Relazione: 7 e mezzo

Esame Programmazione III

Prof. Ciaramella Prof. Di Nardo

Riccardo Aiello Matricola: 0124002251
Crescenzo Esposito Matricola: 0124002375

Indice

1 Introduzione

1.1 Scopo

Il progetto ha l'obiettivo di rappresentare una partita del gioco del 7 e mezzo attraverso un'interfaccia grafica user-friendly. L'applicazione gestisce n giocatori, uno dei quali è controllato dal computer, e applica fedelmente le regole del gioco tradizionale del sette e mezzo, incluse le dinamiche di sballo e distribuzione delle carte.

1.2 Riferimenti

- Regole del gioco del 7 e mezzo: `https://it.wikipedia.org/wiki/Sette_e_mezzo`

2 Descrizione generale del sistema

2.1 Obiettivi del sistema

Il sistema ha come obiettivo simulare una partita del gioco del 7 e mezzo.
Deve gestire il punteggio dei giocatori, determinare i vincitori e aggiornare in tempo reale i dati del mazziere e dei giocatori (computer incluso). Al termine di ogni partita, il sistema permette la visualizzazione della classifica e il salvataggio delle partite, che possono essere riprese successivamente, ripartendo dall'ultimo turno giocato.

2.2 Vincoli

- Utilizzo del linguaggio Java e JavaFX per l'interfaccia utente.
- Implementazione secondo i principi SOLID.
- Gestione delle eccezioni
- Utilizzo di pattern di progettazione.

3 Architettura del sistema

3.1 Panoramica dell'architettura

Il sistema è diviso principalmente in

- **model**: Gestione la logica del gioco.
- **view**: Gestione dell'interfaccia grafica.

4 Patterns utilizzati

Abbiamo scelto di utilizzare pattern comportamentali per gestire la comunicazione e l'interazione tra gli oggetti del sistema. Ci è sembrata la scelta adatta data la natura interattiva che abbiamo dato al gioco.

4.1 Strategy

Il pattern **Strategy** è stato il primo ad essere implementato, con lo scopo di differenziare i ruoli dei giocatori che vengono quindi assegnati a tempo di esecuzione

Di seguito viene mostrata l'implementazione della strategia del computer come richiesto nella traccia.

```java
public class StrategiaComputer implements Strategia {
    @Override
    public boolean applicaStrategia(Mano mano) {
        boolean manovuota;
        if (mano.primaCarta() == null)
            manovuota = true;
        else
            manovuota = mano.primaCarta().getValore() < 4;
        return manovuota && mano.getValore() < 5;
    }

    @Override
    public int daiGettoni(Giocatore computer, int mediaPuntate) {
        int puntataFissa = 5;
        if(mediaPuntate<5) {
            computer.daiGettoni(puntataFissa);
            return puntataFissa;
        } else{
            computer.daiGettoni(mediaPuntate);
            return  mediaPuntate;
        }
    }
}
```

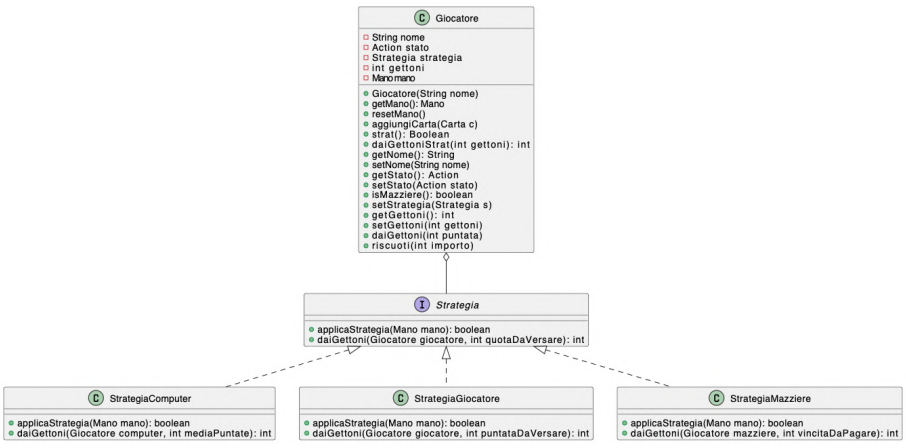

Figura 1: Strategy Pattern

4.2 Observer

Il pattern **Observer** ci ha permesso di separare la gestione dei suoni e dell'aggiornamento dell'interfaccia dalla logica di gioco. Questo ha reso il sistema flessibile e facilmente estensibile, rendendo agevole l'aggiunta di nuovi comportamentali senza stravolgere il codice già esistente.
Soggetto Osservato: La classe 'Partita' si occupa di comunicare agli observers le azioni dei giocatori come la pescata oppure quando un giocatore sballa.

