APPLICATIVO EDUCATIVO SUL GIOCO D'AZZARDO

Progetto di:

Rajiv Subramaniam

Roman Bernatskyi

Alessandro Tondo

Jeremias Arteaga

Roberto Pisano

Shaeek Siraj

Anno scolastico 2024 – 2025

Applicativo educativo sul gioco d’azzardo

Gli sviluppatori

Alla realizzazione di questo progetto hanno contribuito, nonché sviluppato un efficiente team:

- Alessandro Tondo: suggerimenti grafici, guess the number game, modalità automatica, stesura documentazione.
- Rajiv Subramaniam: menu principale, gestione file e account, stesura documentazione, modalità automatica, aiuto coordinamento.
- Roman Bernatskyi: project manager, dice game, stat. chart, presentazione, modalità automatica.
- Jeremias Arteaga: suggerimenti grafici, slot machine game, modalità automatica.
- Shaeek Siraj: idea base, classi e interfacce principali, coin flip game, modalità automatica.
- Roberto Pisano: roulette game, roulette pazza, chat bot, modalità automatica.

Gli strumenti utilizzati

Il progetto è stato realizzato seguendo la consegna data dal docente, mediante l’utilizzo di diversi strumenti e librerie messe a disposizione dal linguaggio Java.

Per il comparto grafico e per l’interfaccia utente sono state utilizzate le librerie di JavaFX, con i rispettivi controller e file fxml, programmati con l’ausilio dell’applicativo SceneBuilder.

Per la parte logica è stata utilizzata una versione recente del Java Developement Kit (Java SE-21).

La suddivisione dei compiti è stata resa possibile grazie alle proprietà della programmazione orientata a oggetti, che ha permesso di sfruttare quattro principi di base: astrazione, polimorfismo, incapsulamento ed ereditarietà.

Grazie a queste caratteristiche, e al modo in cui è stato progettato, il programma risulta essere modulare. Il modulo principale, composto dalle classi contenute nei package ‘main’ ‘fileManager’ ‘gameChart’ e ‘menu’, permette facilmente l’implementazione di nuove funzionalità, seguendo la logica predisposta con le interfacce.

Per garantire una realizzazione moderna e ottimizzata del codice, i componenti del gruppo hanno dovuto approfondire le conoscenze introdotte durante l’anno scolastico, aumentando la loro conoscenza tecnica del linguaggio Java.

Premesse, obiettivi e riflessioni

L’obiettivo del progetto è quello di educare sulle conseguenze del gioco d’azzardo e dimostrare che, per quanto uno possa credere di sapersi controllare, ci sono dei meccanismi intrinsechi nel cervello umano che sono difficilmente controllabili.

Per sfruttare questa caratteristica della mente umana, i casinò si avvalgono dell’uso di colori particolarmente accesi, scritte accattivanti e montepremi elevati, senza però far notare le perdite subite dai giocatori precedenti.

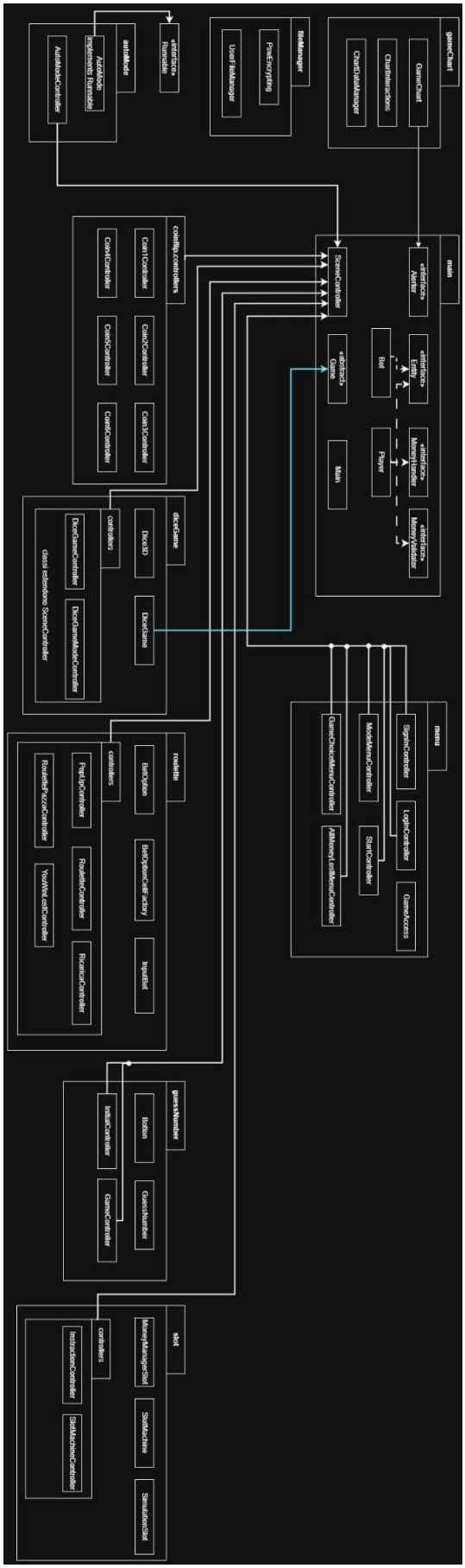
Per questo motivo, seppur si tratti di una simulazione, abbiamo adottato l’uso di colori tipici dei casinò, che, secondo gli psicologi, garantiscono una minor percezione dello scorrere del tempo da parte del giocatore, ma anche un aumento dell'impulsività e della fiducia in una possibile vittoria.

