Il progetto TreSette

Andrea Di Masi, matricola 2193432

Corso di Metodologie di programmazione - Teledidattica

1 Introduzione

Il Tresette è un gioco di carte tradizionale, comune in molte regioni italiane, le cui origini si contendono tra Spagna e Napoli. Si gioca tipicamente in 2, 3 o 4 giocatori. Il mazzo utilizzato è quello da 40 carte (solitamente napoletane, piacentine o altre varianti regionali), suddivise in quattro semi: denari (o ori), coppe, bastoni e spade. Le carte hanno una specifica gerarchia di forza che segue il seguente ordine, dal più forte al più debole: 3, 2, Asso, Re, Cavallo, Fante, 7, 6, 5, 4. Il sistema di punteggio del Tresette segue la regola seguente: un Asso vale un punto intero, mentre le figure (Re, Cavallo, Fante), il 2 e il 3 valgono un terzo di punto ciascuna. Le altre carte (dal 7 al 4) non hanno valore ai fini del punteggio. Questa distinzione tra il valore di presa e il valore di punteggio porta a scelte strategiche interessanti: a volte conviene vincere una presa per aggiudicarsi un Asso, mentre altre volte è più saggio conservare una carta di valore per controllare prese future. Alla fine della mano, si applica una regola di arrotondamento per difetto, per cui si considerano solo i punti interi, mentre si scartano i terzi di punto. Un'altra caratteristica distintiva del gioco è l'obbligo di rispondere al seme, chiamato "palo". Quando un giocatore apre un turno, gli altri sono obbligati a giocare una carta dello stesso seme, se ne possiedono. Se un giocatore non ha carte del seme di mano, può giocare una carta di qualsiasi altro seme (in questo caso si dice che "gioca piombo" o che "taglia"), ma non potrà vincere la presa. Questa regola è la componente strategica principale, poiché i giocatori devono gestire le carte in loro possesso per ottimizzare le vincite e minimizzare le perdite.

Svolgimento del Gioco (partita a 4 giocatori)
La modalità più comune è quella a 4 giocatori, che si sfidano in due coppie contrapposte (i compagni di squadra siedono uno di fronte all'altro). Il mazziere, dopo aver mescolato le carte, le distribuisce in senso antiorario, dieci per ogni giocatore. Il primo giocatore di mano gioca una carta, determinando il "palo" del turno. Gli altri giocatori, a turno, rispondono rispettando l'obbligo di

seme. La presa viene vinta dal giocatore che ha giocato la carta di valore più alto del seme di mano. Chi vince la presa raccoglie le quattro carte giocate e le pone coperte davanti a sé. Sarà poi lui a iniziare il turno successivo. Il gioco prosegue per 10 turni, fino all'esaurimento di tutte le carte. Una peculiarità del Tresette giocato in coppia è la possibilità di comunicare con il proprio compagno tramite tre segnali convenzionali, da effettuare solo quando si è di mano all'inizio del turno:

- Busso: si bussa con le nocche sul tavolo. Comunica al compagno di giocare la carta più alta che possiede di quel seme, con l'intenzione di proseguire a giocare lo stesso palo nei turni successivi.
- Volo: si lancia la carta sul tavolo facendola "volare". Indica che quella è l'ultima carta di quel seme che si possiede.
- Liscio: si fa scivolare la carta sul tavolo. comunica che si possiedono altre carte di quel seme, oltre a quella giocata.

Alla fine della "smazzata" (le 10 prese), ogni coppia somma i punti delle carte che ha raccolto. Oltre al valore delle singole carte, si applicano le seguenti regole:

- Punto dell'ultima presa: la coppia che si aggiudica l'ultima presa della mano ottiene un punto aggiuntivo.
- Cappotto: se una coppia non riesce a effettuare nemmeno una presa, si dice che subisce "cappotto". La coppia avversaria, in questo caso, vince immediatamente la partita o ottiene 17 punti, invece di 11.

  L'obiettivo del gioco è raggiungere un punteggio prestabilito dai giocatori, che solitamente è di 21, 31 o 41 punti.

Implementazione del gioco in java
Nel progetto presente sono state implementate le regole e la variante fino ad'ora descritte, mentre altre varinti e opzioni sono state escluse. Può essere utile menzionare ciò che è stato escluso: la regola opzionale dell'Accusa; la varinte a due giocatori ("a spizzico"); la variante a tre giocatori. Queste componeti potranno essere aggiunte in futuro grazie alla modularità del design attuale.

2 Scelte di design

2.1 Architettura generale e pattern di alto livello

L'architettura segue il paradigma Model–View–Controller (MVC) con responsabilità nettamente separate: il package *model* contiene le dinamiche del gioco (giocatori, mano, punteggi, regole e ciclo di gioco) e coordina il flusso tramite *GameManager* e *deal.Deal*; il package *view* contiene i componenti

Swing responsabili del rendering e dell'interazione dell'utente (ad esempio *GameFrame*); il package *controller* fornisce l'interfaccia di controllo tra UI e modello, in particolare *GameController*. Quest'ultimo riceve *GameManager* nel costruttore, espone i comandi necessari all'interfaccia (ad esempio startGame, playHumanCard, pauseGame, resumeGame) e si occupa di tradurre le notifiche del modello in eventi destinati alla vista. La comunicazione è quindi bidirezionale ma mediata esclusivamente dal controller: la View non chiama mai il Model direttamente, ma invia le richieste d'uso al *GameController*, che delega al *GameManager* o *deal.Deal*. Allo stesso tempo, il Model non conosce la View e notifica il proprio stato tramite *ModelEvents*, che il Controller adatta in *ViewEvent* consumabili dalla UI. Per minimizzare l'accoppiamento, gli eventi del modello portano con se un *DealSnapshot* immutabile che rappresenta l'istantanea della mano (indice della mano, giocatore corrente, carte sul tavolo, dimensioni delle mani, carte vinte, eventuale vincitore dell'ultima presa, possibilità di segno e stato di pausa, oltre alla mano dell'umano). La View utilizza esclusivamente queste informazioni per aggiornare l'interfaccia, senza dover interrogare il Model, mentre il Controller mantiene stabile il contratto verso la UI traducendo i *ModelEvents* nei corrispondenti *ViewEvent*. Nel package *profile* la separazione delle responsabilità è ulteriormente enfatizzata dalla coppia *ProfileRepository* e *ProfileService*, che distingue l'accesso ai dati dalla logica di servizio, con *UserProfile* a modellare lo stato utente. Un esempio concreto dell'inizio del flusso MVC a seguito di un'azione dell'utente è mostrato nei diagrammi UML, nel diagramma nominato "Sequenza: giocata carta -> propagazione a UI", dove si vede la trasformazione della scelta dell'utente in un comando verso il Model e la successiva propagazione degli aggiornamenti verso la View tramite

2.2 Pattern Observer

Il progetto usa direttamente le API di *java.util.Observable* e *java.util.Observer* per realizzare il flusso tra model, controller e view. Gli eventi di dominio sono definiti in *model.events.ModelEvents* come record immutabili (ad esempio *CardPlayed*, *DealEnded*, *ScoresUpdated*). L'origine degli eventi è la classe *Deal*: quando nella mano del giocatore avviene qualcosa di rilevante la *Deal* costruisce un'istanza appropriata di *ModelEvents* e la trasmette ai suoi osservatori chiamando *notifyObservers(...)* sull'istanza *Deal* (Observable). Il flusso segue questa struttura:

1) *Deal* genera un evento *ModelEvents.** (ad esempio *CardPlayed*, *DealEnded*) e chiama *notifyObservers(event)*.

2) *GameManager* si registra come osservatore della *Deal* tramite *deal. addObserver(this)* (vedi *GameManager.attachToDeal(Deal)*). Quando la *Deal* notifica, *GameManager.update(Observable, Object)* viene invocato con un argument che contiene un *ModelEvents.Event*.

3) In *GameManager.update(...)* c'è una logica specifica: se l'*argument* è un'istanza di *DealEnded*, *GameManager* invoca *handleDealEnded(dealEnded.snapshot())*. In *handleDealEnded(...)* il *GameManager* aggiorna i punteggi tramite *ScoreManager*, costruisce e notifica *ModelEvents.ScoresUpdated* ed eventualmente *ModelEvents.GameEnded*.

4) Dopo questo controllo, *GameManager.update(...)* chiama *setChanged(); notifyObservers(argument);* e inoltra l'evento originale all'osservatore *GameController*. Questo perché quasi tutti gli eventi che *Deal* invia sono destinati al controller.

5) *GameController* si registra come osservatore di *GameManager* nel costruttore (*gameManager.addObserver(this)*). Quando riceve un *ModelEvents.** in *GameController.update(...)* riconosce il tipo concreto, costruisce l'equivalente *ViewEvent* e lo pubblica chiamando il suo *publish(ViewEvent)* che esegue *setChanged(), poi notifyObservers(event)*.

6) La view osserva il *GameController* e riceve i *ViewEvent*. Questi sono record immutabili che includono un *DealSnapshot*, una classe che fotografa lo stato della mano, e aggiornano la UI. Si noti che questi Snapshot sono neutri e forniscono variabili immutabili, in modo da evitare il rischio che la View operi cambiamenti sul model.