Di seguito mostro la classe 'Sound' come dimostrazione della semplicità di estensione avuta usando il pattern observer per cui è bastato il riutilizzo del metodo update dell'interfaccia.

```
1  public class Sounds implements gameObserver {
2      @Override
3      public void update(Action action, String... message) {
4          // inizio del suono in base al messaggio inviato dall'observed
5          Platform.runLater(() -> {
6              String path = "src/main/resources/it/uniparthenope/sette_e_mezzo/sounds/";
7              try {
8                  switch (action) {
9                      case match:
10                         break;
11                     case busted:
12                         playSounds(path + "game-over.wav");
13                         break;
14                     case setteMezzo:
15                         playSounds(path + "applause.mp3");
16                         break;
17                     case pescato:
18                         break;
19                     case matta:
20                         playSounds(path + "success.wav");
21                         break;
22                 }
23             }   catch(URISyntaxException e){
24                     throw new RuntimeException(e);
25                 }
26         });
27     }
28
29     private void playSounds(String fileAudio) throws URISyntaxException {
30         AudioClip sound = new AudioClip(new File(fileAudio).toURI().toString());
31         sound.play();
32     }
33 }
```

Osservatori Concreti: Gli osservatori concreti sono in questo caso 'GameUI' e 'Sounds', implementano l'interfaccia 'gameObserver' e reagiscono in base al valore ricevuto modificando l'interfaccia o facendo partire i suoni.

Action: L'enum 'Action' definisce i tipi di eventi che possono essere notificati.