Nonostante i soldi in gioco non siano reali, le sensazioni e i comportamenti impulsivi del giocatore risultano essere simili alla realtà anche se meno accentuati. Il gioco, dunque, rende possibile una educazione sicura e con conseguenze limitate, cosa che non sarebbe possibile in un casinò reale. Ciò permette a chiunque di entrare direttamente a contatto con il gioco d’azzardo, sperimentandone il reale effetto e comprendendo i propri comportamenti.

Parliamo di conseguenze limitate in quanto il rischio di sviluppare una dipendenza dal gioco, anche se minimo è presente, poiché anche una simulazione virtuale potrebbe indurre a giocare d’azzardo.

Per rendere la simulazione realistica, il gioco non è truccato in alcun modo: la possibilità di vincere è tanto alta quanto quella di perdere tutto. Uno dei motivi che può aumentare il rischio di perdita è dovuto ai comportamenti impulsivi dei giocatori: vincere una modesta cifra in un gioco, potrebbe spingere il giocatore a puntare tutto ciò che ha vinto in precedenza con la probabilità di azzerare il guadagno.

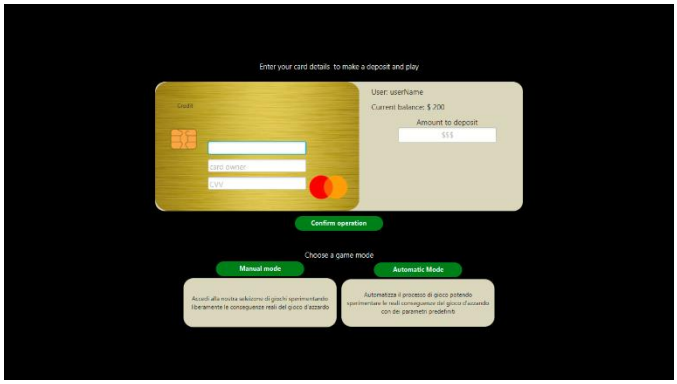
Schema UML semplificato



Manuale d'uso

All'avvio del programma si presenta una schermata di accesso, dove l'utente potrà creare un nuovo account, oppure accedere ad uno già creato.

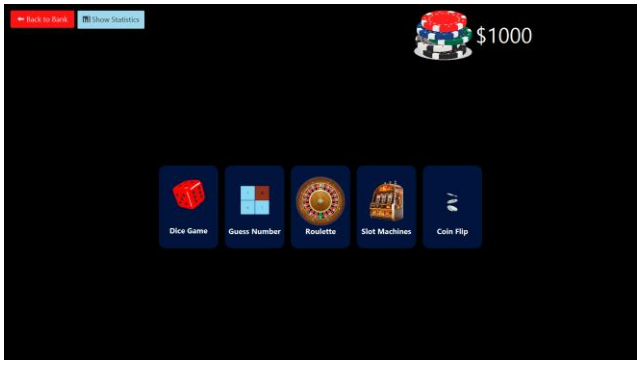
I dati dell'account vengono salvati in locale all'interno di una apposita cartella (denominata 'CasinoFiles'). Saranno accessibili solamente mediante la password scelta dall'utente, dunque gli accessi sono gestiti in modo criptato per garantire un discreto livello di sicurezza.



Dopo aver eseguito l'acceso, l'utente ha la possibilità di inserire dati bancari verosimili e scambiare le fiches, che verranno utilizzate nella simulazione. Ciò serve per conferire al gioco un maggiore livello di realismo, rendendo l'intera esperienza il più possibile verosimile e coinvolgente.

In seguito all'utente sarà richiesto di scegliere una modalità di gioco, manuale o automatica.

La modalità manuale permette di sperimentare direttamente le conseguenze del gioco d'azzardo, risultando essere migliore per conoscere emozioni e comportamenti.