In pratica, *GameManager* agisce da mediatore: riceve eventi da *Deal*, applica logica solo per l'evento *DealEnded*, produce eventi derivati (*ScoresUpdated*, *GameEnded*) e rilancia l'evento ricevuto verso il controller. Gli altri eventi generati dalla *Deal* (ad esempio *CardPlayed*, *TrickStarted*, *TrickEnded*, *Sign*) vengono ricevuti in *GameManager.update(...)* e inoltrati senza ulteriori effetti applicativi: *GameManager* li rilancia con *notifyObservers(argument)*.

Esempio concreto: sequenza per la giocata di una carta *Deal* crea *ModelEvents.CardPlayed(playerId, cardCode, cardText, snapshot)* e chiama *notifyObservers(...)*. *GameManager.update(...)* riceve l'evento e, non essendo *DealEnded*, non esegue lo scoring con *handleDealEnded*, ma chiama *setChanged(), poi notifyObservers(argument)*. *GameController*, registrato presso il *GameManager*, esegue *update(...)*: riceve *ModelEvents.CardPlayed*, costruisce *new ViewEvent.CardPlayed(...)* usando i dati dell'evento e il *DealSnapshot*, e invia questo *ViewEvent* alla view tramite *publish(...)*. La view riceve *ViewEvent.CardPlayed* e aggiorna la UI usando lo snapshot immutabile. Gli eventi sono immutabili (record) e forniscono un *DealSnapshot*: la UI lavora su copie immutabili dello stato del model.

2.3 Listener

I listener sono concettualmente simili agli observer e ricevono eventi a cui reagiscono. I listener sono implementati attraverso interfacce build-in di

java, o anche costruite ad-hoc. Nel progetto sono particolarmente usate nella UI e spesso conducono ad azioni immediate, come quelle legate a pulsanti o mouse. Ad esempio in *HumanHandPanel* il bottone dei segni ha un *addActionListener(...)* che chiama *toggleSignMenu()* (e può anche suonare un effetto con *AudioManager.playClick()*), mentre i pannelli delle carte registrano listener del mouse per intercettare il click sulla carta da giocare. A differenza dell'Observer, il listener è locale e sincrono: riceve l'evento nell'EDT (Event Dispatch Thread) della UI ed esegue l'azione (aprire un menu, mandare un comando al controller, riprodurre un suono) senza passare attraverso il meccanismo di publish e subscribe del modello. È stato usato per evitare di aggiungere complessità di design per eventi legati alla UI.

2.4 Callback

I callback sono porzioni di codice passate a una classe o a un metodo perché vengano eseguite più tardi in risposta a un evento o a una condizione. Nel progetto sono usati soprattutto come oggetti eseguibili di tipo *Runnable* o come interfacce funzionali (Consumer<T>). In pratica si costruisce un'azione in un punto del programma e la si dà a un componente che la eseguirà quando serve (ad esempio attraverso il costruttore).

Esempi concreti: in *JTresette* viene creato un *Runnable* chiamato *backToMenu* e passato a *GamePanel* per permettere a quest'ultimo di ritornare al menu senza conoscere la logica di navigazione. *ProfilesPanel* riceve un *Runnable onBack* per la navigazione e un Consumer<UserProfile> *onProfileSelected* che viene chiamato quando l'utente sceglie un profilo. Il vantaggio principale è il disaccoppiamento tra chi definisce il comportamento e chi lo esegue. Rende flessibile la composizione dei comportamenti a runtime. Il limite è che usare solo *Runnable* fa perdere semantica sul significato dell'azione.

2.5 Constructor injection

La constructor injection consiste nel fornire a un oggetto le sue dipendenze attraverso il costruttore. Nel progetto la composizione degli oggetti avviene manualmente nella fase di bootstrap, in particolare dentro *JTresette*. Per questo possiamo dire che sia una *manual DI*.

Esempi pratici: *new GamePanel(gameController, playerNamesById, backToMenu)* e *new ProfilesPanel(profilesAdapter, back, onAvatarSelect)* hanno componenti che ricevono le loro dipendenze dall'esterno. *JTresette* funge da *composition root*: costruisce i servizi e li passa ai componenti che ne hanno bisogno.

2.6 Adapter e Port (ProfilesAdapter)

`ProfilesAdapter` è un intermediario tra la UI dei profili e il servizio di persistenza (`ProfileService`). Esso svolge la funzione di Adapter o Port: traduce e nasconde i dettagli di implementazione del servizio verso la parte di interfaccia utente.

Nel progetto l'adapter incapsula operazioni come lettura e scrittura dei profili, l'interazione con *SelectedProfileHolder* e le chiamate di utilità come *FantasyNameProvider*. Così la UI chiama metodi dell'adapter e non deve conoscere dove o come i dati vengono salvati. Il vantaggio è l'isolamento delle scelte implementative dalla UI e migliore mantenibilità

3 Le classi di TreSette

3.1 Il main: JTresette

La classe *JTresette* è il punto di ingresso dell'applicazione. È una classe con soli metodi statici, pensata per inizializzare l'interfaccia grafica, collegare i servizi del profilo, costruire i pannelli principali e avviare la partita. Il metodo main esegue l'avvio nell'Event Dispatch Thread (EDT) chiamando *SwingUtilities.invokeLater*, quindi tutte le operazioni di creazione dell'interfaccia avvengono su questo Thread.

Il metodo main costruisce un oggetto *GameFrame* (la finestra), crea un *BackgroundLayer* che funge da contenitore centrale per i vari schermi, inizializza il *ProfileService* con la directory dei profili (sotto la home dell'utente) e l'adapter *ProfilesAdapter* che mette a disposizione le operazioni di gestione profili per la UI. Quindi chiama *buildMainMenu* per creare il pannello del menu principale, lo inserisce come centro del *BackgroundLayer*, imposta lo schermo sul frame e rende la finestra visibile. Infine mostra una splash screen tramite *showSplash*.

buildMainMenu crea un *MainMenuPanel* e imposta un'implementazione anonima di *MainMenuPanel.MainMenuActions* usando *setActions*. Questa implementazione definisce tre callback: *onNewGame* che apre la schermata di nuova partita tramite *showNewGamePanel*, *onProfiles* che apre la gestione profili tramite *showProfilesPanel* e *onExit* che chiude l'applicazione chiamando *System.exit(0)*. Il pannello del menu non contiene la logica di navigazione, ma espone le action che qui vengono fornite da *JTresette* per navigare tra le sezioni.

showNewGamePanel costruisce un *NewGamePanel* e copia l'avatar selezionato (se presente in *SelectedProfileHolder*) chiamando updateAvatar (per mostrare l'immagine dell'avatar). Poi imposta le Actions con un'implementazione anonima di NewGamePanel.Actions che espone due metodi: onBack e onStart.

onBack ripristina il menu principale come centro del *BackgroundLayer*. onStart riceve i parametri scelti dall'utente (difficulty e winningScore), costruisce la lista di *Player* chiamando buildPlayers, crea un *GameManager* con quei giocatori e il punteggio scelto dall'utente, costruisce un *GameController* legato al *GameManager*, costruisce una LinkedHashMap che associa gli id dei giocatori ai loro nomi, prepara unRunnable backToMenu che riporta al menu, crea un *GamePanel* passando *gameController*, playerNamesById e backToMenu, imposta il *GamePanel* come schermo pieno con background.setFull(gamePanel), collega l'aggiornamento delle statistiche di profilo con linkProfileStatsUpdate e infine chiama gameController.startGame().

buildPlayers costruisce la lista di quattro giocatori. Il primo è sempre un *HumanPlayer* con id "P1" e nome preso da *SelectedProfileHolder* quando presente (altrimenti usa la stringa "Tu"); chiama *FantasyNameProvider*.reserve(humanName) per evitare duplicati e poi crea tre *BotPlayer* con id "P2", "P3", "P4", usando *FantasyNameProvider*.next() per generare nomi e passando lo stato di difficoltà ottenuto da parseDifficulty. Questo metodo definisce l'ordine e gli id usati dal resto dell'applicazione.

linkProfileStatsUpdate aggiorna le statistiche del profilo selezionato quando una partita finisce. Se non esiste un profilo selezionato (*SelectedProfileHolder*.isSet() == false), non fa nulla. Altrimenti ottiene il profilo corrente e registra un *Observer* su gameController. L'observer controlla gli eventi ricevuti e, se l'argomento è un *GameEnded*, verifica se il giocatore umano ha vinto verificando la presenza di "P1" o "Team1" tra i winnerIds. Poi chiama profileService.recordGameResult con il nickname e il risultato; se il profileService restituisce un profilo aggiornato, lo imposta in *SelectedProfileHolder*.

showProfilesPanel costruisce la schermata di gestione profili. *JTresette* prepara un Runnable back che rimette il menu principale nel *BackgroundLayer*, e un Consumer<UserProfile> onAvatarSelect che, se l'utente scelto non è nullo, aggiorna l'immagine dell'avatar nel menu principale chiamando menu.updateAvatar(userProfile.getAvatarPath()). Poi costruisce un *ProfilesPanel* passando *ProfilesAdapter*, back e onAvatarSelect, e lo imposta come centro del *BackgroundLayer*.

showSplash costruisce una semplice splash screen che blocca le interazioni per un breve intervallo. Prende il glassPane del *GameFrame*, imposta il layout, disabilita ricorsivamente i componenti della content pane con setEnabledDeep(content, false), crea un *SplashOverlay* passando un Runnable che riabilita la UI e nasconde il glass quando la splash termina. Per impedire clic accidentali il metodo aggiunge listener vuoti al glass pane, svuota il glass, aggiunge lo splash e lo rende visibile.

setEnabledDeep abilita e disabilita ricorsivamente il componente Swing preso come argomento (esclusivamente lo splash) e tutti i suoi figli.

parseDifficulty converte la stringa scelta dall'interfaccia in un *GameDifficultyState*. Accetta MEDIUM e HARD; qualsiasi altro valore (incluso null) ricade su EASY.

