SAPIENZA
UNIVERSITÀ DI ROMA

Progetto di Metodologie di Programmazione, 2024-2025
JTressette

Corso del Prof. Stefano Faralli

Realizzato da:
Gaia Scintu - 1913313
Corso Presenza M-Z

INDICE

STRUTTURA DELLA RELAZIONE

I. Descrizione generale del progetto pag 2

II. Decisioni progettuali relative alle specifiche richieste

- **A.** Gestione del profilo utente, nickname, avatar, partite giocate, vinte e perse, livello … pag. 3
- **B.** Gestione di una partita completa con un giocatore umano contro 1, 2 o 3 giocatori artificiali pag. 3-4
- **C.** Uso appropriato di MVC, Observer Observable e di altri design pattern pag. 4
- **D.** Adozione di Java Swing o JavaFX per la GUI pag. 5
- **E.** Utilizzo appropriato di stream pag. 5
- **F.** Riproduzione di audio sample pag. 6
- **G.** Animazioni ed effetti speciali pag. 6-7

III. Design pattern utilizzati, dove e perchè pag. 7 a 10

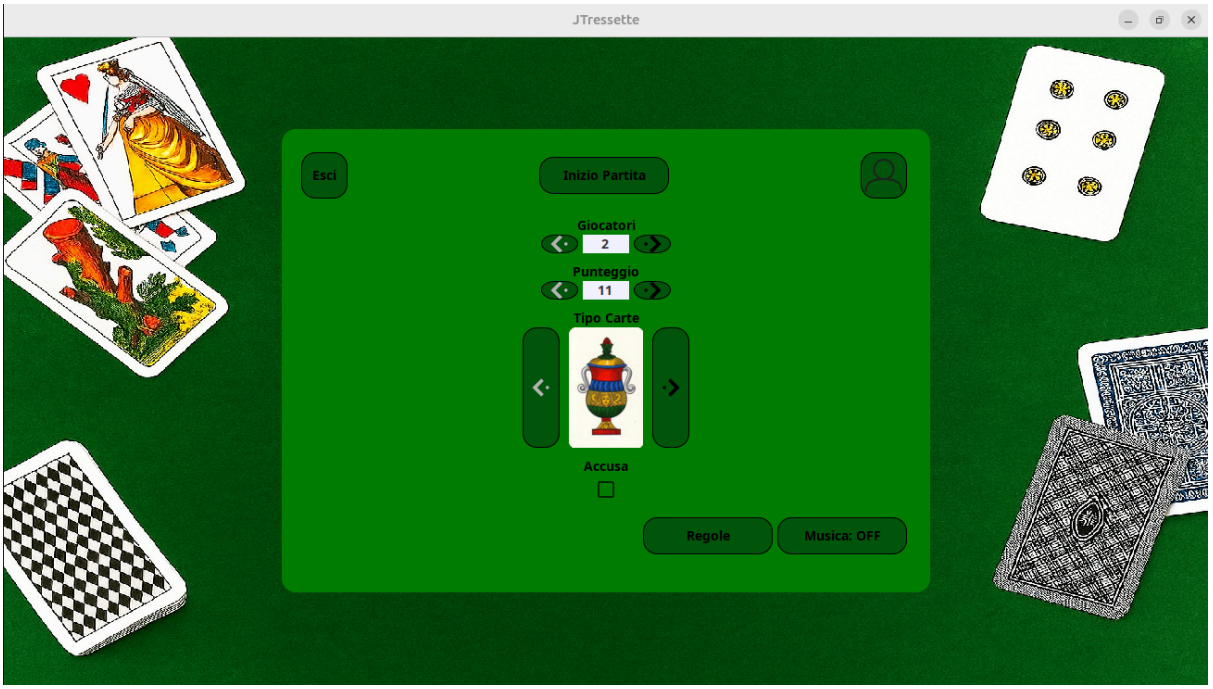
IV. Utilizzo degli stream (si trovano in partita tesette) pag. 10 a 14

I. DESCRIZIONE GENERALE DEL PROGETTO

Il progetto sviluppato ha come obiettivo la realizzazione del gioco di carte Tresette, utilizzando il linguaggio di programmazione Java e seguendo i principi della programmazione orientata agli oggetti. L'idea nasce dalla volontà di riprodurre digitalmente un gioco tradizionale, curando sia l'aspetto logico che quello grafico, per offrire un'esperienza interattiva e coinvolgente.