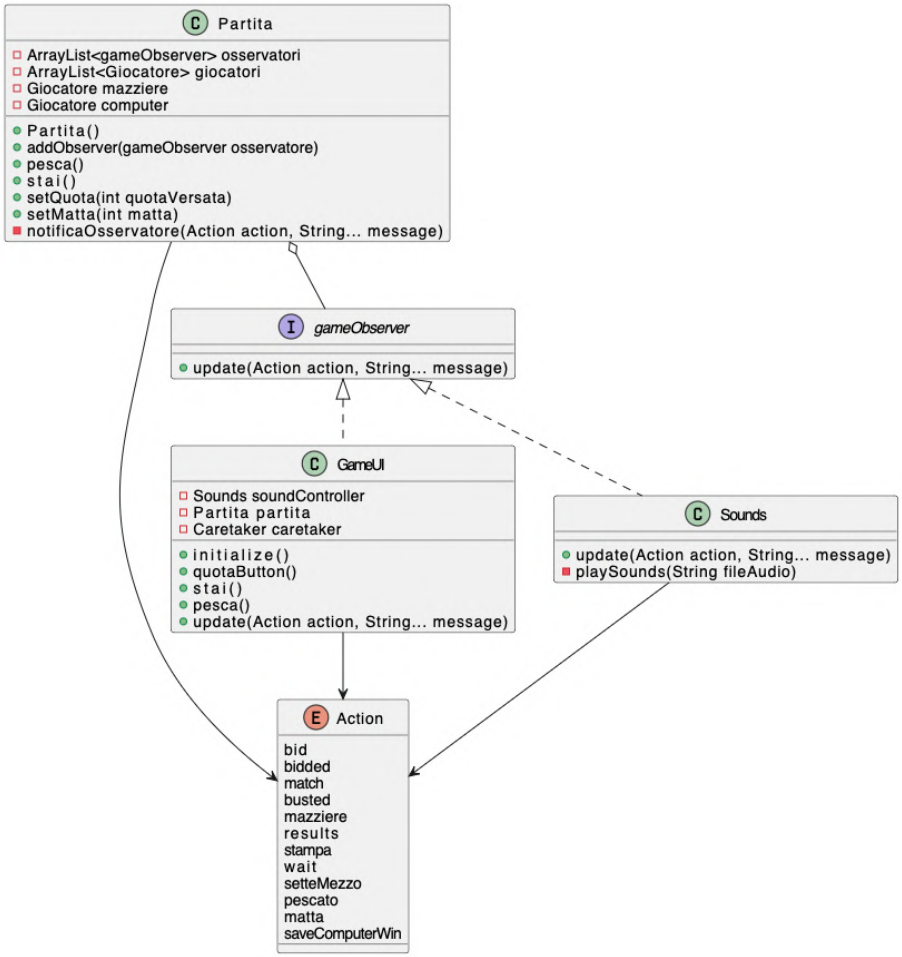

Figura 2: Observer Pattern

4.3 State

Il pattern **State** è stato implementato per gestire lo stato dinamico dei giocatori durante la partita. Questo ci ha permesso di aggiornare le interfacce grafiche in tempo reale a seconda dello stato corrente del giocatore, rendendo l'esperienza più intuitiva e interattiva.

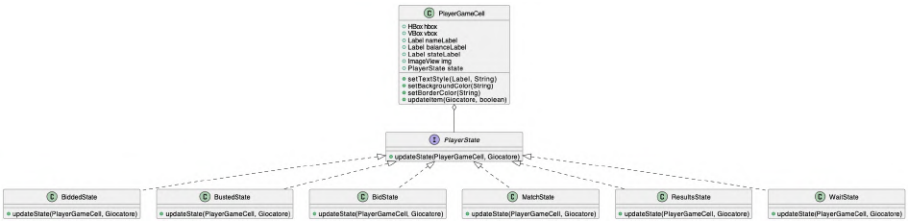

Figura 3: State Pattern

Nella classe PlayerGameCell è definita l'interfaccia PlayerState con cui poi saranno composti gli stati derivati.

Ecco di seguito il codice per modificare lo stato che riusa gli stessi enum già definiti per il pattern observer in modo da non complicare ulteriormente la logica del codice.

```java
state = switch (player.getStato()) {
    // cambiamento di stato in base al valore dell'enum ricevuto
    case bid -> new BidState();
    case match -> new MatchState();
    case bidded -> new BiddedState();
    case wait -> new WaitState();
    case busted -> new BustedState();
    case results -> new ResultsState();
    default -> null;
};
```

4.4 Memento

Abbiamo usato il pattern **Memento** per implementare funzionalità di **salvataggio e ripristino** dello stato del gioco. Ogni partita può essere memorizzata ssotto forma di uno stato salvato, che può essere ripristinato in qualsiasi momento. Questo garantisce la continuità del gioco, sia nel caso in cui si voglia giocare un altro turno, sia se si volesse riprendere una partita giocata tempo fa.