Per riassumere, *JTresette* svolge il ruolo di direttore: crea istanze di view (*GameFrame*, *BackgroundLayer*, *MainMenuPanel*, *NewGamePanel*, *ProfilesPanel*, *GamePanel*), crea servizi di dominio e adapter (*ProfileService*, *ProfilesAdapter*), costruisce il modello e il controller per la partita (*GameManager*, *GameController*) e connette questi componenti tra loro passando callback (implementazioni di *MainMenuActions*, interfaccia interna di *MainMenuPanel* e NewGamePanel.Actions), *Runnable* per la navigazione e Consumer UserProfile per la notifica di selezione avatar. Le action impostate con setActions sono fornite da *JTresette* e definiscono la navigazione e l'avvio della partita. Il *GameController* è creato e passato al *GamePanel*, mentre l'aggiornamento delle statistiche del profilo è collegato registrando un *Observer* direttamente sul controller.

4 Il model

4.1 *Card*, *CardSuit*, *CardValue* e *Deck*

Card rappresenta una singola carta di gioco ed è immutabile: viene costruita con il costruttore public Card(CardSuit, CardValue) e fornisce metodi utili come getCode(), toString(), oltre a equals(Object) e hashCode() usati per il confronto e come chiave nelle mappe. *Card* è usata da *Deck* quando il mazzo viene inizializzato, compare nelle mani dei giocatori e viene passata ai metodi del gioco come GameManager.playHumanCard(Player, Card). La sua dipendenza principale sono gli enum *CardSuit* e *CardValue*. Il valore restituito da getCode() (ad es. SETTE_DENARI) viene usato come identificatore nelle comunicazioni tra model e controller.

CardValue è un enum che definisce i dieci valori di carta usati (per esempio SETTE, ASSO, RE) e incapsula tre attributi: il nome (getValueName()), il valore di confronto per determinare il vincitore di una presa (getGameValue()) e il valore di punteggio finale (getPoints()). *CardValue* è usato da *Card* durante la costruzione degli oggetti e viene consultato dal modello quando serve stabilire l'ordine delle carte in una presa o calcolare i punti associati alle carte vinte.

CardSuit è un enum con i quattro semi (DENARI, BASTONI, COPPE, SPADE) e il metodo getSuitName() per ottenere il nome del seme. È usato direttamente da *Card* e da *Deck* durante l'inizializzazione del mazzo.

Deck rappresenta il mazzo da gioco (40 carte) e viene costruita tramite il costruttore public Deck() che chiama initializeDeck() per creare tutte le combinazioni di *CardSuit* e *CardValue* (4 x 10). Espone operazioni come

shuffle() che mescola l'ordine interno delle carte, size() e isEmpty() per interrogare lo stato. È utilizzata dallla parte del model che si occupa della routine della mano, ovvero *Deal*/*Deal2v2*, e indirettamente dal *GameManager*. Dipende da *Card*, *CardSuit*, *CardValue* e da *EmptyDeckException* per la gestione dell'errore di pescaggio (eccezione presente, ma non rilevante per il 2vs2). Internamente usa una List<Card> mutabile.

4.2 *Table* e *Trick*

Trick memorizza le giocate di una singola presa in ordine e determina il palo di apertura. Tra i metodi: addPlay(Player, Card) che imposta il palo se è la prima carta giocata e registra la coppia player-card, getPalo() che restituisce un Optional<CardSuit>, e altri metodi di interrogazione. L'optional ritornato da getPalo() è utile perché è un metodo molto rilevate per la routine di gioco. Se manca il palo dopo la prima giocata, significa che c'è un errore grave nel codice. Per questo Deal gestisce l'optional con un getPalo().orElseThrow). *Trick* è utilizzata da *Table* e, indirettamente, da *Deal* per raccogliere le carte giocate. Le sue dipendenze sono *Card*, *CardSuit* e *Player*. La struttura interna usa una LinkedHashMap<Player,Card> per preservare l'ordine di gioco e restituire viste immutabili dello stato quando necessario.

Table rappresenta il tavolo di gioco e mantiene lo stato della presa corrente tramite un *Trick* interno. Fornisce operazioni principali come addCard(Player, Card), clearTableAndReturnCards() e metodi di interrogazione. È un componente del modello usato dalle classi che gestiscono il ciclo della mano (*Deal*/*Deal2v2*). *Deal* chiama table.addCard(...) e table.clearTableAndReturnCards() per risolvere la presa; consulta table.getPalo(). Altri componenti che consultano lo stato del tavolo sono *SignManager* (controlli sui segni) e i bot (*BotPlayer*/*BotStrategyEngine*). Le dipendenze dirette sono: *Trick*, *Card*, *CardSuit*, *Player*.

4.3 *Hand*

Hand è il contenitore per le carte dei giocatori e ha una capacità di 10 carte. Fornisce operazioni di gestione usate da *Player* e dalla logica di gioco (es. *Deal*: addCard(Card) per aggiungere, removeCard(Card) per rimuovere una specifica istanza, moveCard(int fromIndex, int toIndex) per riordinare. Espone viste immutabili del proprio stato con getAllCards() e getAllCardsCode(), oltre a utilità come size(), isEmpty() e clear().

4.4 *Player*

Player è la classe astratta che ha come campi degli identificativi (id, username), campi che ne definiscono lo stato, *Hand* per la mano, lista delle carte vinte e l'eventuale teamId. Fornisce il costruttore Player(String id, String

username), l'operazione di gioco playCard(Card), i metodi per manipolare la mano (addCard(Card), moveCard(int, int)) e la gestione del punteggio con addWonCards(List<Card>) e resetForNewGame(). I metodi per conoscere lo stato sono: getHandCards(), getWonCards(), getId(), getUsername().

4.5 *HumanPlayer*

HumanPlayer è una sottoclasse minimale di *Player* che non aggiunge stato né comportamenti specifici. Il costruttore HumanPlayer(String id, String username) delega alla superclasse. La sua presenza è puramente semantica.

4.6 *BotPlayer*

BotPlayer estende *Player* delegando la logica decisionale a *BotStrategyEngine*, principalmente per via della sua complessità. Il costruttore BotPlayer(String id, String username, GameDifficultyState) costruisce lo strategy engine con il livello di difficoltà scelto. Per le decisioni espone decideCard(Table) (chiama strategyEngine.chooseCard(table, handCards)), decideSign(Table) (chiama strategyEngine.chooseSign(table, getHandCards())) e onSign(SignEvent) (inoltra l'evento a strategyEngine.observeSign(event)). Le dipendenze sono: *Table*, *SignType*, *GameDifficultyState* e *SignEvent*.

4.7 *BotStrategyEngine*

BotStrategyEngine è il motore decisionale dei giocatori bot. Prima di tutto costruisce le mosse legali con legaMoves() (se esiste palo impone il Palo, altrimenti tutte le carte sono legali) e, se la lista legale contiene una sola carta, la restituisce subito. Subito dopo si applica il fattore di casualità (actionNoise): con una probabilità legata alla difficoltà. Più il gioco è difficile, meno sarà probabile che il bot giochi in modo casuale. Se non c'è ancora un palo (cioè il bot è il primo a giocare), la scelta passa per due percorsi. Se il bot aveva pianificato in precedenza un BUSSO (planned BussoPalo non nullo) favorisce quel seme e sceglie la carta di quel seme con il massimo valore di punti (usa highestPointsOfSuitOrLowest), per dare coerenza al segno inviato. Se non c'è, chiama findBestLeadCard: questa funzione prende la carta con il valore di gioco più alto e la gioca se supera la soglia di "forte" (la costante STRONG _ GAME _ VALUE, ad esempio carte come Asso), altrimenti ritorna la carta più debole (minByGameValue). In pratica il bot gioca aggressivo quando ha risorse forti, altrimenti è conservativo e gioca la più debole. Quando c'è già un palo sul tavolo l'engine valuta la situazione concreta delle carte già piazzate. Calcola la carta attualmente vincente tramite GameRules.getWinningCard e determina se quella carta appartiene al compagno (usando l'oggetto Team) per capire se è il caso di non sovrapporsi. Se il compagno sta vincendo e il bot può seguire il palo, il comportamento è conservativo: gioca la carta più bassa legale del palo (minByGameValue(legal))