Dal punto di vista tecnico, il gioco è stato strutturato attraverso una serie di classi che rappresentano le componenti fondamentali: le carte, il mazzo, i giocatori, la partita e le regole. Ogni classe è stata progettata per gestire in modo autonomo le proprie responsabilità, favorendo la modularità e la leggibilità del codice. Questo approccio ha permesso di organizzare il progetto in maniera chiara e scalabile, facilitando eventuali estensioni o modifiche future.

Per quanto riguarda l'interfaccia grafica, è stata realizzata utilizzando la libreria Java Swing, con l'obiettivo di rendere il gioco accessibile e intuitivo. L'interfaccia consente all'utente di visualizzare le carte, effettuare le proprie mosse, seguire l'andamento della partita e accedere a funzionalità aggiuntive come la configurazione delle impostazioni, la consultazione delle regole e la gestione del proprio profilo utente. Pur essendo semplice, l'interfaccia è stata pensata per essere funzionale e facilmente utilizzabile.



II. DECISIONI PROGETTUALI RELATIVE ALLE SPECIFICHE RICHIESTE

A. Gestione del profilo utente, nickname, avatar, partite giocate, vinte e perse, livello…

Una delle specifiche richieste riguarda la gestione del profilo utente, elemento fondamentale per personalizzare l'esperienza di gioco e tracciare i progressi del giocatore. Per soddisfare questa esigenza, è stato realizzato un pannello grafico dedicato, integrato nell'interfaccia principale, che consente all'utente di configurare e visualizzare le proprie informazioni personali.

All'interno di questo pannello, l'utente può:

- scegliere un'immagine di profilo tra quelle disponibili,
- inserire o modificare il proprio nickname,
- consultare le statistiche di gioco, tra cui il numero di partite giocate, vinte e perse,
- visualizzare il proprio livello, calcolato in base alle prestazioni ottenute nel tempo.

Questi dati vengono salvati e aggiornati automaticamente al termine di ogni partita, permettendo una gestione coerente e persistente del profilo. La progettazione di questa sezione ha richiesto particolare attenzione all'organizzazione dei dati e alla loro integrazione con la logica di gioco, garantendo un aggiornamento fluido e una visualizzazione chiara all'interno dell'interfaccia.

Questa scelta progettuale ha contribuito a rendere il sistema più completo e orientato verso l'utente, migliorando l'interazione e offrendo un'esperienza più coinvolgente e personalizzata.

B. Gestione di una partita completa con un giocatore umano contro 1, 2 o 3 giocatori artificiali

Una delle funzionalità del progetto è la possibilità di avviare una partita completa in cui l'utente può giocare contro uno, due o tre giocatori artificiali. Questa modalità è stata pensata per offrire flessibilità e adattabilità, permettendo all'utente di scegliere il numero di partecipanti.

Per gestire questa funzionalità, è stato realizzato un pannello iniziale che consente all'utente di selezionare il numero di giocatori virtuali con cui desidera confrontarsi. L'interfaccia è semplice e intuitiva, e guida l'utente nella configurazione della partita prima dell'avvio. In base alla scelta effettuata, il sistema provvede a far visualizzare la schermata di gioco, adattando la disposizione delle carte, dei nomi e delle aree di interazione per riflettere il numero di partecipanti.