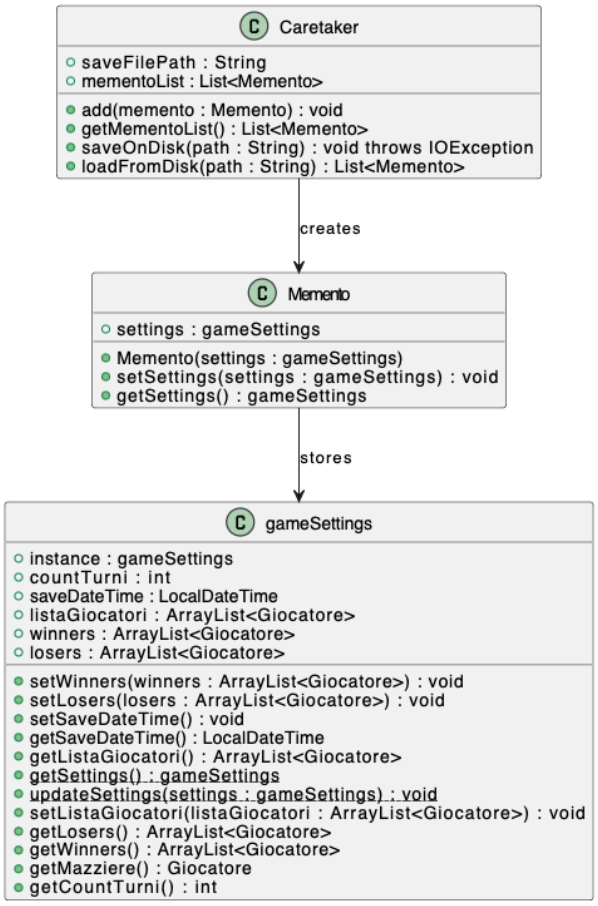

Figura 4: Memento Pattern

5 diagrammi UML

5.1 Diagrammi delle classi

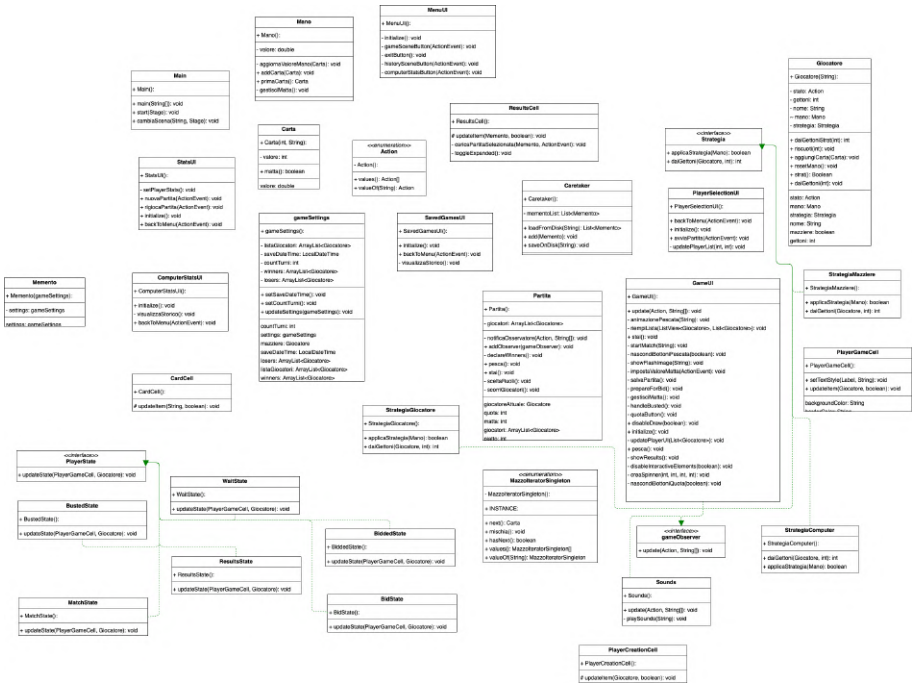

Figura 5: Diagramma delle classi: metodi e attributi

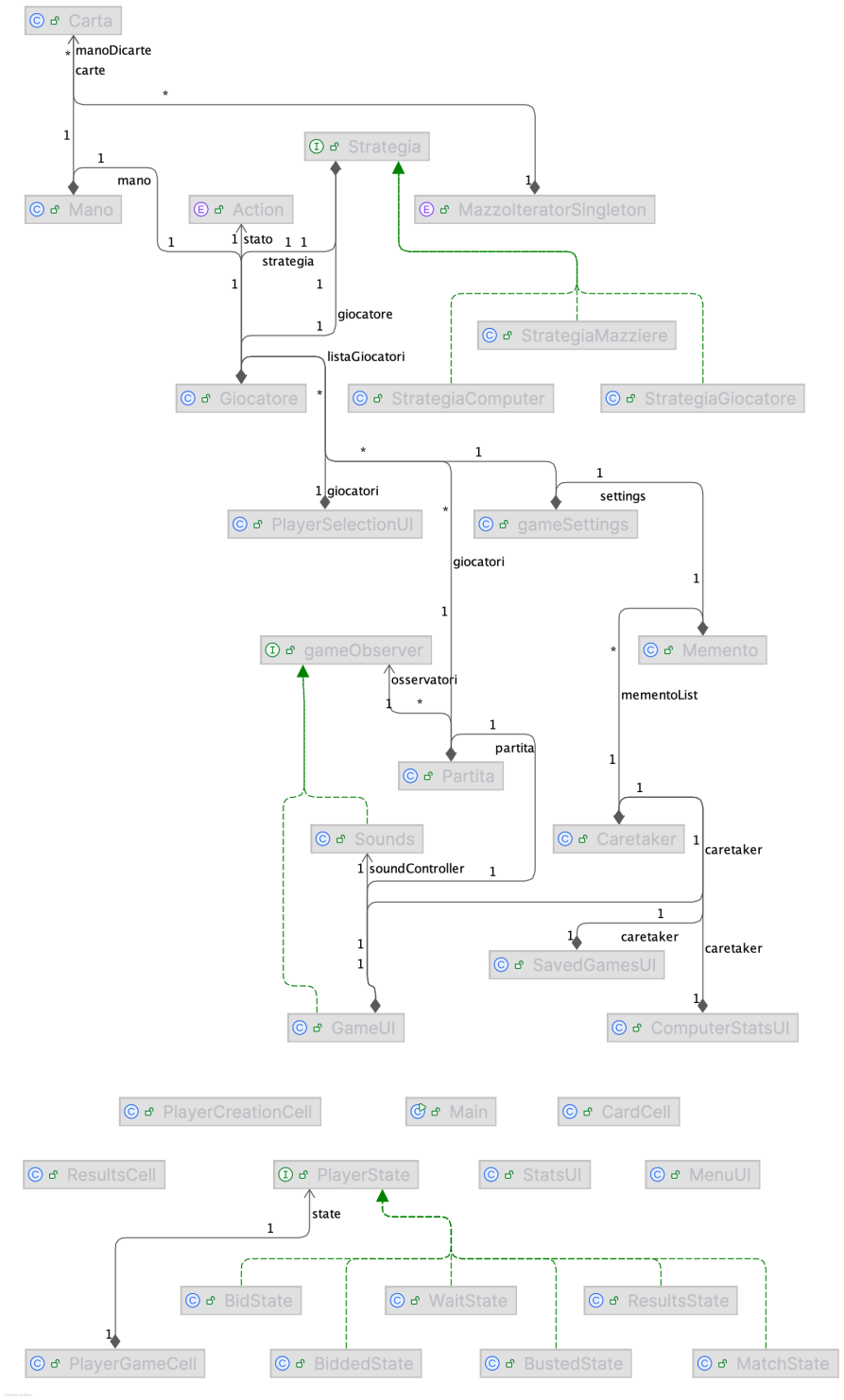

Figura 6: Diagramma delle classi: dipendenze

5.2 Diagramma di sequenza

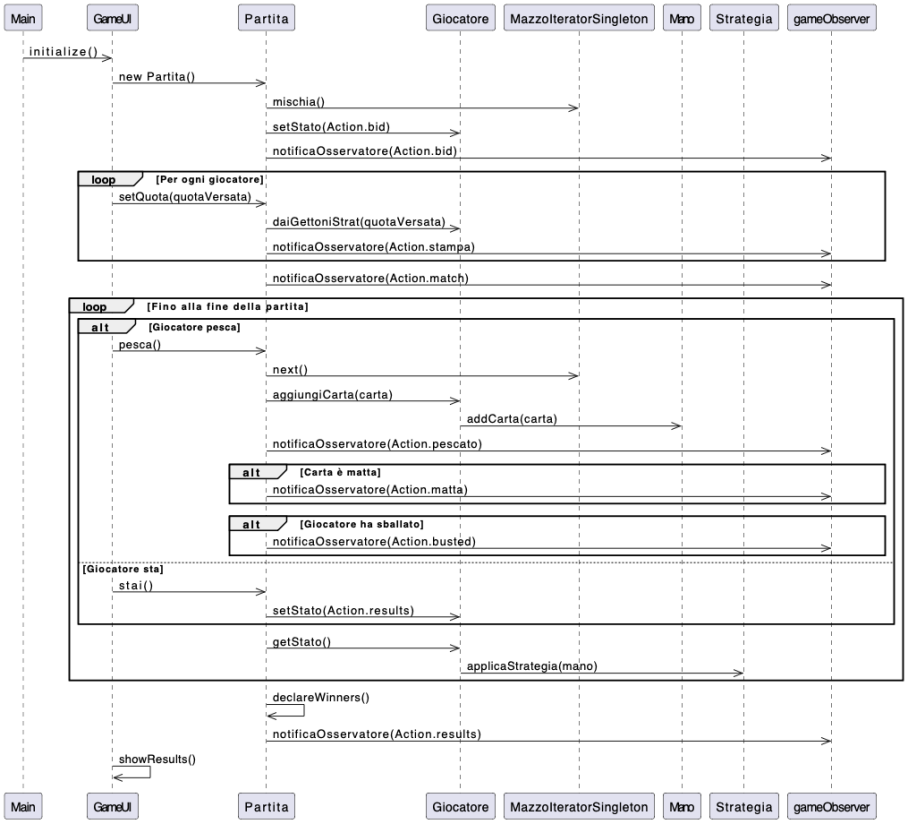

Figura 7: Diagramma di sequenza

6 Scenari di utilizzo

6.1 Avvio di una partita

1. L'utente avvia una nuova partita selezionando il numero di giocatori, può modificare il nome di ogni giocatore e il numero di gettoni iniziale.

2. I giocatori a turno versano le quote selezionando quanto puntare tramite uno spinner. Il computer punta in automatico, il mazziere è escluso.