per non sprecare risorse. Se non può seguire, sfrutta l'occasione per scartare punti: sceglie la carta con il massimo valore in punti nella lista legale. Se il compagno non è vincente cerca invece di ribaltare la situazione: costruisce la sottolista di legal, winning, contenente le carte legali che battono la carta vincente (usando GameRules.cardBeats) e, se esistono, sceglie la più economica tra quelle che vincono (minByGameValue(winning)). Quindi tenta di vincere con carte economiche. Se non esistono carte vincenti, scarta la più debole. La scelta del segno è separata: chooseSign calcola un segno ideale con computeIdealSign. Per prima cosa cerca un BUSSO con selectBussoSuit, che raggruppa le carte per seme e considera qualificante un seme con almeno due carte forti ($game\ value \geq STRONG\_GAME\_VALUE = 8$) oppure un seme con una top ($game\ value \geq STRONG\_GAME\_VALUE+1$)e almeno una seconda carta di quel seme. Seleziona il seme migliore privilegiando il numero di forti, poi il numero di top, poi la dimensione. Se trova un seme così, imposta plannedBussoPalo e restituisce BUSSO. Se non trova BUSSO valuta la quantità di carte che danno punti: se ci sono molte carte con punti restituisce LISCIO. Se non ha carte forti e ha pochi punti restituisce VOLO. Altrimenti NONE.

observeSign è un punto d'estensione: viene chiamato quando arriva un SignEvent e oggi è no-op. Per ragioni di tempo si è deciso di non portare a termine questa logica. Idealmente il BotStrategyEngine deve capire cosa fare quando riceve un segno dal compagno, adottando una logica specifica per ogni segno. Sarà possibile implementare questa logica in un futuro refactoring.

Gli helper: legalMoves filtra per palo; minByGameValue restituisce la carta con il minore valore di gioco (per scartare o giocare conservativamente); highestPointsOfSuitOrLowest è una scelta pensata (dal bot) per inviare il segno di BUSSO; findBestLeadCard implementa la regola "gioca la carta più forte se è nettamente forte, altrimenti la più debole" basandosi sui valori di CardValue.

4.8 Team

Team rappresenta un team di Tre Sette identificato da un id e composto da al massimo due *Player* membri. Il costruttore Team(String, List<Player>) valida la dimensione e i duplicati, copia la lista fornita, assegna il teamId ai membri tramite *Player*.assignTeam(String) e rende la lista interna immutabile. Espone metodi accessori semplici, getId() e getMembers(), e metodi di utilità come getCurrentDealRawPoints() che somma i punti delle carte vinte dai membri, e contains(*Player*) per verificare l'appartenenza. L'idea di calcolare solo i RawPoints deriva dal fatto che la responsabilità del calcolo è affidata a *ScoreManager* e *ScoreCalculator*.

4.9 SignManager e SignType

SignManager applica le regole di invio dei segni durante una mano: mantiene la lista dei players (con l'umano all'indice 0) e il boolean signUsedThisTrick per assicurare che sia emesso al massimo un segno per trick. Espone canPlayerMakeSign(*Player*, *Table*, *Player*) per verificare le condizioni (turno in corso, tavolo vuoto e segno non ancora usato) e sendSign(*Player*, *SignType*, *Table*, *Player*) che crea l'evento *SignEvent* e setta signUsedThisTrick true. Fornisce inoltre resetDeal() e onTrickEnded() per resettare lo stato.

SignType è l'enum che definisce i possibili segni utilizzabili: NONE, BUSSO, VOLO e LISCIO.

4.10 ScoreManager e ScoreCalculator

ScoreManager si occupa di coordinare i punteggi: gameScores, che accumula i punteggi delle mani in una mappa (id-punteggio), e lastDealTeamPoints, che salvano i punteggi di fine mano (id-punteggio). Gli altri campi sono: il riferimento a *ScoreCalculator*, il lastDealWinnerId, il winningScoreTarget e la lista finalWinnerIds. Il metodo principale updateTeamGameScores(*List*<*Team*>, *String*) calcola i raw points con calculator.rawTeamPointsFromTeams, arrotonda con calculator.roundRawPoints, applica eventuale bonus di fine mano e la regola del cappotto, salva lo snapshot in lastDealTeamPoints e accumula i punti nel gameScores (tramite accumulate()). checkForGameWinner() verifica il raggiungimento di winningScoreTarget ed eventualmente aggiorna finalWinnerIds con gli id dei vincitori e restituisce true se il gioco è finito. La classe espone accessori come getScore, getAllScores, getLastDealTeamPoints, getFinalWinnerIds e getWinningScoreTarget per consultare lo stato.

ScoreCalculator è l'utility usata da *ScoreManager* che incapsula la logica di calcolo dei punteggi: fornisce rawTeamPointsFromTeams(*List*<*Team*>) per aggregare i punti grezzi (double) dalle istanze di *Team*, roundRawPoints(Map<String,Double>) per arrotondare i punti grezzi secondo le regole del progetto (in questo caso si è scelto di arrotondare per eccesso se la somma a la parte decimale <= 0.9, perché significa che ci sono 3 carte con 1/3 di valore), applyWinnerBonus(Map<String,Integer>, *String*) per applicare il bonus dell'ultimo trick e applyCappotto(Map<String,Integer>) per trasformare i punteggi quando si verifica il cappotto. Segue le regole di model.*GameRules*.

4.11 Deal, Deal2v2, BotMoveScheduler

Deal dirige tutta la singola mano ed estende Observable. E' il componente che tiene l'ordine di gioco dei players, il *Deck*, la *Table*, il *SignManager* e i timer necessari allo scheduling dei bot. *GameManager* chiama i metodi di *Deal* in momenti rilevanti, come l'avvio della mano o quando il giocatore

umano deve fare delle scelte. Tuttavia, *Deal* dirige il resto della mano in autonomia. Le responsabilità riguardano: avviare la mano con start(), ricevere le giocate del player umano tramite playHumanCard(*Player*, *Card*), accettare le giocate dei bot tramite playCardFromBot(*BotPlayer*, *Card*), gestire i segni con handlePlayerSign(*Player*, *SignType*) e sospendere o riprendere il flusso con setPaused(boolean). Deal usa un metodo privato chiamato takeGameSnapshot() che scatta un'istantanea, *DealSnapshot*, esposta nell'invio agli observer. La notifica agli observer avviene usando i record di *ModelEvents*, ovvero: DealStarted, TrickStarted, CardPlayed, TrickEnded, DealEnded, Sign). Queste notifiche avvengono ogni volta che lo stato cambia in modo rilevante. Internamente *Deal* coordina la seguente sequenza: prima chiama resetPlayers() e initialDeal() per preparare le mani, poi decide chi parte (implementato in determineStartingPlayerIndex() nelle sottoclassi) e imposta il turno corrente. Quando arriva una giocata viene eseguita la logica centrale in executePlay(...): rimuove la carta dalla mano del giocatore, la mette in table, notifica *ModelEvents*.CardPlayed e poi, se tutti i giocatori hanno giocato, pianifica la risoluzione della presa con scheduleTrickResolution() (che crea un javax.swing.Timer che invoca onTrickResolutionTimer(...)). Allo scadere del timer, onTrickResolutionTimer chiama resolveTrick(), che determina il vincitore della presa, assegna le carte vinte, aggiorna lastTrickWinner, notifica *ModelEvents*.TrickEnded / *ModelEvents*.TrickStarted o *ModelEvents*.DealEnded se la mano è finita (isDealFinished()). Come si potrà immagine, i timer sonoo utili per dare aiutare la view a dare tempo al giocatore umano di vedere le carte giocate e la fine della mano. Una nota importante: poiché la risoluzione delle prese e lo scheduling dei bot usano javax.swing.Timer, le callback vengono eseguite sull'Event Dispatch Thread (EDT), il singolo thread che Swing usa per gestire eventi e aggiornare la UI. Per questo la logica in resolveTrick() e nelle callback dei timer deve essere breve, altrimenti congelerebbero l'interfaccia. Il vantaggio dell'uso di javax.swing.Timer è che le callback possono aggiornare direttamente la UI senza chiamare SwingUtilities.invokeLater(...).

Deal2v2 è una sottoclasse specializzata di *Deal* che implementa le regole della variante 2 vs 2. Questa variante è mantenuta distinta dalla classe *Deal* per permettere eventuali estensioni di modalità in futuro. Si occupa della scelta dell'indice di partenza, della condizione di fine mano e delle azioni post presa (ad es. logica che aggiorna chi comanda la mano successiva in base alla presa).