Ogni giocatore artificiale è gestito tramite una logica autonoma, che simula il comportamento reale di un avversario umano, rispettando le regole del Tresette e prendendo decisioni in base allo stato della partita. Questo approccio ha richiesto una progettazione attenta del flusso di gioco, con particolare attenzione alla sincronizzazione tra le mosse dei giocatori, alla gestione dei turni e all'aggiornamento dell'interfaccia in tempo reale.

La possibilità di giocare contro più avversari virtuali ha reso il progetto più completo e stimolante, offrendo all'utente un'esperienza di gioco variabile e personalizzabile.

C. Uso appropriato di MVC, Observer Observable e di altri design pattern

Nel progetto è stato adottato un approccio strutturato basato su alcuni dei principali design pattern, con l'obiettivo di garantire una buona separazione delle responsabilità, una maggiore manutenibilità del codice e una gestione efficiente dell'interazione tra le componenti.

In particolare, è stato implementato il pattern MVC (Model-View-Controller):

- Il Model contiene tutta la logica del gioco, comprese le regole, lo stato della partita, la gestione delle carte, ecc… .
- La View è responsabile della rappresentazione grafica, realizzata con Java Swing, e include tutti i pannelli visivi con cui l'utente interagisce.
- Il Controller funge da collettore e gestisce la comunicazione tra Model e View, interpretando le azioni dell'utente e aggiornando lo stato del gioco di conseguenza.

Per sincronizzare l'interfaccia con lo stato del gioco, è stato utilizzato il pattern Observer/Observable. In questo contesto, il modello è stato implementato come Observable, mentre i pannelli grafici agiscono come Observer. In questo modo, ogni volta che lo stato del gioco cambia, il modello notifica automaticamente il cambiamento a tutte le componenti osservatrici, garantendo un aggiornamento coerente e reattivo dell'interfaccia.

Oltre a MVC e Observer/Observable, sono stati adottati altri pattern creazionali:

- Il pattern **Singleton** è stato utilizzato per la classe **Utente**, poiché nel gioco è presente un solo giocatore umano, mentre gli altri sono virtuali. Lo stesso pattern è stato applicato al **GestoreAudio** e al **GestoreFile**, che si occupano rispettivamente della riproduzione dei suoni e della lettura/scrittura delle informazioni dell'utente.
- Il pattern **Factory** è stato impiegato per la creazione dei mazzi, con una superclasse astratta **MazzoFactory** dedicata che genera i due tipi di mazzi in base alla scelta del tipo di carte (Napoletane o Piacentine). Inoltre crea anche un mazzo contenente soli assi di coppe di entrambe le tipologie di carte per far scegliere all'utente quelle che preferisce.

L'adozione di questi pattern ha reso il progetto più solido, modulare e facilmente estendibile, facilitando l'integrazione di nuove funzionalità e la gestione del codice nel tempo.

D. Adozione di JavaSwing e/o JavaFX per la GUI

Per la realizzazione dell'interfaccia grafica del progetto è stata adottata la libreria Java Swing, principalmente per la sua semplicità d'uso e la facilità di apprendimento. Questa scelta si è rivelata particolarmente adatta al contesto del progetto, che non prevedeva la necessità di una grafica avanzata o particolarmente elaborata.

Java Swing è una libreria consolidata e ben documentata, integrata nativamente in Java, che offre tutti gli strumenti necessari per costruire interfacce grafiche funzionali. La sua ampia diffusione e la disponibilità di esempi e risorse online hanno facilitato lo sviluppo e il debug, rendendo il processo più fluido e accessibile.

In sintesi, la scelta di Java Swing è stata guidata dalla volontà di utilizzare uno strumento semplice, diretto e funzionale, perfettamente in linea con le esigenze di un progetto didattico come questo, focalizzato sulla realizzazione di un gioco di carte tradizionale con interfaccia grafica personalizzata ma non troppo complessa.

E. Utilizzo appropriato degli Stream

Nel progetto troviamo l'utilizzo di Stream, ovvero funzionalità che permettono di gestire in modo facile e veloce collezioni di dati. Permettono di eseguire operazioni come filtraggio, trasformazioni, ecc..

