (un arrayList che viene aggiornato ad ogni nuovo piazzamento di carta). Consequenzialmente al piazzamento della carta la classe **PlayerArea** aggiorna anche i contatori per le risorse e oggetti visibili sulla plancia di gioco e nel caso aggiunge punti all'attributo **Points** di **Player**.

Giocata la carta il giocatore ne deve pescare una nuova dal mazzo risorse (coperto o scoperto) o dal mazzo oro (coperto o scoperto) presente nella classe **Table**: in ordine, **inTableR** e **ShowR** per le carte risorsa. **inTableG** e **ShowG** per le carte oro. Il controller fa partire il metodo **drawCard** che chiama il metodo in table **getCard** con la posizione indicata dal giocatore e ristretta nell'enumerazione **DrawType** che racchiude i mazzi coperti e scoperti dei due tipi oro e risorse. A questo punto **Table** va aggiornato negli attributi carte scoperte in base alla pesca del Player che ha scelto le carte visibili: vengono invocati i metodi **updateGDeck** e **updateRDeck**, che fanno un pop della carta in cima alla pila e la posizionano nella casella delle carte visibili del tipo mancante. Così si conclude un turno e controller si assicura di chiamare il prossimo giocatore con **getNextPlayer**. Controller inoltre fa partire l'**endGame** (ultimo turno senza il pescaggio di carte, ma solo con il piazzamento di una carta) nel caso in cui uno tra i due metodi **check20Points** o **checkEmptyDeck** ritornino una ***TRUE*** (un giocatore ha raggiunto i 20 punti o le pile in table dei mazzi Oro e Risorse sono finite).



**Analisi della Matrice Dashboard**

La matrice è la struttura dati chiave utilizzata per gestire il posizionamento delle carte nel gioco. Ecco come funziona:

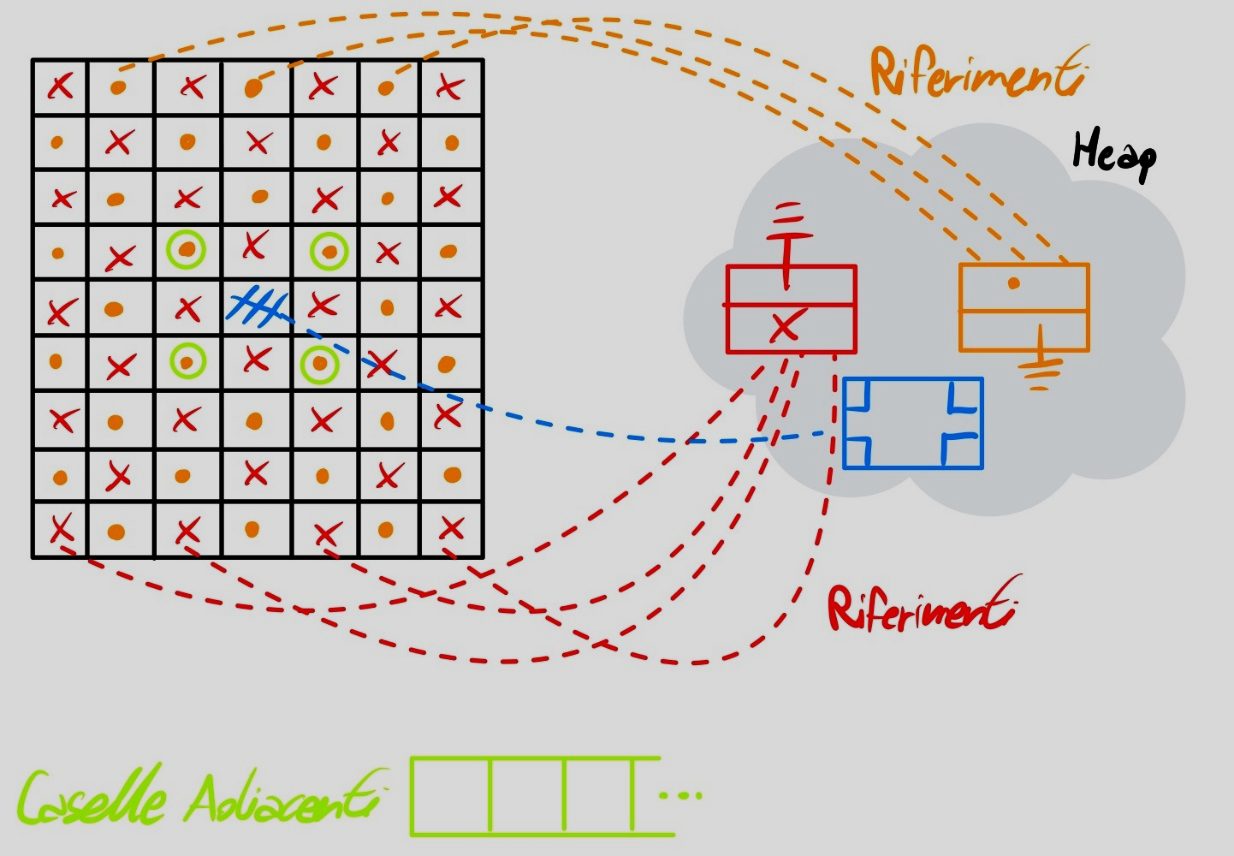
* **Dashboard della Matrice**: La matrice è rappresentata come **dashboard: Cell** **[][]**, dove ogni cella può contenere una carta giocabile (**PlayableCard**).
* **Posizionamento delle Carte**: Le carte possono essere inserite nella matrice tramite il metodo **insertCard**, che prende una carta (**inCard: Res&Gold**) e le coordinate di caselle adiacenti e disponibili salvate come stringa nell’array **adjacentPosition** (**index: string**) per posizionare la carta correttamente.
  + Per le **GoldCard** è prevista una verifica delle condizioni in piazzamento se in Front: il metodo **verifyGold (in inCard: GoldCard)** che visualizza le condizioni sulla carta Gold indicate dall’attributo **requirements: RESOURCE** e le confronta con i contatori della classe **PlayerArea** per verificare la disponibilità di risorse necessarie alla carta
* **Aggiornamento delle Posizioni Adiacenti**: Dopo l’inserimento di una carta, il metodo **updateAdjacentPositions** aggiorna l’array di posizioni adiacenti e aventi l’attributo **available: boolean *TRUE*** per riflettere il nuovo stato della matrice e avere un dominio di caselle utilizzabili dal giocatore al prossimo turno.
* **Verifica delle Condizioni**: nell’**endGame** vengono verificati gli obiettivi in comune e privato di ogni giocatore tramite il metodo **verifyGoal** controlla se una carta obiettivo (**GoalCard**) soddisfa le condizioni di vittoria basate sulla disposizione attuale delle carte nella matrice.