BotMoveScheduler è un classe helper che introduce un ritardo visivo prima che un bot esegua la sua mossa. Usa un javax.swing.Timer (callback eseguite sull'EDT) per: verificare se il giocatore corrente è un bot, ottenere la carta scelta (decideCard) e chiamare *Deal*.playCardFromBot. Gestisce l'annullamento e il ri-scheduling per evitare timer sovrapposti. Entrando nello specifico, scheduleIfBotTurn() determina prima il bot che sta giocando

il turno (usa this.currentBot(), che chiama deal.getCurrentPlayer()), poi iniziliazza *java.swing.Timer*.Passa al timer un java.awt.event.ActionListener() e sovrascrive l'azione da performare quando il timer scade. Le azioni che vengono impostate da compiere alla fine del timer sono i metodi di *BotPlayer* necessari a giocare il turno. Al termine di esse scheduleIfBotTurn() chiama timer.setRepeats(false), per impedire al timer di ripetere l'azione, e timer.start() per far partire il conto alla rovescia. La decisione di separare questa classe da *Deal* è dovuta alla necessità di avere un codice più pulito e testabile.

4.12 GameManager

GameManager è il coordinatore della partita: crea e avvia le mani, inizializza e coordina currentDeal, aggiorna i punteggi tramite *ScoreManager* e funge da inoltro degli eventi verso il *GameController*. Quando il controller vuole avviare una mano chiama *GameController*.startNextDeal() (o startGame()), che delega a *GameManager*.startNextDeal(). startNextDeal() verifica lo stato della partita (fine o partita già in atto), crea un nuovo *Deal2v2*, chiama attachToDeal(currentDeal) (che esegue deal.addObserver(this)), applica la pausa se necessario (currentDeal.setPaused(true)) e infine invoca currentDeal.start(), poi il dealCounter viene incrementato. Durante l'esecuzione, la *Deal* notifica eventi di modello usando notifyObservers(...) con istanze di *ModelEvents*, come DealStarted, TrickStarted, CardPlayed, TrickEnded. Poiché *GameManager* è registrato come osservatore della *Deal*, il suo metodo update(Observable, Object) viene invocato per ogni evento. *GameManager*.update(...) tratta all'interno l'evento DealEnded: chiama handleDealEnded(*DealSnapshot*) che traduce l'ultimo vincitore della presa in un vincitore del team, poi chiama scoreManager.updateTeamGameScores(...), ricava i punti dell'ultima mano con scoreManager.getLastDealTeamPoints() e poi notifica *ModelEvents*.ScoresUpdated. Se viene rilevato un vincitore di partita con scoreManager.checkForGameWinner() emette anche *ModelEvents*.GameEnded e imposta gameOver = true. Per tutti gli eventi (incluso DealEnded) *GameManager* chiama setChanged() e notifyObservers(argument), cioè inoltra lo stesso *ModelEvents* a chi lo osserva, ovvero. *GameController*,che si è registrato come osservatore nel suo costruttore. Quando riceve un *ModelEvents*, il suo update(...) traduce ciascun *ModelEvents* in un corrispondente *ViewEvent* (per esempio da *ModelEvents*.CardPlayed a *ViewEvent*.CardPlayed) e poi pubblica il *ViewEvent* verso la UI. Questa scelta di design è stata orientata a marcare il disaccoppiamento tra Model e View. Rispetto al flusso inverso, le chiamate dell'interfaccia utente arrivano al modello tramite *GameController* che risolve id e oggetti e delega a *GameManager*: playCard(...) invoca gameManager.playHumanCard(...) che a sua volta delega a currentDeal.playHumanCard(...); lo stesso vale per makeSign, l'invio del segno, o l'azione di pausa, resume e di stop della partita. Dal punto di vista delle

invarianti, *GameManager* assume che gli eventi provenienti da *Deal* siano coerenti con lo snapshot della mano (anche perché sono creati sempre da quest'ultimo) e che *ScoreManager* riceva team e giocatori corretti.

4.13 DealSnapshot, ModelEvents e SignEvent

DealSnapshot è un oggetto immutabile che rappresenta lo stato corrente di una mano e viene costruito da Deal.takeGameSnapshot() per trasferire tutte le informazioni rilevanti al controller e alla UI, tramite *GameManager* come visto. Contiene l'indice della mano (dealIndex), l'id del giocatore corrente (currentPlayerId), le dimensioni delle mani (handSizes) e il conteggio delle carte vinte (wonCards) per ciascun giocatore, l'elenco dei codici delle carte attualmente sul tavolo (tableCards), l'id dell'ultimo vincitore di presa (lastTrickWinnerId), un flag che segnala se il giocatore corrente può emettere un segno (canCurrentPlayerSign), lo stato di pausa e infine la mano concreta del giocatore umano (humanHand). Tutte le collezioni vengono copiate in modo immutabile nel costruttore, assicurando che lo snapshot non venga mutato dopo la creazione. Da questo punto di vista, SnapShot è una classe neutra, utilizzabile dalla view, evitando che essa possa agire sulle istanze del model.

ModelEvents è un contenitore dei record immutabili che modellano gli eventi tra *Deal*, *GameManager* e *GameController*. Ogni evento implementa l'interfaccia Event (marker) e include eventi specifici della routine della mano (DealStarted, DealEnded), del ciclo di presa (TrickStarted, TrickEnded), di gioco (CardPlayed con playerId, codice carta e testo descrittivo, e Sign che incapsula un *SignEvent* e lo snapshot), e di punteggio e termine partita (ScoresUpdated, GameEnded). La scelta di record è dovuta al fatto che sono immutabili e generano automaticamente costruttori e metodi accessori. Inoltre, permette di raggruppare tutti gli eventi in *ModelEvents*. Come già spiegato, questi sono gli eventi del flusso che vengono notificati tra Observer/Observable tra model e controller.

SignEvent è usata per rappresentare un segno giocato da un giocatore. Contiene il *Player* mittente (sender), il tipo di segno (*SignType*) e un boolean che indica se il segno proviene dal compagno del giocatore umano (fromTeammateOfHuman). Fornisce metodi accessori e getDisplayMessage() che restituisce una stringa di testo leggibile, usata dall'interfaccia per mostrare la notifica del segno. Questo oggetto è usato in *ModelEvents*.Sign così che sia possibile trasportare informazioni sul segno insieme alla *DealSnapshot* che descrive il contesto in cui il segno è stato emesso. Anche in questo caso, la classe è final perché il segno non deve essere alterato da altre classi. Quindi, quando viene inviato un segno si crea una nuova istanza.

4.14 GameRules

GameRules è la classe final che racchiude le costanti e le regole fondamentali del gioco Tre Sette. In essa sono definite le costanti di gioco (numero di carte, carte per giocatore, la carta di partenza, il 4 di Denari e i vari target di punteggio) e le funzioni che formalizzano le regole sul palo, sul confronto tra carte e sull'individuazione del vincitore di una presa. In particolare, il metodo isValidPlay verifica se una giocata è lecita rispetto al palo presente sul tavolo (se il tavolo è vuoto la giocata è automaticamente valida; se il palo è presente il giocatore deve rispettarne il seme, a meno che non ne sia privo). Le operazioni di confronto, getWinningCard e cardBeats, usano il valore di gioco associato ai valori delle carte per stabilire quale carta, tra quelle con il palo corrente, è la più alta. getWinningCard restituisce la carta vincente tra le carte di una lista valida, mentre cardBeats determina se una carta "challenger" sovrasta la carta corrente vincente sotto il palo attivo. Infine, getTrickWinner riceve la mappa, con chiave giocatori e valore carta, e il palo e restituisce il giocatore che ha piazzato la carta di maggior valore del palo.

5 Profile

Prima di descrivere più nel dettaglio le classi di Profile, è utile spiegare il flusso e il collegamento con gli altri package.
L'interazione parte quando l'utente chiede di elencare, creare, cancellare o aggiornare un profilo. La view prende i dati inseriti (nickname e avatar) e li passa all'adapter (*ProfilesAdapter*), che fa da ponte verso (*ProfileService*). Quest'ultimo permette di mantenere separare la view dalla logica del profilo e ottimizzare il decoupling. Per creare un profilo l'adapter chiama profileService.create(...): se la creazione è possibile il servizio costruisce il nuovo *UserProfile*, lo mette nella mappa in memoria e chiede al repository di salvarlo su disco. Il salvataggio scrive l'oggetto in un file il cui nome è generato con safeFileName(nickname) + ".dat" usando *ObjectOutputStream* (serializzazione Java). *ProfileService*.create(...) restituisce l'oggetto creato oppure null in caso di duplicato. l'adapter converte il risultato in un booleano dato alla view (true se creato, false altrimenti). La view quindi, se il valore è valore true, chiude la dialog e chiama refresh() su *ProfilesPanel*; altrimenti mostra un messaggio tipo "Creazione fallita" (per esempio nickname già esistente). Quando *ProfilesPanel*.refresh() viene eseguito chiede all'adapter la lista aggiornata dei profili, che a sua volta la prende da *ProfileService*. La view ricostruisce le card dei profili e le aggiunge al contenitore: ogni *ProfileCard* legge i getter di *UserProfile* (getGameLost(), getGamesPlayed(), getGamesWon(), getWinRate()) per riempire le statistiche. Alla fine la UI chiama revalidate() e repaint() per rendere visibili i cambiamenti.

5.1 Le classi di Profile

UserProfile è un oggetto immutabile che ha nickname, avatarPath e le statistiche di gioco (gamesPlayed, gamesWon, gamesLost). Ogni modifica alle statistiche viene applicata creando una nuova istanza tramite updateStats(...). Questo serve per preservare l'immutabilità. La classe implementa *Serializable* e dichiara serialVersionUID = 1L per gestire la compatibilità delle versioni serializzate. I getter espongono i dati utili alla view, ad esempio il numero di vittorie e il winrate.