Modalità manuale

Modalità automatica

Dal punto di vista statistico e analitico la modalità automatica è in grado di fornire una simulazione in tempo reale in modo molto rapido e con parametri di gioco preimpostati. L'utente può scegliere uno stile gioco più conservativo oppure rischioso, in base alla puntata minima e massima da lui selezionati, così come la durata della simulazione nonché i giochi da includere in essa.

Da un punto di vista logico, il progetto è strutturato in package, ovvero sottocartelle della source folder. Tutte le classi correlate sono all'interno di un package, consentendo una compressione ogni gioco presenta un package dedicato al cui interno sono presenti le varie classi che utilizza, ed è istruttivo osservare che ogni programmatore ha un modo di ragionare diverso; dunque, l'imposizione di alcune regole sullo sviluppo è stato fondamentale alla realizzazione di un progetto coeso e ottimizzato e ha reso più facile il bug detection e il debug stesso.

main

Il package principale si chiama main, all'interno del quale sono presenti tutte le classi che forniscono la logica di gioco. Dunque, abbiamo l'interfaccia Alerter che si occupa semplicemente di dare le basi per i messaggi di errore.

Successivamente abbiamo l'interfaccia Entity, la quale si occupa di fornire la base per la creazione di un'entità di gioco, come può essere il bot o il player.

La classe del bot implementa l'interfaccia Entity, e contiene tutto il necessario per il funzionamento del bot dei messaggi, al quale vengono fornite le informazioni dalle sottoclassi dei giochi, che vengono elaborate e restituite.

La classe player implementa diverse interfacce, tra le quali Entity, MoneyHandler, MoneyValidator. Si occupa di fornire tutto il necessario per la gestione del giocatore, possiede dunque il nome, l'utilizzo dei soldi, contiene il numero dei round vinti e persi, il record di guadagno, una verifica del budget minimo per iniziare a giocare e un rating di vittoria/sconfitta. Fornisce tutto il necessario all'utilizzo dell'account.

Inoltre, abbiamo la classe SceneController che è la classe che permette il cambio di scena in ogni gioco; dunque, fornisce il codice base per effettuare il cambio di scena, attraverso il metodo SwitchScene, questo per ridurre il più possibile le righe di codice utilizzate e sfruttare le potenzialità della OOP.

menu

La logica del menu principale è racchiusa all'interno del package menu.

La classe GameAccess si occupa della scelta dell'utente tra un nuovo accesso o un'utenza già creata. In base alla sua scelta, l'utente comunicherà con due classi differenti, LogInController, e SignInController.

La prima si occuperà dell'accesso ai dati dell'account di un'utenza già creata, che vengono criptati tramite una password scelta dall'utente e salvati all'interno di cartelle in locale.

La seconda invece permetterà all'utente di creare un nuovo account.

In seguito, la classe GameChoiceMenuController, si occupa della scelta del gioco una volta effettuato l'accesso al menu principale; quindi, estende la classe SceneController necessaria ad effettuare il cambio di scena.

ModeMenuController è la classe che permette di scegliere il tipo di modalità di gioco desiderata, tra le opzioni ricordiamo esserci manuale e automatica.

Lo startController è il controller che si occupa di effettuare il cambio di scena per il LogIn o il SignIn Controller.

gameChart

La logica del package gameChart si occupa della visualizzazione dei grafici relativi alle statistiche di gioco.

All'interno di questo package, la classe principale è GameChartController, che si occupa della gestione dell'interfaccia grafica dedicata all'analisi dei risultati ottenuti nelle varie partite giocate.

Attraverso l'utilizzo di grafici a barre o a torta, l'utente può visualizzare in modo chiaro ed intuitivo i dati aggregati o dettagliati sulle sue performance. I dati vengono recuperati da file locali salvati durante l'esperienza di gioco, e successivamente elaborati per fornire un resoconto visivo.

Il controller comunica con eventuali classi di servizio o utility per il recupero e la formattazione dei dati. Inoltre, estende SceneController per permettere il ritorno al menu principale o il passaggio ad altre scene informative.

Questo modulo ha lo scopo di fornire un feedback all'utente sull'andamento delle sue partite e sulle scelte di gioco più frequenti, rendendo così l'esperienza più immersiva e personalizzata.

fileManager

Il package fileManager si occupa della gestione dei file locali e della sicurezza dei dati salvati, in particolare delle credenziali utente.

Al suo interno troviamo due classi principali: UserFileManager e PswEncrypting.

La classe UserFileManager gestisce tutte le operazioni di lettura, scrittura, aggiornamento e cancellazione di file utente. È responsabile della creazione delle directory necessarie sul desktop dell'utente, dove vengono salvati i dati degli account e le statistiche della modalità automatica. Offre metodi per salvare testo su file, leggere contenuti, aggiungere testo in append, aggiornare completamente un file, e cancellare intere cartelle associate a un utente. I percorsi sono gestiti dinamicamente in base all'ambiente dell'utente, garantendo compatibilità e flessibilità.