Questi, andando a sostituire i classici cicli, permettono di gestire in maniera più efficiente le operazioni da fare su collezioni di dati.

F. Riproduzione di audio sample

Nel progetto è stata implementata una classe dedicata alla gestione dell'audio, denominata Gestore Audio, con l'obiettivo di arricchire l'esperienza utente attraverso la riproduzione di effetti sonori contestuali e con musica di sottofondo. Questa componente centralizza il controllo dei diversi tipi di audio utilizzati durante la partita, garantendo coerenza e semplicità nell'integrazione.

Durante lo svolgimento del gioco, il Gestore Audio avvia una musica di sottofondo che accompagna il giocatore in modo continuo e discreto, contribuendo a creare un'atmosfera più coinvolgente. Oltre alla musica ambientale, la classe gestisce effetti sonori specifici che si attivano in risposta a determinate azioni:

- Quando l'utente **clicca un bottone**, viene riprodotto un suono che conferma l'interazione.
- Quando una **carta viene trasferita dalla mano al centro del banco**, viene emesso un effetto sonoro che simula il movimento e rafforza il feedback visivo.

La classe Gestore Audio è stata progettata in modo modulare, con metodi dedicati per ciascun tipo di suono, rendendo il codice facilmente estendibile e manutenibile. Questo approccio ha permesso di integrare l'audio in modo fluido con la logica di gioco e con l'interfaccia grafica, senza appesantire la struttura generale del progetto.

L'introduzione dell'audio ha migliorato significativamente l'interazione utente, rendendo il gioco più dinamico e realistico, pur mantenendo una struttura semplice e coerente con gli obiettivi didattici del progetto.

G. Animazioni ed effetti speciali

Per rendere l'interfaccia grafica più dinamica e coinvolgente, nel progetto sono stati integrati alcuni effetti speciali e animazioni che accompagnano le azioni principali del gioco. Pur mantenendo una struttura semplice, queste animazioni contribuiscono a migliorare l'esperienza utente e a rendere il gioco più intuitivo e gradevole.

Un altro effetto implementato è lo "shekeraggio" della carta quando questa non può essere selezionata. Questo accade, ad esempio, quando è stato dichiarato un seme (palo) e il giocatore tenta di giocare una carta di seme diverso pur avendo carte valide. L'animazione di scuotimento comunica in modo diretto che l'azione non è consentita, evitando errori e migliorando la comprensione delle regole.

Infine, al termine della partita, se il giocatore ha vinto, viene attivata un'animazione con coriandoli che celebra la vittoria. Questo effetto festoso aggiunge un tocco di leggerezza e soddisfazione, premiando il giocatore con un feedback visivo positivo.

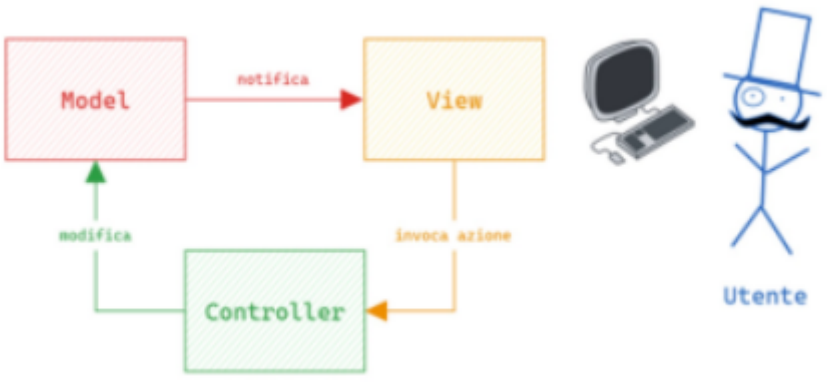
Tutte queste animazioni sono state realizzate con Java Swing, sfruttando timer e transizioni per simulare movimenti e cambiamenti visivi. L'integrazione è stata pensata per essere non invasiva, mantenendo la fluidità del gioco e rispettando la semplicità dell'interfaccia.