Il salvataggio e il caricamento dei profili sono gestiti da *ProfileRepository*, che lavora su file con estensione .dat. Il metodo loadAll() apre la directory dei profili usando Files.newDirectoryStream(dir, "*.dat") e per ogni file chiama load(Path), che deserializza l'oggetto tramite *ObjectInputStream* e lo converte in *UserProfile*; file illeggibili o con contenuti non validi vengono ignorati catturando *IOException*, *ClassNotFoundException* o *ClassCastException*, così da non compromettere l'avvio dell'applicazione. Per salvare, save(UserProfile) si assicura che la directory esista (File.createDirectories(dir)), costruisce un nome file sicuro tramite safeFileName(nickname) + ".dat" e serializza l'oggetto su disco con *ObjectOutputStream*. delete(String) rimuove il file corrispondente con Files.deleteIfExists(...), ignorando errori minori per evitare blocchi. La funzion safeFileName(...) normalizza il nickname sostituendo i caratteri non ammessi con underscore, evitando nomi problematici.

ProfileService, mantiene una mappa in memoria (Map<String, *UserProfile*>) riempita all'avvio da repository.loadAll(). Espone operazioni semplici: list() restituisce i profili ordinati, create(...) verifica l'esistenza del nickname e, se libero, crea la nuova istanza, la registra in memoria e la passa al repository per il salvataggio; recordGameResult(...) prende il profilo esistente, costruisce una nuova istanza con le statistiche aggiornate e la salva immediatamente. delete(...) rimuove il profilo dalla mappa e richiama il repository per eliminare il file. In questo modo le modifiche hanno effetto immediato sullo storage e lo stato in memoria rimane coerente.

Lo stato del profilo selezionato è gestito da *SelectedProfileHolder*, un contenitore statico minimale che espone get(), set(*UserProfile*) e isSet(). Viene usato come stato condiviso tra i pannelli dell'interfaccia e per l'aggiornamento delle statitistiche a fine partita: è una soluzione semplice per mantenere accessibile il profilo corrente senza complicare il flusso di eventi.

6 Utils

AudioManager è una utility molto semplice che riproduce clip audio brevi presi dal classpath: ogni metodo statico (es. playClick(), playWinner(), playIntro(), ecc.) apre un nuovo *Clip*, lo avvia e registra un listener che chiude il *Clip*

quando finisce (il listener intercetta la fine della riproduzione), così i suoni possono sovrapporsi senza tenere risorse aperte a lungo. Gli errori audio vengono ignorati silenziosamente, per non interrompere l'app se l'audio non è disponibile. Nella UI viene usato con chiamate dirette e immediate dentro gli action listener dei bottoni: ad esempio, dentro l'actionPerformed di un pulsante si trova una riga come *AudioManager*.playClick() per dare feedback sonoro all'utente. Questa soluzione non ottimizza il disaccoppiamento tra view e utils, filosofia seguita in questo progetto, ma è una soluzione semplice per un gioco di questa dimensione.

FantasyNameProvider fornisce nomi italiani, principalmente tradizionali (per riflettere uno stile retrò), per i bot evitando ripetizioni finché la scorta non è esaurita (praticamente impossibile). Mantiene una lista base di nomi, una pool modificabile e un set di nomi riservati. Chiamando reserve(name) si impedisce che quel nome venga dato ai bot (utile quando un giocatore sceglie un nickname), mentre next() pesca casualmente un nome disponibile rimuovendolo dalla pool e segnandolo come riservato; se la pool è vuota genera un nome di fallback tipo Bot123.

7 La View

7.1 Flusso della view

Il flusso della view parte dall'avvio (main): viene creato il *GameFrame*, ovvero la finestra principale, e dentro di esso viene impostato il *BackgroundLayer*. Subito dopo si costruisce il menu principale (*MainMenuPanel*) e lo si inserisce nel background usando background.setCentral(menu), che lo centra sopra l'immagine di sfondo. A questo punto la finestra viene resa visibile. Quasi in parallelo viene chiamato showSplash: sul glass pane del frame viene inserito il *SplashOverlay*. Si usa il GlassPane perché è pensato appositamente per sovrapporre view temporanee alla finestra. Il glass pane blocca input e, quando lo splash termina, il callback nasconde il glass pane e riattiva i contenuti sottostanti. Da lì l'interazione dell'utente avviene nel *MainMenuPanel*: i pulsanti chiamano le azioni fornite (MainMenuActions). Se l'utente sceglie "Nuova partita", il flusso passa al metodo che costruisce il pannello (*NewGamePanel*) e lo include sempre tramite background.setCentral(). Quando l'utente conferma tramite passaggio al *GameController*, vengono creati i giocatori, il *GameManager* e il *GameController*, costruito il *GamePanel* (interfaccia della partita) e, a differenza dei menu, qui si usa background.setFull(gamePanel) per fargli occupare tutta l'area utile. Durante la partita gli eventi del modello (mosse, punteggi, fine gioco) passano dal *GameController* ai listener della view (il *GamePanel* si registra o viene agganciato indirettamente) e aggiornano l'interfaccia. Quando arriva un evento di fine partita il callback previsto (ad esempio "torna al menu") richiama di nuovo background.setCentral(menu)

ritornando allo stato iniziale spiegato, eccetto lo splash. Il passaggio tra ogni schermata avviene sempre attraverso il *BackgroundLayer*, che incapsula lo sfondo. Questo mantiene lo stile uniforme ed evita di sostituire ogni volta il contentPane del frame, quindi lo sfondo rimane costante.

Le classi del package *View.Common*(*AvatarComponents*, *BackgroundLayer*, *ImageResources*, *SplashOverlay*, *WMenuPanel*) contengono funzioni e componenti UI riutilizzabili in più schermate dell'app: utility per caricare e normalizzare risorse grafiche, container di sfondo e pannelli base per menu, componenti delle immagini degli avatar e lo splash che è usato all'avvio. Metterle in common evita un codice ridondante e fornisce rendering riutilizzabile per diverse classi.

ImageResources ImageResources è una classe per il caricamento delle immagini. Espone il metodo *load(String)* che restituisce un *Image* memorizzato in una *ConcurrentHashMap* per evitare caricamenti ripetuti. Internamente usa *ImageIO*.read(file) per caricare l'immagine. La cache usa *computeIfAbsent* per garantire che, a runtime, l'immagine venga caricata una sola volta per chiave. QUindi, chiamate successive restituiscono la stessa immagine nella cache.

AvatarComponents è un helper per creare e aggiornare *JLabel* che mostrano avatar. Espone setAvatarSize(int) per la dimensione condivisa, createAvatar(String, Integer) per creare un *JLabel*, e updateAvatar(JLabel, String, Integer) per aggiornare un'etichetta esistente.

BackgroundLayer è un pannello che disegna l'immagine di sfondo (ImageResources.TABLE) e applica un overlay scuro controllato da darkenAlpha. Fornisce metodi setCentral(Component) e setFull(Component) per posizionare il contenuto sopra lo sfondo. La logica di painting copia Graphics e disegna l'immagine, quindi applica la tinta nera. È la componente usata come pannello che contiene gli altri componenti e mantiene lo sfondo coerente tra schermate.

MenuPanel è un pannello base per schermate di menu, con layout verticale e helper per titolo, sottotitolo, pulsanti stilizzati e una box in stile wood (immagine in resources) che dipinge un'immagine di legno o un blocco colorato. Centralizza gli stili dei bottoni e il comportamento dei box in legno così tutte le schermate menu hanno lo stesso aspetto senza duplicare codice.

SplashOverlay è un componente grafico semplice che mostra il logo al centro con una durata fissa . Non gestisce di per sé il blocco degli input durante lo splash, responsabilità lascia a *JTresette* (si veda showSplash(...), rende visibile la glass pane, e disabilita la ricezione dei click del mouse.

MainMenuPanel è il pannello principale del menu di gioco, con pulsanti per avviare una nuova partita, gestire profili e uscire. I pulsanti hanno ActionListener che richiamano i metodi dell'interfaccia MainMenuActions (ad esempio onNewGame(), onProfiles(), onExit()) impostata con setActions(...). Prima di invocare l'azione, il codice riproduce un effetto sonoro chiamando AudioManager.playClick()

NewGamePanel è un pannello Swing per configurare e avviare una nuova partita (difficoltà, punteggio). Usa layout verticali (BoxLayout) e un box grafico che contiene un CardLayout per i sottomenu. I pulsanti hanno ActionListener che gestiscono la navigazione tra le schede e aggiornano lo stato interno (NewGameState). Il codice riproduce un effetto sonoro chiamando AudioManager.playClick() in ogni listener

NewGameState è una Classe di stato che memorizza le scelte dell'utente (difficoltà, modalità, punteggio) e fornisce etichette per i pulsanti. Non ha listener diretti, ma viene aggiornata dai listener dei pulsanti di NewGamePanel.