La classe PswEncrypting, invece, è dedicata alla sicurezza delle password, attraverso l'uso del cifrario di Cesare. Offre metodi per cifrare e decifrare le password, che vengono poi salvate su file tramite UserFileManager. Questa classe incapsula quindi una logica semplice ma efficace di protezione, utile in contesti didattici o di prototipazione.

Nel complesso, il package fileManager fornisce l'infrastruttura necessaria per una gestione sicura e ordinata dei dati dell'utente, mantenendo una buona separazione tra la logica di cifratura e quella di accesso ai file.

autoMode

La classe AutoMode implementa un sistema di simulazione automatizzata per testare strategie di gioco d'azzardo in condizioni controllate. Avviata con parametri configurabili (capitale iniziale, limite scommesse, durata round/totale), gestisce in parallelo:

- Selezione Dinamica Giochi: Scelta casuale da una lista predefinita (enabledGames) tra dadi, roulette, slot e altri, ciascuno con logiche di risultato integrate (es. DiceGame.diceResult(bet) restituisce guadagno/perdita).
- Gestione Scommesse Adaptive: Calcola puntate tra SMALLEST_BET (10) e maxBet, adattandosi al capitale corrente. Utilizza AtomicInteger per aggiornamenti thread-safe del saldo, evitando race condition.

Monitoraggio Real-Time:

- Grafico Interattivo: LineChart aggiornato ogni 100ms tramite buffer ConcurrentLinkedQueue, mantenendo solo ultimi 100 punti per efficienza. Assi ridimensionati dinamicamente (adjustChartViewport()) per seguire l'andamento.
- Etichetta di Stato: Mostra il gioco corrente con aggiornamento asincrono (Platform.runLater()).
- Logiche di Arresto: Termina simulazione per bancarotta (currentMoney < SMALLEST_BET) o timeout (totalTimeMillis), mostrando alert contestuali. In caso di chiusura manuale, interrompe thread e libera risorse (stopSimulation()).
- La raccolta dei dati permette di tener traccia delle seguenti informazioni: round giocati/vinti/persi; scommessa massima e guadagno singolo massimo; distribuzione utilizzo giochi (gameCounts)

Report Finale: Al termine, genera un riepilogo testuale con pulsante di salvataggio su file (nome univoco basato su timestamp). Il formato include dettagli finanziari e statistiche d'uso, salvati in UserFileManager.getAutoModePath().

Ottimizzazioni Tecniche:

UI Reattiva: Aggiornamenti grafici schedulati (ScheduledExecutorService) evitano blocchi del thread principale.

Memoria Controllata: Rimozione punti grafico obsoleti previene consumo eccessivo.

Cross-Game Compatibility: Interfaccia uniforme per risultati tramite switch-case, facilmente estendibile a nuovi giochi.

Il sistema funge sia da strumento analitico per bilanciamento economici, sia da ambiente di stress-test per verificare sostenibilità strategie di scommessa a lungo termine, replicando condizioni reali con variabilità controllata.

Descrizione dei moltiplicatori dei giochi

Dice Game

```
/// Calculate totals
int playerTotal = Arrays.stream(playerDiceResults).sum();
int botTotal = Arrays.stream(botDiceResults).sum();

// Determine outcome
int margin = playerTotal - botTotal;
int result = 0;

player.incrementGamesPlayed();
this.startMoney = player.getMoney(); //for final message calculation
```

```
int[] playerDiceResults = new int[currentMode];
int[] botDiceResults = new int[currentMode];

for (int i = 0; i < currentMode; i++) {
    playerDiceResults[i] = (int) (Math.random() * 6) + 1; // Player's roll
    botDiceResults[i] = bot.rollDice(); // Bot's roll
}
```

Il sistema determina gli esiti finanziari delle partite basandosi sul margine di vittoria/sconfitta tra giocatore e bot, utilizzando una struttura a livelli predefiniti combinata con feedback visivi dinamici. Il processo inizia calcolando i totali dei dadi: playerTotal (somma dei valori dei dadi del giocatore) e botTotal (somma dei valori dei dadi del bot), da cui si ricava il margin (differenza tra i due totali).

Per le vittorie (margin > 0), il sistema applica moltiplicatori progressivi:

- Margine ≥ 10: moltiplicatore 4x (es. puntata 10 → +40), con notifica "+4X SUPER WIN!"
- Margine 5-9: moltiplicatore 0.75x (es. 10 → +7.5, arrotondato a 7), notifica "+1X"
- Margine 3-4: moltiplicatore 0.5x (es. 10 → +5), notifica "+0.5X"
- Margine 1-2: moltiplicatore 0.25x (es. 10 → +2.5 → 2), notifica "+0.25X".