III. DESIGN PATTERN UTILIZZATI DOVE E PERCHE'

- ➔ **MVC - Observer/Observable**
- ➔ **Singleton**
- ➔ **Factory**

MVC - Observer/Observable

Si decide di usare il pattern architetturale MVC che utilizza il pattern comportamentale Oberserv/Observable. Si utilizza l'MVC perchè ha un modo semplice ma efficace per tenere separata la logica del gioco dalla parte grafica.



Classe Model (Modello-Observable)

La classe Model è responsabile della logica di gestione del gioco (gestione dello stato dell'utente, gestione creazione e controllo delle partite, gestione file, ecc..). Inoltre, Model estende la classe Observable che ha come ruolo quello di notificare le informazioni/cambiamenti verso le componenti registrate come Observer. In questo modo, ogni variazione rilevante nello stato del modello viene comunicato automaticamente alle viste, garantendo un aggiornamento coerente e reattivo dell'interfaccia utente.

Classe Controller (Controller)

La classe Controller viene utilizzata per gestire in modo chiaro il flusso di comunicazione tra la logica del gioco (Model) e l'interfaccia grafica (View). In pratica, viene utilizzato come un punto centrale che raccoglie le azioni dell'utente(dati/informazioni) e le inoltra al modello, senza mischiare codice grafico con la logica di gioco. Inoltre, ad ogni richiesta della View gestisce anche il passaggio di dati da visualizzare nella parte grafica del gioco.

Classe View(Vista-Observer) - Classe Pannello (Observer)

La classe View rappresenta la finestra principale dell'interfaccia grafica dell'applicazione. È il punto di accesso visivo per l'utente e funge da collegamento tra la GUI e il controller, gestendo la visualizzazione dei vari pannelli e delegando le azioni principali.

La classe Pannello rappresenta una base astratta per tutti i pannelli grafici dell'applicazione. Estende JPannel di Java Swing e introduce alcune personalizzazioni utili a garantire uno stile visivo uniforme. la classe implementa l'interfaccia Observer, permettendo ai pannelli derivati di ricevere notifiche dal modello ogni volta che lo stato del gioco cambia.

Singleton

Classe Utente

La classe Utente rappresenta il giocatore umano all'interno dell'applicazione. È progettata per essere istanziata una sola volta per sessione, in quanto il software prevede la presenza di un unico utente reale. Per garantire un accesso centralizzato e coerente a questa istanza in tutto il codice, la classe adotta il pattern Singleton.

Classe GestoreAudio

La decisione di implementare la classe GestoreAudio come Singleton nasce dall'esigenza di avere un controllo centralizzato della gestione audio all'interno dell'applicazione. Dal momento che gli effetti sonori e la musica di sottofondo devono essere coordinati e condivisi tra diverse componenti del sistema, è fondamentale che esista una sola istanza responsabile di queste operazioni. Il pattern Singleton garantisce proprio questo: evita la creazione di istanze multiple che potrebbero generare conflitti o ridondanze, semplifica l'accesso al gestore audio da qualsiasi parte del codice e assicura che tutte le riproduzioni sonore siano gestite in modo uniforme. Inoltre, considerando che l'applicazione è pensata per un singolo utente, l'approccio Singleton si adatta perfettamente al contesto, offrendo una soluzione elegante ed efficiente per la gestione dell'audio.

Classe GestoreFile

Nel progettare la classe GestoreFile, si è scelto di adottare il pattern Singleton per riflettere il ruolo di questa componente come punto unico di accesso alle operazioni di lettura e scrittura su file JSON. In un contesto applicativo, è importante evitare che più istanze indipendenti possano interferire tra loro o generare comportamenti incoerenti. Centralizzando l'interazione con i file, il Singleton consente di mantenere ordine e prevedibilità, riducendo il rischio di errori e semplificando la gestione delle eccezioni. Inoltre, questa scelta favorisce una maggiore chiarezza architetturale: ogni parte dell'applicazione sa esattamente dove rivolgersi per salvare o recuperare dati strutturati, senza dover istanziare nuovi oggetti. In definitiva, il pattern Singleton non è solo una soluzione tecnica, ma anche una scelta progettuale che contribuisce alla robustezza e alla leggibilità del sistema.