GamePanel è il contenitore centrale dell'interfaccia di gioco e riceve dal controller gli eventi adattati per la view che il controller riceve dal model. Nel costruttore la classe costruisce e dirige i sottocomponenti principali: la barra superiore (*GameTopPanel*), la vista del tavolo (*GameBoardView*), la barra delle carte del giocatore umano (*HumanHandPanel*) e il layer per popup di punteggio (*ScorePopupLayer*). Qui avviene il collegamento cruciale: il *GameController* viene memorizzato e registrato come *Observable* a cui *GamePanel* si iscrive (pattern Observer). Quando il controller pubblica un *ViewEvent*, GamePanel.update lo riceve, interpreta il tipo e intraprende azioni diverse: riproduce l'effetto audio relativo (es. carta giocata o fine partita), inoltra l'evento alla barra del giocatore umano e alla *GameBoardView*, e poi prova a estrarre uno snapshot di partita (*DealSnapshot*) dall'evento. Se esiste uno snapshot lo applica chiamando boardView.updateSnapshot(snap). In pratica *GamePanel* riceve gli eventi dal controller, coordina le reazioni globali (audio, stop/continue di gioco) e delega il rendering dettagliato alla *GameBoardView* e agli altri pannelli. Altre funzioni rilevanti: il metodo che apre o nasconde il menu di pausa (togglePauseMenu) mette il controller in pausa o resume e crea/ rimuove il *PauseOverlay* (schermata durante la partita), mentre l'azione di ritorno al menu (onMainMenu) chiama controller.stopGame() e invoca il callback backToMenu.

GameBoardView si occupa del rendering del tavolo di gioco: posiziona le carte sul tavolo, mostra le box dei bot, le pile vinte e le etichette con i nomi. Nel costruttore legge l'ordine dei giocatori, costruisce i suoi componenti interni (table panel, bot hand, won piles, name labels) e conserva

un *Consumer*<String> che viene usato per inviare brevi messaggi al top bar (per esempio notifiche testuali tipo "Carta giocata da P2"). Il metodo onEvent(ViewEvent) è il punto d'ingresso per gli eventi: la BoardView semplicemente li inoltra ai suoi sotto-componenti (tableBox, bot boxes) e poi usa un *EventMessageFormatter* per produrre, quando opportuno, un messaggio testuale che viene passato al consumer (e quindi mostrato nel top bar). La view non applica automaticamente tutti gli snapshot dentro onEvent nella versione corrente: l'aggiornamento dello stato completo avviene invece tramite updateSnapshot(*DealSnapshot*), chiamato dal *GamePanel*. updateSnapshot prende lo snapshot (stato della mano, indice della mano, ecc.), aggiorna le pile vinte e altre componenti visive e applica il relayout. Il layout è manuale: *GameBoardView* sovrascrive doLayout() e chiama relayout() che chiama i metodi di posizionamento (positionTableBox, positionPlayerBoxes, positionWonPiles, ecc.).

GameFrame è la finestra principale dell'applicazione. Nel suo costruttore imposta le proprietà base della finestra (titolo, dimensione fissa 1100×800, comportamento di chiusura) e un BorderLayout. Il metodo chiave è setScreen(Component), che sostituisce il contenuto principale del frame con il componente passato (il *BackgroundLayer* che contiene il menu o il *GamePanel*). In *JTresette*.main si crea il *GameFrame*, si costruisce il *BackgroundLayer* e si passa il menu principale: quando parte una partita *GamePanel* e tutte le sue viste vengono inserite come schermo con setScreen.

MainMenuPanel è il pannello principale che l'utente vede all'avvio: eredita lo stile e gli helper da *WMenuPanel* per il titolo, sottotitolo, bottoni stilizzati e la wood box che contiene i controlli. Il pannello è pensato per essere indipendente dalla logica dell'app: espone una interfaccia MainMenuActions con tre callback (nuova partita, gestione profili, uscita) che *JTresette* imposta tramite setActions. Quando l'utente preme i pulsanti, *MainMenuPanel* riproduce un click e attiva la callback relativa. L'avatar è mostrato usando *AvatarComponents* per creare e aggiornare il *JLabel* avatar: questo lo rende coerente con gli altri menu.

NewGamePanel estende lo stesso stile *WMenuPanel* ma offre un'interfaccia più ricca per configurare la nuova partita. La classe contiene uno stato locale (*NewGameState*) che mantiene la difficoltà e il target di punti scelto dall'utente. L'interazione avviene con card layout interno: c'è una schermata principale con i pulsanti "Difficoltà", "Punteggio", "Avvia partita" e "Indietro", e due sottomenù (difficoltà e punteggio) che vengono mostrati alternando le card quando l'utente seleziona le opzioni. I bottoni usano ActionListener anonimi che riproducono il suono di click e aggiornano lo stato. Per avviare realmente la partita *NewGamePanel* espone un'interfaccia Actions che *JTresette* imposta con setActions; il pulsante "Avvia Partita" invoca

actions.onStart(state.getDifficulty(), state.getWinningScore()). Anche qui l'avatar è gestito tramite *AvatarComponents*.

*NewGameState*è una piccola classe di supporto che incapsula le scelte dell'utente: la difficoltà e il punteggio di vittoria. Fornisce getter e setter semplici e un metodo difficultyLabel() che fornisce la rappresentazione testuale in italiano.

PauseOverlay è il pannello sovrapposto mostrato quando l'utente mette in pausa la partita. Eredita da *WMenuPanel* per lo styling "wood" e i pulsanti coerenti. La classe riceve da *GamePanel* un oggetto Actions con tre callback (continua, menu principale, esci) che vengono chiamate quando l'utente preme i rispettivi pulsanti. Ogni pulsante esegue prima AudioManager.playClick() e poi la callback corrispondente: ad esempio action.onMainMenu() libera il controllo al *GamePanel*, che poi chiama controller.stopGame(), hideOverlay() e ritorna al menu principale. action.onContinue() invece provoca hideOverlay() e controller.resume(). action.onExit() chiama System.exit(0). *PauseOverlay* viene aggiunto come layer sopra *GameBoardView* da GamePanel.togglePauseMenu() e delega tutta la logica di pausa/ritorno al gioco al *GamePanel*/*GameController*.

CardComponent è il componente grafico che rappresenta la faccia di una carta. Si occupa esclusivamente del rendering: carica l'immagine corrispondente tramite *ImageResources*, disegna la carta e applica l'effetto di evidenziazione quando richiesto. Viene creato da *HumanCardsPanel* per ogni carta della mano e espone metodi semplici per ottenere il codice della carta e per cambiare lo stato di evidenziazione.

HumanCardsPanel è il contenitore che organizza e gestisce i *CardComponent* della mano umana. Gestisce l'interazione dell'utente: click per tentare la giocata, drag-and-drop per riordinare le carte. Quando l'utente esegue un'azione valida, il pannello invia comandi al *GameController* (playCard, moveHumanCard) e si aggiorna sulla base delle risposte del modello.

HumanHandPanel incapsula *HumaCardsPanel*, il pulsante per i segni e il relativo *SignMenuPanel*. Riceve eventi tramite SnapshotUtil.extract e, quando arriva uno snapshot, aggiorna la mano, determina la disponibilità del pulsante Segni e resetta lo stato interno dei componenti. Coordina la presentazione della mano e traduce l'interazione utente in chiamate al *GameController* (es. makeSign).

BotHandComponent rappresenta graficamente la mano di un avversario mostrando dorso delle carte sovrapposte e il conteggio delle carte rimanenti. Riceve gli stessi eventi di vista, estrae lo snapshot e aggiorna il cardCount,

quindi ridisegna se necessario

EventBoxPanel è il piccolo contenitore che mostra all'utente l'ultimo messaggio di gioco: è trasparente che visualizza testi brevi e mantiene l'ultimo messaggio per evitare aggiornamenti ridondanti.

EventMessageFormatter traduce gli eventi di gioco in frasi per l'utente. Riceve l'ordine dei giocatori e la mappa dei nomi e, per ciascun tipo di evento rilevante (giocata, segno, fine mano, fine partita), costruisce una frase sintetica in italiano o restituisce null quando non è necessario mostrare nulla. La sua funzione è separare la logica testuale dal codice di layout, permettendo di modificare i messaggi senza toccare la UI.

GameTopPanel organizza la porzione superiore della schermata di gioco: a sinistra presenta i punteggi, al centro il pulsante Menu e a destra l'area eventi. Gestisce il posizionamento dei widget, espone metodi per aggiornare i punteggi e per ricevere messaggi da visualizzare, ed esegue l'azione di menu quando l'utente preme il relativo pulsante. Non genera i testi degli eventi: li riceve già pronti e delega a *EventBoxPanel* la loro visualizzazione.

SignMenuPanel è un piccolo popup con stile "wood" che permette al giocatore umano di inviare un "segno" (Busso, Volo, Liscio). È pensato per essere aggiunto da *HumanHandPanel* nella sua *JLayeredPane* e usa il posizionamento assoluto. Il pannello espone un'interfaccia Actions che il genitore implementa per ricevere gli eventi: quando l'utente preme un pulsante il pannello invoca actions.onSend(type), mentre il bottone di chiusura invoca actions.onClose().