Nelle sconfitte (margin < 0), vengono applicate penalità proporzionali:

- Margine ≤ -5: perdita totale della puntata (1x, es. 10 → -10), notifica "-1X"
- Margine -3 a -4: perdita del 75% (es. 10 → -7.5 → -7), notifica "-0.75X"
- Margine -1 a -2: perdita del 50% (es. 10 → -5), notifica "-0.5X".

```
// WIN CASE
if (margin > 0) {
    if (margin >= 10){
        result = (int)(betAmount * 4); // jackpot profit
        showMultiplier("+4X SUPER WIN!");
    }
    else if (margin >= 5) {
        result = (int)(betAmount * 0.75); // modest profit
        showMultiplier("+1X");
    } else if (margin >= 3) {
        result = (int)(betAmount * 0.5); // small profit
        showMultiplier("+0.5X");
    } else {
        result = (int)(betAmount * 0.25); // tiny reward
        showMultiplier("+0.25X");
    }

    player.addMoney(result);
    controller.showMoneyAnimation(result, true);
    player.incrementGamesWon();
    bot.incrementGamesLost();

// LOSS CASE
} else if (margin < 0) {
    int loss;

    if (margin <= -5) {
        loss = (int)(betAmount); // full bet lost
        showMultiplier("-1X");
    } else if (margin <= -3) {
        loss = (int)(betAmount * 0.75); // lose 75% of bet
        showMultiplier("-0.75X");
    } else {
        loss = (int)(betAmount * 0.5); // lose 50% of bet
        showMultiplier("-0.5X");
    }

    player.subtractMoney(loss);
    controller.showMoneyAnimation(loss, false);
    player.incrementGamesLost();
    bot.incrementGamesWon();

// TIE CASE
} else {
    showMultiplier("TIE!");
}
```

Il feedback visivo è gestito dal metodo showMultiplier(), che mostra un testo animato (es. "+4X") con un FadeTransition di 2 secondi: l'opacità parte al 100% (setFromValue(1.0)) e raggiunge lo 0% (setToValue(0.0)), creando un effetto di dissolvenza. Le modifiche monetarie (addMoney()/subtractMoney()) utilizzano il casting a intero per evitare valori frazionari, con animazioni dedicate per guadagni (verdi) e perdite (rosse).

I dati statistici vengono aggiornati in tempo reale: incremento di partite giocate/vinte/perse per giocatore e bot, e aggiornamento del grafico storico (gameChart.addData()). Le soglie dei margini (10/5/3/-3/-5) bilanciano rischio e ricompensa, premiando vittorie decisive ma limitando perdite eccessive. L'uso di moltiplicatori asimmetrici (max +4x vs max -1x) incentiva scommesse aggressive mantenendo sostenibilità economica. Il sistema è progettato per massimizzare l'engagement attraverso feedback immediati e una curva di difficoltà calibrata.

Guess Number

Il sistema di moltiplicatore adottato nel gioco è stato progettato per adattarsi dinamicamente al comportamento del giocatore, penalizzando i tentativi ripetuti e premiando la rapidità nella scelta del numero corretto. Questo valore viene aggiornato attraverso il metodo updateMultiplier(), che calcola un coefficiente numerico destinato a influenzare l'importo della vincita o della perdita. Tale coefficiente varia in modo continuo e si adatta al numero di tentativi effettuati rispetto al totale massimo consentito, che a sua volta dipende dall'intervallo numerico selezionato.

Adattamento all'intervallo di gioco

Il numero totale di tentativi (totalAttempts) è calcolato in base alla lunghezza dell'intervallo scelto (ad esempio da 1 a 15, oppure da 1 a 30). All'aumentare dell'intervallo, aumenta anche la difficoltà e, di conseguenza, il numero di tentativi disponibili. Il sistema del moltiplicatore tiene conto di questo aspetto per garantire un bilanciamento coerente del gameplay: il valore del moltiplicatore diminuisce in funzione del rapporto tra tentativi effettuati e tentativi totali, non in base al valore assoluto.

Struttura del calcolo

Il calcolo è suddiviso in due fasi principali:

1. **Fase Lineare (≤ 80% dei tentativi)**
   Quando il numero di tentativi effettuati è inferiore o uguale all'80% del totale disponibile, il moltiplicatore decresce in maniera lineare secondo la formula:

$$multiplier = baseMultiplier \times (1 - progress0.8) \text{multiplier} = \text{baseMultiplier} \times \left(1 - \frac{\text{progress}}{0.8}\right)$$multiplier=baseMultiplier×(1−0.8progress)

   Dove progress rappresenta la frazione dei tentativi effettuati. Questo garantisce che il giocatore parta con un moltiplicatore massimo (baseMultiplier) e lo veda decrescere progressivamente con ogni errore, fino a raggiungere 0 all'80% dei tentativi.