Factory

Classe MazzoFactory

La scelta di adottare il pattern Factory per la creazione dei mazzi di carte nasce dall'esigenza di gestire in modo flessibile e scalabile la varietà di mazzi previsti dall'applicazione, come quelli napoletani, piacentini o composti solo da assi. Ogni tipologia ha una struttura e una logica di composizione differenti, e incapsulare queste differenze in classi dedicate permette di mantenere il codice ordinato, modulare e facilmente estendibile.

Il pattern Factory si rivela particolarmente adatto in questo contesto perché consente di delegare la responsabilità della creazione degli oggetti (in questo caso, i mazzi) a sottoclassi specializzate, evitando che il resto dell'applicazione debba conoscere i dettagli specifici di ciascun tipo di mazzo. In questo modo, l'aggiunta di nuove varianti può avvenire senza modificare il codice esistente, ma semplicemente creando una nuova sottoclasse della factory.

Classe FactoryMazzoAssiCoppe

La classe FactoryMazzoAssiCoppe rappresenta una concreta applicazione del pattern Factory, progettata per generare un mazzo composto esclusivamente da assi di coppe, uno per ciascun tipo di mazzo previsto dall'applicazione. La scelta di creare una factory separata per gli assi di coppe risponde a un'esigenza funzionale ben precisa: offrire una versione semplificata del mazzo, utile in contesto come la scelta del tipo di mazzo.

Classe FactoryMazzoCartePiacentine

La classe FactoryMazzoCartePiacentine rappresenta una concreta applicazione del pattern Factory, progettata per generare un classico mazzo composto esclusivamente da carte di tipo Piacentine. La scelta di creare una factory separata per il tipo di mazzo risponde a un'esigenza funzionale ben precisa: offrire una versione personalizzata del mazzo, utile per fare scegliere all'Utente con quale tipo di carte preferisce giocare la partita.

Classe MazzoFactoryCarteNapoletane

La classe FactoryMazzoCarteNapoletane rappresenta una concreta applicazione del pattern Factory, progettata per generare un classico mazzo composto esclusivamente da carte di tipo Napoletane. La scelta di creare una factory separata per il tipo di mazzo risponde a un'esigenza funzionale ben precisa: offrire una versione personalizzata del mazzo, utile per fare scegliere all'Utente con quale tipo di carte preferisce giocare la partita.

IV. UTILIZZO DEGLI STREAM

Nel progetto gli stream sono stati utilizzati principalmente per la logica di gestione nella partita. In particolare possiamo trovarle nella classe PartitaTresette nei seguenti metodi:

- determinaPrimoGiocatore()

Questo metodo permette di determinare qual'è il giocatore a dover iniziare la partita rispetto a chi ha ricevuto il 4 di denari alla distribuzione delle carte.

In questo caso viene utilizzato lo stream per determinare il giocatore che nelle sue carte possiede il 4 di denari. Viene costruito uno stream sugli elementi della mappa dei giocatori, si filtra chi ha tra le proprie carte il 4 di denari, si estrae la chiave (ovvero il tipo di giocatore) e si prende il primo risultato possibile.

```java
// Cerca il giocatore che ha il 4 di denari
Optional<TipoGiocatore> found = giocatori.entrySet().stream()
    .filter(e -> e.getValue().getCarte().stream()
        .anyMatch(c -> c.getSeme() == Seme.DENARI && c.getValore() == Valore.QUATTRO))
    .map(Map.Entry::getKey)
    .findFirst();

if (found.isPresent()) {
    return found.get();
}
```

- verificaCartaScelta()

Questo metodo viene utilizzato per verificare se la carta scelta è valida rispetto al palo(ovvero la carta che determina il seme obbligatorio da giocare, se possibile). Nel corpo del metodo viene utilizzato lo stream per verificare se il giocatore che ha scelto la carta da giocare possiede almeno una carta dello stesso seme del palo.