3. Vengono distribuite le carte e inizia la prima sfida contro il mazziere.

4. Ogni giocatore può chiedere nuove carte o fermarsi.

5. La partita prosegue finché tutti i giocatori hanno completato il loro turno.

6. Alla fine della partita, si calcolano i risultati e vengono mostrati. Inoltre vengono salvati su file insieme allo stato finale della partita.

6.2 Rigiocare una partita

1. L'utente visualizza lo storico delle partite salvate.
2. Può visualizzare le statistiche dell'ultimo turno giocato (vincitori, sconfitti) e quanti gettoni restano ai vari giocatori.
3. L'utente può selezionare un'altra parita (se presente), o decidere di rigiocare la partita che sta visualizzando.

6.3 Visualizzare le vittorie del computer

1. L'utente visualizza lo storico dei turni vinti dal computer.
2. Può visualizzare le statistiche del turno giocato (vincitori, sconfitti) e quanti gettoni restano ai vari giocatori.
3. E' in oltre presente un counter che tiene conto di tutte le vittorie del computer.

7 Requisiti non funzionali

- **Scalabilità**: Il sistema deve essere in grado di gestire da 2 a 4 giocatori più il computer.
- **Sicurezza**: Il sistema deve gestire le eccezioni.
- **Performance**: Deve rispondere in tempo reale alle azioni dell'utente.
- **Manutenibilità**: Il codice è scritto seguendo i principi SOLID, facilitando la modifica e l'estensione del progetto.

8 Tecnologie utilizzate

- **Frameworks e librerie**: JavaFX per la parte grafica, e utilizzo di File I/O per il salvataggio su file.

9 UI

Realizzata con JavaFX, ecco elencate le varie schermate della partita.