2. **Fase Esponenziale (> 80% dei tentativi)**
   Oltre la soglia dell’80%, il moltiplicatore diventa negativo, segnalando una penalità severa. In questa fase, il decremento è regolato da una funzione esponenziale:

$$multiplier = -1 \times (1 - e^{-5 \cdot negativeProgress})$$multiplier=−1×(1−e−5·negativeProgress)

Dove negativeProgress è la frazione dei tentativi oltre l’80%. L’uso dell’esponenziale produce una discesa rapida verso -1, penalizzando drasticamente chi raggiunge il limite dei tentativi senza successo.

```java
public double updateMultiplier() {
    attempts++;
    double progress = (double) attempts / totalAttempts;

    if(progress <= 0.8) {
        multiplier = baseMultiplier * (1 - (progress/0.8));
    } else {
        double negativeProgress = (progress - 0.8)/0.2;
        multiplier = -1 * (1 - Math.exp(-5 * negativeProgress));
    }

    multiplier = Math.round(multiplier * 100.0) / 100.0;
    return multiplier;
}
```

Roulette

```java
// Numeri
if (buttonText.equals(nUscito.toString())) {
    Main.player.addMoney(chipValue * 35); // Moltiplicatore x35 per i numeri
    animateButton(button);
}
// Colori
if (buttonText.equalsIgnoreCase("RED") && redNumbers.contains(nUscito)) {
    Main.player.addMoney(chipValue * 2); // Moltiplicatore x2 per il rosso
    animateButton(button);
} else if (buttonText.equalsIgnoreCase("BLACK") && blackNumbers.contains(nUscito)) {
    Main.player.addMoney(chipValue * 2); // Moltiplicatore x2 per il nero
    animateButton(button);
}
// Pari/Dispari
if (buttonText.equalsIgnoreCase("ODD") && nUscito % 2 == 1) {
    Main.player.addMoney(chipValue * 2); // Moltiplicatore x2 per il dispari
    animateButton(button);
} else if (buttonText.equalsIgnoreCase("EVEN") && nUscito % 2 == 0) {
    Main.player.addMoney(chipValue * 2); // Moltiplicatore x2 per il pari
    animateButton(button);
}
// Dozzine
if (buttonText.equalsIgnoreCase("1st 12") && nUscito >= 1 && nUscito <= 12) {
    Main.player.addMoney(chipValue * 3); // Moltiplicatore x3 per la prima dozzina
    animateButton(button);
} else if (buttonText.equalsIgnoreCase("2nd 12") && nUscito >= 13 && nUscito <= 24) {
    Main.player.addMoney(chipValue * 3); // Moltiplicatore x3 per la seconda dozzina
    animateButton(button); // Animazione per il bottone vincente
} else if (buttonText.equalsIgnoreCase("3rd 12") && nUscito >= 25 && nUscito <= 36) {
    Main.player.addMoney(chipValue * 3); // Moltiplicatore x3 per la terza dozzina
    animateButton(button);
}
// Colonne (righe)
if (currentRow.contains(nUscito) && !rowGotRight) {
    System.out.println("Row got Right");
    rowGotRight = true;
    Main.player.addMoney(chipValue * 3); // Moltiplicatore x3 per la riga
    animateButton(button);
}
// 1-18 / 19-36
if (buttonText.equalsIgnoreCase("1 to 18") && nUscito >= 1 && nUscito <= 18) {
    Main.player.addMoney(chipValue * 2); // Moltiplicatore x2 per 1-18
    animateButton(button);
```

Per il sistema di moltiplicatori della roulette sono stati utilizzati gli standard della roulette reale; ad esempio, se si indovina il colore, si viene ricompensati con il doppio della puntata. Per determinare quali bottoni controllare, è stato creato un algoritmo che scansiona tutti i bottoni presenti nella griglia e seleziona quelli che contengono un'immagine (la chip). Una volta individuati, l'algoritmo analizza il testo del bottone per identificare la puntata corrispondente e, confrontandola con il numero uscito, aggiunge denaro in caso di vincita. In caso di vittoria, il bottone lampeggia di giallo per indicare all’utente quale puntata ha avuto successo