Viene creato uno stream, sulla lista delle carte del giocatore e viene cercata almeno una carta dello stesso seme di quella del palo.

```java
// Caso 3: il seme è diverso -> va bene solo se il giocatore non ha carte del seme del palo
List<Carta> carteDelGiocatore = giocatori.get(TipoGiocatore.UTENTE).getCarte();
boolean hasSeme = carteDelGiocatore.stream()
    .anyMatch(c -> c.getSeme() == cartaPalo.getSeme());
return !hasSeme;
```

- giocaCartaPc()

Questo metodo viene utilizzato per definire della logica sulla scelta delle carte da giocare da parte del PC (isIA). Anche in questo caso la presenza della carta palo influisce sulla scelta della carta da giocare. Gli stream vengono utilizzati solo se la carta palo è presente.

Gli stream che vengono utilizzate in questo metodo sono due:

> Data la presenza del palo, il primo stream che incontriamo permette di filtrare le carte del pc per selezionare solo quelle che corrispondono al seme del palo. Questo permette di ottenere tutte le carte valide che potrebbe giocare il pc. Se il risultato dello stream è un insieme vuoto il pc sceglie una carta random tra le sue carte.
>
> Il secondo stream che incontriamo viene utilizzata invece se sono presenti carte dello stesso seme del palo; in questo caso lo stream viene utilizzato per cercare la carta con valore massimo (nella scala delle regole Tressette - vedi regole) e sceglie la più alta.

```java
// Filtra le carte con il seme desiderato
List<Carta> carteDelSeme = carteDelPc.stream()
    .filter(c -> c.getSeme() == cartaPalo.getSeme())
    .collect(Collectors.toList());

// Se non ci sono carte di quel seme, restituisci una a caso
if (carteDelSeme.isEmpty()) {
    int idxCartaRandom = new Random().nextInt(carteDelPc.size());
    cartaScelta = carteDelPc.get(idxCartaRandom);
}
else
{
    // Trova la carta con il valore massimo
    cartaScelta = carteDelSeme.stream()
        .max((a, b) -> Integer.compare(a.getValore().ordinal(), b.getValore().ordinal()))
        .orElse(carteDelSeme.get(0));
}
```

- verificaAccuse()

Questo metodo viene utilizzato per verificare la presenza delle accuse all'interno delle carte di un giocatore. All'interno del metodo vengono utilizzati una serie di stream per analizzare le carte del giocatore e verificare se il giocatore può dichiarare una o più accuse(NAPOLI, BONGIOCO, SUPER BONGIOCO - vedi regole)

Gli stream che vengono utilizzati per verificare l'accusa del NAPOLI sono i seguenti:

1) Viene utilizzato uno stream per organizzare le carte del giocatore in base al seme. Lo stream genera una mappa che ha come chiave i tipi di seme e come valori le liste delle carte associate. Questo stream viene utilizzato per verificare l'accusa del NAPOLI (presenza di ASSO, DUE e TRE dello stesso seme).
2) Il secondo stream, legato al primo, filtra e raggruppa per ogni seme solo le carte ASSO, DUE e TRE.
3) Se il secondo stream (2) genera dei raggruppamenti completi ovvero contenenti ASSO, DUE e TRE dello stesso seme, viene utilizzato lo stream che filtra le carte per valore ASSO, DUE e TRE e ne restituisce la loro lista.

```java
// Mappa: seme -> carte con quel seme
Map<Seme, List<Carta>> cartePerSeme = carte.stream()
    .collect(Collectors.groupingBy(Carta::getSeme));

// Verifica Napoli: asso, due, tre dello stesso seme
for (Map.Entry<Seme, List<Carta>> entry : cartePerSeme.entrySet()) {
    Map<Valore, List<Carta>> perValore = entry.getValue().stream()
        .filter(c -> c.getValore() == Valore.ASSO || c.getValore() == Valore.DUE || c.getValore() == Valore.TRE)
        .collect(Collectors.groupingBy(Carta::getValore));

    if (perValore.keySet().containsAll(Set.of(Valore.ASSO, Valore.DUE, Valore.TRE))) {
        List<Carta> napoli = Stream.of(Valore.ASSO, Valore.DUE, Valore.TRE)
            .map(perValore::get)
            .flatMap(List::stream)
            .collect(Collectors.toList());
        accuse.add(new Pair<>(Accusa.NAPOLI, napoli));
    }
}
```