TableCardsPanel rappresenta l'area centrale del tavolo dove sono poste le carte giocate nella mano corrente. Riceve eventi di vista, estrae lo snapshot della mano e aggiorna la visualizzazione chiamando updateFromSnapshot, che ricostruisce i *CardComponent* per le carte attualmente sul tavolo. Il layout ha una spaziatura sufficiente per mostrare fino a quattro carte in ordine centrale. Il pannello è responsabile soltanto del rendering della "tabella".

WonPileComponent visualizza le carte vinte da un singolo giocatore come una piccola pila di dorsi sovrapposti. Mantiene un contatore wonCount aggiornato tramite updateFromSnapshot(*DealSnapshot*), calcola le dimensioni preferite in base al numero di pile visibili e disegna le immagini del dorso (o un fallback grafico se l'immagine non è disponibile).

ScorePopup è il componente che si occupa esclusivamente di presentare all'utente un riassunto quando una mano è terminata oppure quando la partita è finita. Internamente costruisce poche etichette e un pulsante di conferma: il metodo pubblico chiamato per mostrarlo (ad es. showDeal o showGame)

riceve i dati da visualizzare (mappe di punteggi, nomi dei vincitori) e una callback esterna che riceve da *ScorePopupLayer* che le inserisce nel suo costruttore . Tra queste c'è dealAction che chiama controller.confirmDealRestul() e viene fornita a ScorePopup.showDeal.

ScorePopupLayer è lo strato che collega la UI ai dati di gioco: mantiene lo stato necessario (punteggi cumulativi, punti dell'ultima mano, identificatori dei vincitori, flag di fine partita) e riceve gli eventi di View dal controller tramite il metodo pubblico onEvent(*ViewEvent*). Quando arriva un evento che aggiorna i punteggi (ScoresUpdated) la layer aggiorna i suoi dati e, se opportuno, chiede al *ScorePopup* di mostrare il popup di fine mano; quando arriva un evento di fine partita (GameEnded) aggiorna i punteggi finali e mostra il popup di fine partita. Oltre all'istanza di *ScorePopup*, ha un metodo per aggiornare la barra superiore dei punteggi (TopBarUpdater): quando i punteggi cambiano *ScorePopupLayer* chiama quel callback per tenere sincronizzato l'header dell'app.

8 Note sugli Stream

In questa sezione verrà esposta una parte dei metodi stream utilizzati nel progetto, cercando di spiegare nel dettaglio il loro funzionamento. La breve spiegazione iniziale di ogni paragrafo fa riferimento allo stream e non al metodo in cui è usato.

BotStrategyEngine.chooseCard usa uno stream per determinare se il compagno sta vincendo: isMateWinning = table.getCardsOnTable().entrySet() .stream() .anyMatch(e -> e.getValue().equals(currentWinning) && team .getMembers().contains(e.getKey())).
table.getCardsOnTable().entrySet() restituisce un Set<Map.Entry<Player, Card» delle carte giocate sul tavolo. stream() crea uno Stream<Map.Entry <Player, Card» anyMatch(...) trova la entry che il cui valore, ovvero la carta vincente è uguale alla currentWinning, e che allo stesso tempo conteine il giocatore membro del team del giocatore this.

BotStrategyEngine.chooseCard filtra le carte che battono la winningCard: winning = legal.stream().filter(c -> GameRules.cardBeats(c, currentWinning, palo)).toList(). legal è la lista di mosse legali. filter(...) trattiene solo le carte che, secondo le regole e il palo, battono l'attuale carta vincente. toList() raccoglie il risultato in una lista immodificabile.

BotStrategyEngine.highestPointsOfSuitOrLowest sceglie la carta di palo con più punti: legal.stream().filter(c -> c.getSuit() == suit).max(Comparator.comparingDouble(c -> c.getValue().getPoints())) .orElseGet(() -> minByGameValue(legal)). Filtra le carte legali del seme desiderato e prende quella

col punteggio più alto. Se non ne ha, ripiega sulla legale con valore di gioco minimo.

Deal2v2.isDealFinished verifica la fine della mano: getPlayers().stream() .allMatch(Player::hasNoCards). getPlayers() fornisce i giocatori nell'ordine; allMatch(...) torna true se tutti non hanno più carte, altrimenti false.

GameController.playCard ricerca una carta per codice: p.getHandCards() .stream().filter(c -> cardCode != null && cardCode.equals(c.getCode())) .findFirst().orElse(null). stream() sulla mano. filter(...) evita NullPointerException e confronta il codice. findFirst() prende la prima corrispondenza. orElse(null) restituisce null se assente.

Team.Team rileva duplicati tra i membri: copy.stream().distinct().count(). distinct() rimuove duplicati in base a equals. count() restituisce il numero di membri unici, confrontato poi con la size per validare.

GameBoardView.setupPlayerBoxes deriva gli id dei bot: playerIds.stream() .filter(id -> !id.equals("P1")).toList(). filter(...) rimuove l'id umano "P1". toList() produce l'elenco immutabile degli id bot, mantenendo l'ordine per il layout.

GameBoardView.setupPlayerBoxes filtra gli id bot: playerIds.stream().filter(id -> !id.equals("P1")).toList(). Crea uno stream degli id, filtra via "P1" (umano) e colleziona gli id bot mantenendo l'ordine, utile per il layout delle tre postazioni bot.

Deal.takeGameSnapshot estrae i codici carta dell'umano: players.get(0) .getHandCards().stream() .map(Card::getCode).toList(). Mappa le carte della mano del primo giocatore ai rispettivi codici stringa e le raccoglie per lo snapshot della UI.

Deal2v2.isDealFinished verifica la fine mano: getPlayers().stream() .allMatch(Player::hasNoCards). Ritorna true se tutti i giocatori hanno mano vuota, altrimenti false.

Deal2v2.determineStartingPlayerIndex cerca la carta iniziale: players.get(i) .getHandCards().stream() .anyMatch(c -> c.equals (GameRules .STARTING_CARD)). Per ciascun giocatore, verifica se possiede la carta di avvio (4 di Denari) e usa il primo match trovato.

GameController.playCard cerca la carta per codice: p.getHandCards().stream() .filter(c -> cardCode != null && cardCode.equals(c.getCode())) .findFirst().orElse(null). Filtra la mano confrontando il codice, oltre al check card-

Code != null, prende la prima corrispondenza o null se assente.

GameController.getPlayerIds mappa gli id dei giocatori: gameManager .getPlayers() .stream().map(Player::getId).toList(). applica getId a ogni istanza di Player per ottenere gli id dei giocatori che vengono poi inseiriti in una list.

GameRules.isValidPlay controlla se il giocatore ha il palo: player .getHandCards().stream() .anyMatch(c -> c.getSuit() == palo). anyMatch(...) verifica l'esistenza di almeno una carta del palo richiesto: se true e la carta giocata è non segue il palo, la mossa è invalida.

ScoreManager.checkForGameWinner trova il punteggio massimo: int max = gameScores.values().stream().max(Integer::compareTo).orElse(-1). Scansiona i punteggi correnti e restituisce il massimo (o 1 se vuoto), applicando compareTo di Integer, per valutare la condizione di vittoria.

ScoreManager.checkForGameWinner estrae gli id dei vincitori: winners = gameScores.entrySet().stream().filter(e -> e.getValue() == max e.getValue() >= winningScoreTarget).map(Map.Entry::getKey).toList(). Filtra le entry con punteggio massimo sopra soglia e agisce poi sulle chiavi (id partecipanti), poi raccoglie la lista dei vincitori.

ScoreCalculator.applyCappotto verifica il cappotto: boolean cappotto = matchPoints.values().stream().anyMatch(v -> v == 0). Controlla se esiste almeno un partecipante con 0 punti. il metodo poi applyCappotto() attiva poi l'assegnazione dei punti cappotto agli altri.

Hand.getAllCardsCode restituisce i codici delle carte in mano: playerCards .stream().map(Card::getCode).toList(). Mappa tutte le carte della mano ai loro codici stringa (tramite getCode) e restituisce una lista che sarà usata dalla UI.

9 Conclusione

Il progetto TreSette fa leva su scelte architetturali pragmatiche e modulari per realizzare un gioco completo: un model ben separato dalla view tramite observer e publish, un controller che funge da mediatore e componenti UI riutilizzabili consentono manutenzione. Il design generale è pensato per favorire l'estendibilità: nuove varianti di regole, miglioramenti della logica dei bot o altro. Tutto ciò può essere aggiunto con modifiche locali senza rompere l'intero sistema, rendendo il progetto un buon punto di partenza per successivi affinamenti e sperimentazioni.

10 Note Finali

Per la stesura di questo documento, ci si è avvalsi del supporto dell'assistente AI Gemini (versione Flash, Google) per la revisione grammaticale e la generazione parziale della documentazione Javadoc. L'AI è stata talvolta usata come metodo di ricerca per comprendere meglio l'uso di alcuni metodi build-in di Java. Si sottolinea che ogni suggerimento fornito dall'AI è stato attentamente supervisionato e approvato dall'autore, che si assume la piena responsabilità del contenuto finale.