Slot Machines

```
private double getMultiplier(int count, int maxSlot) {
    if (maxSlot == 12) { // 4x3
        return count == 3 ? 2.0 : count == 4 ? 3.0 : 0.0;
    }
    else if (maxSlot == 9) { // 3x3
        return count == 2 ? 1.2 : count == 3 ? 2.0 : 0.0;
    }
    else { // 3x1
        return count == 2 ? 1.2 : 0.0; // Rimosso il caso 3 simboli
    }
}
```

getMultiplier (count,maxSlot): i parametri formali prendono i valori del conto di quanti simboli uguali ci sono per riga e il massimo delle slot(es. 3X1 = 3, ecc.). In base al massimo delle slot, usa la variabile count per vedere quali moltiplicatori usare in base ai simboli uguali: 2 simboli: 1.2x; 3 simboli: 2x; 4 simboli: 3x

getJackpotMultiplier(maxSlot): in base alle slot, se tutte le righe hanno 3(o 4 se viene scelto 3x4) simboli uguali allora fa jackpot, questo metodo viene invocato e dà come moltiplicatori: 3x1: 5x; 3x3: 7.5x; 3x4: 10x

```
private double getJackpotMultiplier(int maxSlot) {
    switch (maxSlot) {
        case 3:
            return 5;
        case 9:
            return 7.5;
        case 12:
            return 10;
        default:
            return 5;
    }
}
```

checkResults(maxSlot): minWin determina con un if con condizione che maxslot sia 12 che il minimo per vincere sia 3,altrimenti è 2.Essendo che è una matrice firstSymbol è in coordinate 0 0.
Con un for controlla se si ha vinto usando l'ArrayList winCounts che all'interno ha quante volte si ha vinto per riga, in base a se si ha raggiunto la soglia minima per vincere, la variabile hasWin cambia, se ha tutte le righe riempite di simboli uguali allora fullwin torna true e ritorna il valore "f" per indicare un jackpot, altrimenti ritorna hasWin che può essere "none" in caso sia falso o "multi-win" in caso sia vero

```
private String checkResults(int maxSlot) {
    int minWin = (maxSlot == 12) ? 3 : 2;
    boolean hasWin = false;
    boolean fullWin = true;
    int firstSymbol = rows[0][0];

    for(int i = 0; i < rows.length; i++) {
        if(winCounts.get(i) >= minWin) hasWin = true;
        for(int j = 0; j < rows[i].length; j++) { // Usa la lunghezza della riga
            if(rows[i][j] != firstSymbol) fullWin = false;
        }
        // Verifica jackpot

    }

    System.out.println("\nRISULTATO FINALE:");
    System.out.println("Vincite valide: " + hasWin);
    System.out.println("Jackpot: " + fullWin);

    return fullWin ? "f" : hasWin ? "multi-win" : "none";
}
```

```java
public void endGame() {

      Main.player.incrementGamesPlayed();

        // Calcola la differenza netta
        int differenzaNetta = finishMoney - startMoney;
        slot.updateMoneyAnimation(
                startMoney,          // Soldi iniziali pre-spin
                finishMoney,         // Soldi finali post-spin
                lblMLCurrency        // Label da aggiornare
            );

            // Aggiorna il modello DOPO l'animazione
            new Timeline(new KeyFrame(
                Duration.millis(700),
                e -> {
                    Main.player.setMoney(finishMoney);
                    if(gameChart != null) {
                        gameChart.addData(restartCount + 1, finishMoney);
                    }



                    if(differenzaNetta > 0) {
                        playerWon();
                        System.out.println("PlayerWon");
                        isGameWon = true;
                        this.gameWonCounter++;
                        Main.player.incrementGamesWon();


                    } else if(differenzaNetta < 0) {
                        playerLost();
                        System.out.println("PlayerLost");
                        isGameWon = false;
                        this.gameLossCounter++;
                        Main.player.incrementGamesLost();
                    } else {
                        playerDraw();
                        isGameWon = null;
                        System.out.println("Equal");

                    }
                }
            )).play();
```

```java
private void victoryState(Label lblWinLose, int maxSlot) {
    StringBuilder message = new StringBuilder();
    double totalMultiplier = 0;
    final int finalTotalMultiplier = 0;
    MoneyManagerSlot mms = new MoneyManagerSlot();

    if(winDescriptor.equals("f")) {
        message.append("JACKPOT! (")
               .append(getJackpotMultiplier(maxSlot))
               .append("x)");
    } else if(winCounts != null && !winCounts.isEmpty()) {
        List<Integer> orderedWinCounts = new ArrayList<>();

        // Organizza le righe nell'ordine corretto
        switch(maxSlot) {
            case 3: // 3x1 - singola riga centrale
                orderedWinCounts.addAll(winCounts);
                break;

            case 9: // 3x3 - righe: UP, MID, DOWN
            case 12: // 3x4 - righe: UP, MID, DOWN
                if(winCounts.size() >= 3) {
                    orderedWinCounts.add(winCounts.get(0)); // UP (prima riga)
                    orderedWinCounts.add(winCounts.get(1)); // MID (seconda riga)
                    orderedWinCounts.add(winCounts.get(2)); // DOWN (terza riga)
                }
                break;
        }

        String[] labels = maxSlot == 3 ?
            new String[]{"Mid"} : // Solo Mid per 3x1
            new String[]{"Up", "Mid", "Down"}; // Etichette standard

        for(int i = 0; i < orderedWinCounts.size(); i++) {
            int count = orderedWinCounts.get(i);
            if(count < (maxSlot == 12 ? 3 : 2)) continue; // Filtra vincite non valide

            double multiplier = mms.calculateMultiplier(count, maxSlot);
            totalMultiplier += multiplier;



            message.append(labels[i])
                   .append(": ")
                   .append(count)
                   .append("x (")
                   .append(multiplier)
                   .append("x)\n");
        }

        if(totalMultiplier > 0) {
            message.append("TOT: ").append(totalMultiplier).append("x");
        } else {
            message.setLength(0);
            message.append("Nessuna vincita valida");
        }
    } else {
        message.append("Nessuna vincita");
    }
```