Gli stream che vengono utilizzati per verificare l'accusa del BONGIOCO e SUPER BONGIOCO sono i seguenti:

1) In questo caso viene utilizzato uno stream per filtrare tutte le carte ASSO, DUE e TRE che vengono raggruppate per valore. Questo stream genera una mappa che ha come chiave il valore delle carte(ASSO, DUE e TRE) e come valore la lista delle carte corrispondenti per ogni tipo di seme disponibile.

   Il risultato di questo stream viene utilizzato per verificare il numero di ASSI, DUE e TRE per dichiarare il BONGIOCO(tre carte dello stesso valore) o SUPER BONGIOCO (quattro carte dello stesso valore)

```
// Mappa: valore -> carte con quel valore (solo asso, due, tre)
Map<Valore, List<Carta>> cartePerValore = carte.stream()
        .filter(c -> c.getValore() == Valore.ASSO || c.getValore() == Valore.DUE || c.getValore() == Valore.TRE)
        .collect(Collectors.groupingBy(Carta::getValore));

for (Map.Entry<Valore, List<Carta>> entry : cartePerValore.entrySet()) {
    List<Carta> gruppo = entry.getValue();
    if (gruppo.size() == 4) {
        accuse.add(new Pair<>(Accusa.SUPERBUONGIOCO, gruppo));
    } else if (gruppo.size() == 3) {
        accuse.add(new Pair<>(Accusa.BUONGIOCO, gruppo));
    }
}
```

Infine vengono utilizzati altri due stream per la dichiarazione effettiva delle accuse. In particolare vengono usati per controllare che le accuse dichiarate non vengano ripetute:

1) Si costruisce uno stream che verifica se le accuse presenti nelle carte correnti sono già state dichiarate. Se l'accusa non è ancora contenuta nella lista totale(quindi non ancora dichiarata), viene aggiunta al risultato dello stream per essere visualizzata nel pannello di gioco e contemporaneamente registrata nella lista delle accuse totali. In caso contrario, lo stream restituisce una lista vuota, evitando ripetizioni.

   Nota-> quando parliamo di lista, facciamo riferimento ad una lista di coppie così formata:

   List< Pair<Accusa, List <Carta> > >

2) Se lo stream precedente restituisce una lista di nuove accuse, viene utilizzato un secondo stream che filtra questa lista e costruisce la lista dei soli tipi di accusa che verrà utilizzata per il calcolo dei punteggi che il giocatore guadagna con ogni accusa.

```java
// Filtro le accuse già dichiarate
List<Pair<Accusa, List<Carta>>> nuoveAccuse = accuse.stream()
    .filter(nuova -> !accuseTotaliTurno.contains(nuova))
    .collect(Collectors.toList());

// Aaggiorno le accuseTotaliTurno
accuseTotaliTurno.addAll(nuoveAccuse);

if(nuoveAccuse.size()>0) {
    List<Accusa> listaAccuse = nuoveAccuse.stream()
            .map(Pair::getFirst)
            .collect(Collectors.toList());

    aggiornaPunteggiConAccuse(listaAccuse);
}
```