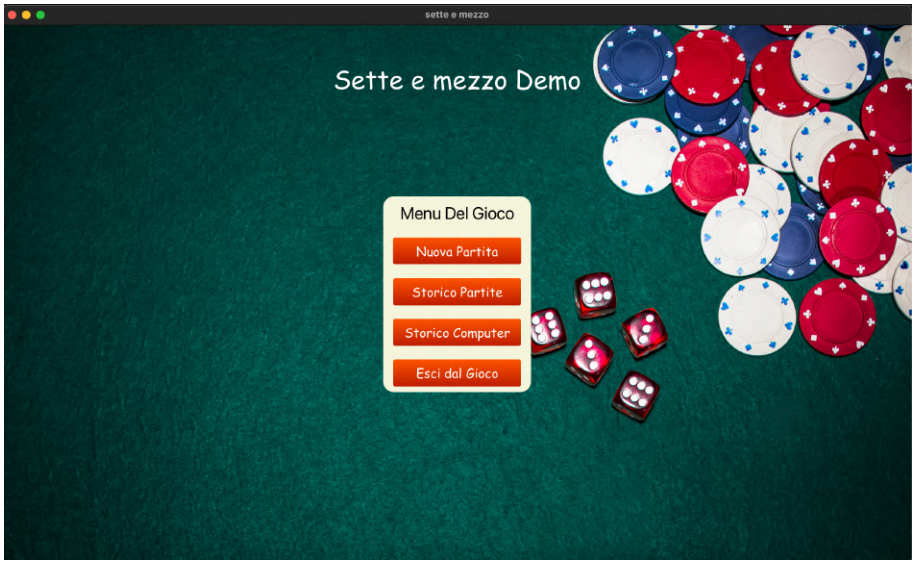

Figura 8: Schermata del menu

Figura 9: Schermata di selezione giocatori

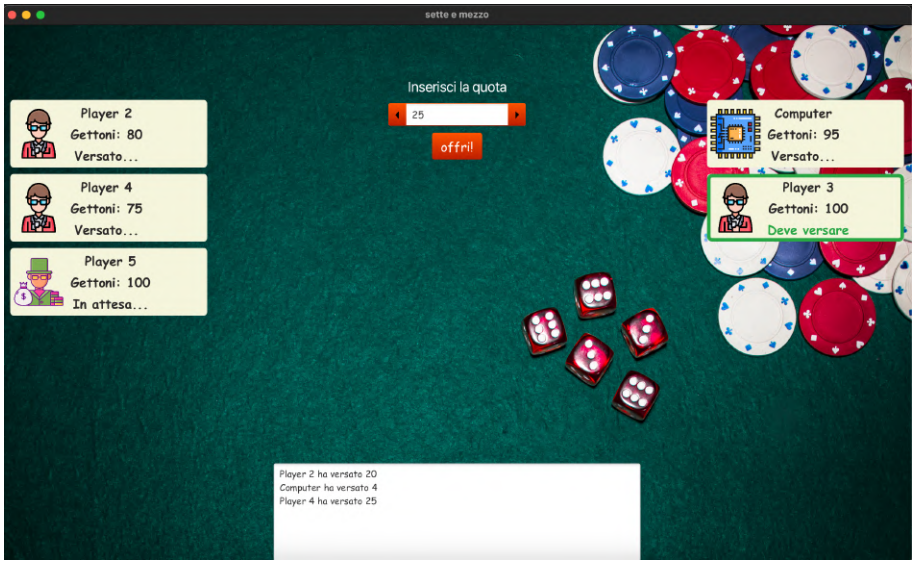

Figura 10: Schermata delle quote

Figura 11: Schermata della partita

Figura 12: Schermata dei risultati

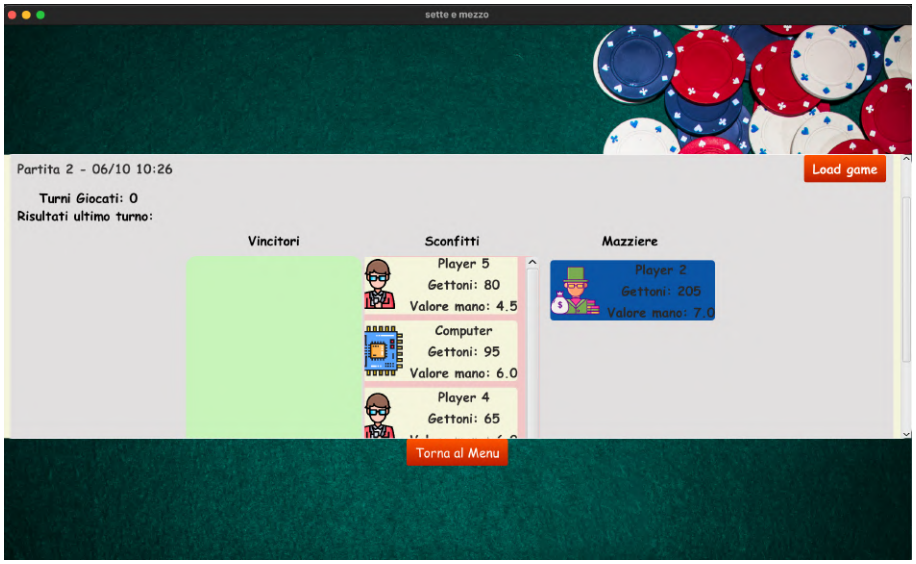

Figura 13: Schermata dei salvataggi