Alla fine del gioco viene invocato endGame(): questo metodo aumenta il numero di partite giocate dal player. Poi calcola la differenzaNetta tra i soldi iniziali e i soldi finali, con vari if viene determinato se la partita è stata vinta, persa o finita in pareggio. In base al risultato aumentano le partite vinte o perse.

VictoryState(lblWinLose,maxSlot): è utilizzata una variabile di tipo StringBuilder per far si che il messaggio si componesse passo per passo: se ha un jackpot(winDescriptor.equals(“f”)) allora usa append per comporre la frase opportuna al caso. Poi controlla se l’ArrayList winCounts è vuoto, se non lo è allora si crea una lista che in base a quante slot si hanno conta le righe per far si che localizzi in quale di essa si abbia vinto(case 9 e 12 hanno lo stesso modo di utilizzo). Poi si creano le etichette “Mid” “Up” e “Down” che verranno usate per far capire dove si trovano le righe in cui si vince. In base a quante volte si vince, si controlla se sono valide e poi viene dato un moltiplicatore in base a quante volte si ha vinto. Poi si compone il messaggio finale che varia in base a l’esito delle vittorie.

Coin Flip

```
d.setOnFinished(event -> {
    boolean monetaValue = new Random().nextBoolean();
    risultatiMonete.add(monetaValue);
    if (monetaValue) {
        coin.setMaterial(new PhongMaterial(Color.GREEN));
    } else {
        coin.setMaterial(new PhongMaterial(Color.RED));
    }
    onFinished.run();
});

d.play();
```

```
List<Cylinder> coins = Arrays.asList(coin1, coin2, coin3, coin4);
final int totalCoins = coins.size();
final int[] finished = {0};

for (Cylinder coin : coins) {
    animateCoin(coin, () -> {
        finished[0]++;
        if (finished[0] == totalCoins) {
            // Tutte le animazioni sono terminate
            corrette = risultatiMonete.stream().filter(r -> r == scelta).count();
```

Il moltiplicatore viene calcolato a partire dall'ArrayList risultatiMonete, che contiene gli esiti dei lanci delle monete (testa o croce). La variabile corrette rappresenta il numero di monete che corrispondono alla scelta del giocatore.

Il metodo animateCoin() effettua i lanci delle monete e assegna a ciascuna il valore testa o croce. Poiché il numero di monete può variare da 1 a 6, viene utilizzato un ciclo for che richiama il metodo tante volte quanti sono gli elementi nell'array coins.

Per calcolare `corrette`, si usa:

- .stream() crea un flusso (stream) di elementi a partire dalla lista.
- .filter() applica una condizione: nel nostro caso, tiene solo i valori che sono uguali alla scelta del giocatore (ad esempio, solo le teste se ha scelto testa).
- .count() restituisce il numero di elementi che hanno superato il filtro, cioè il numero di monete vincenti.

Il moltiplicatore (molt) si calcola confrontando la previsione (n, numero di monete vincenti attese, selezionato con una ChoiceBox) con il risultato reale (ris, ovvero corrette). Se la previsione è esatta (n == ris), si vince e il moltiplicatore è 0. Altrimenti si perde, e molt sarà pari alla differenza assoluta tra previsione e risultato.

```
public int getCorretteInt() {
    return (int) corrette;
}

void guadagno() {
    Integer n = scommessa.getValue();
    int ris = getCorretteInt();

    molt = Math.abs(n - ris);
}
```

Math.abs() serve per trasformare la differenza in valore positivo, così da trattare allo stesso modo errori in eccesso o in difetto.