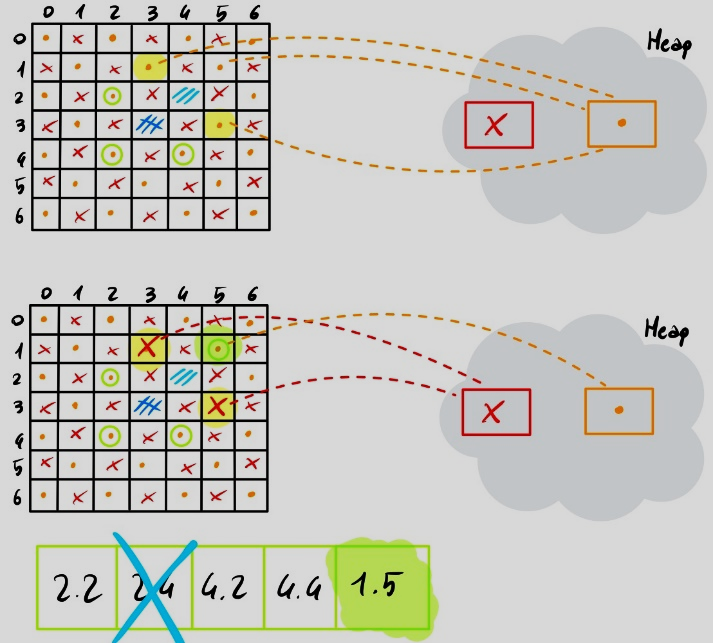
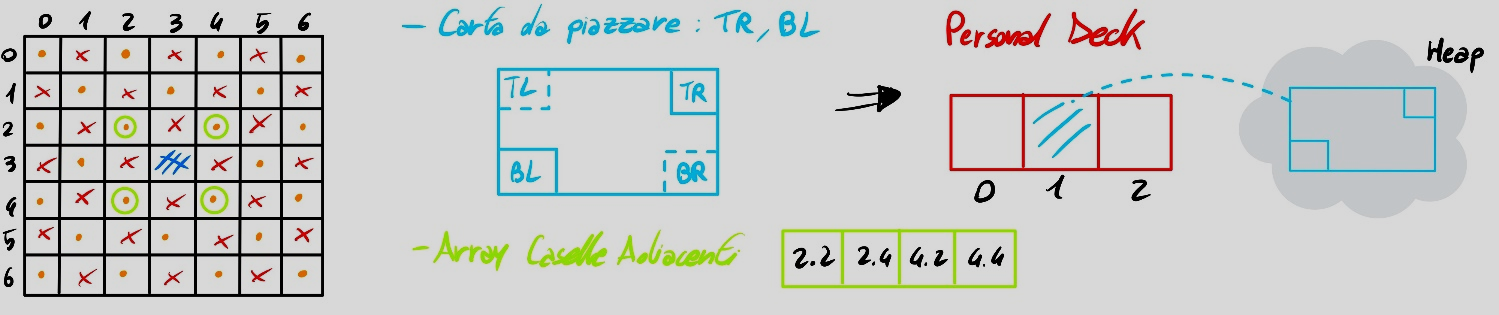
**Inizializzazione della Matrice**

La matrice del gioco è inizializzata come una scacchiera, utilizzando una matrice di tipo Cell. Ogni Cell ha due attributi:

* **content: PlayableCard:** Questo attributo rappresenta la carta che è attualmente posizionata in quella cella.
* **available: boolean**: Questo attributo indica se la cella è disponibile per il posizionamento di una nuova carta.

Le carte sono collegate tra loro attraverso gli angoli, quindi le celle immediatamente adiacenti (sia in verticale che in orizzontale) a una carta appena piazzata non sono utilizzabili. Inoltre, l’attributo **available** viene utilizzato per indicare la non disponibilità di una casella successiva all’angolo invisibile di una carta appena piazzata, aiutandoci a marchiare una casella che non andrà inserita nell’arrayList **adjacentPosition** e impedendo così al giocatore di piazzare successivamente una carta nella casella in questione.



****

**Classe Corner**

La classe **Corner** è utilizzata per rappresentare gli angoli delle carte. Questa classe ha vari attributi che permettono di gestire le interazioni tra le carte e le loro posizioni sulla scacchiera.

* **hidden: boolean**: Questo attributo rappresenta l’esistenza di un angolo utile per un collegamento con un’altra carta. È un attributo **final**, quindi non può essere modificato dopo l’inizializzazione.
* **position: POSITION**: Questo attributo indica la posizione grafica dell’angolo sulla carta. È determinato dall’enumerazione **Position** e non può essere modificato.
* **content: CONTENT**: Questo attributo indica il contenuto di un angolo quando **hidden** è ***TRUE***. È determinato dall’enumerazione **CONTENT** e non può essere modificato.
* **cornerAvailable: BOOLEAN**: Questo attributo indica la disponibilità di un angolo nella plancia di gioco. Se un angolo subisce un collegamento con una carta in procinto di essere piazzata, il suo angolo e il contenuto verranno “nascosti” dalla suddetta carta e non saranno più utilizzabili. Il metodo **setCornerAvailable** serve a modificare il valore di questo attributo.

In questo modo, la classe **Corner** permette di gestire le interazioni tra le carte e le loro posizioni sulla scacchiera.

