Documentazione Tecnica - Gioco del Tresette

Indice

Panoramica del Sistema

Il progetto **Tresette** è un'implementazione completa del gioco di carte tradizionale italiano, sviluppato in Java utilizzando l'interfaccia grafica Swing. Il sistema supporta sia la modalità a 2 giocatori che quella a 4 giocatori, con avversari controllati dall'intelligenza artificiale.

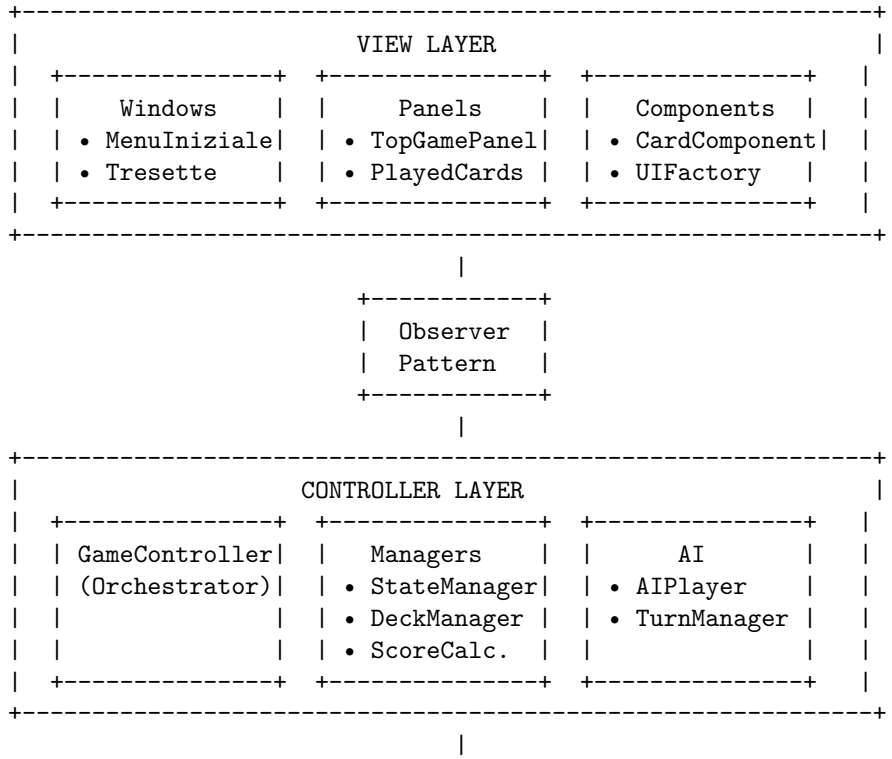
Caratteristiche Principali

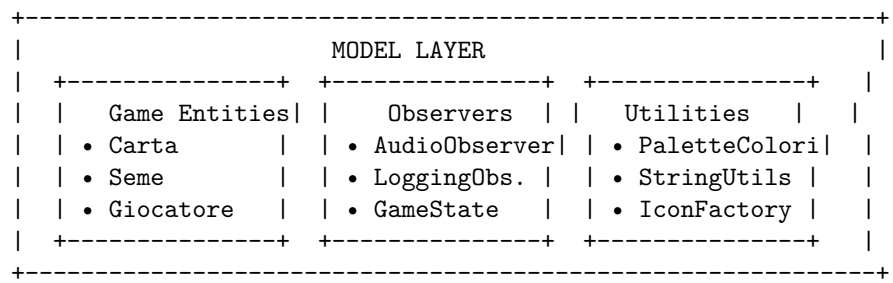
- **Modalità di Gioco**: 2 o 4 giocatori (1 umano + AI)
- **Interfaccia Grafica**: Swing con layout personalizzati
- **Sistema Audio**: Effetti sonori integrati
- **Gestione Statistiche**: Tracking delle performance del giocatore
- **Architettura Modulare**: Separazione clara tra Model, View e Controller

---

Architettura del Software

Il sistema segue il **pattern MVC (Model-View-Controller)** con l'aggiunta del **pattern Observer** per la comunicazione tra componenti:





Package e Organizzazione

`it.uniroma1.tresette.model`

Responsabilità: Entità di dominio e logica di business core - **Carta.java**: Rappresentazione di una carta da gioco - **Seme.java**: Enum per i semi delle carte (COPPE, DENARI, SPADE, BASTONI) - **Giocatore.java**: Modello del giocatore con mano e statistiche - **StatisticheGiocatore.java**: Gestione delle statistiche persistenti

`it.uniroma1.tresette.model.observer`

Responsabilità: Implementazione del pattern Observer - **GameStateObserver.java**: Interfaccia per gli observer - **GameStateObservable.java**: Classe osservabile per eventi di gioco - **GameState.java**: Enum degli stati del gioco - **AudioObserver.java**: Observer specifico per l'audio - **LoggingObserver.java**: Observer per il logging degli eventi

`it.uniroma1.tresette.controller`

Responsabilità: Logica di controllo e coordinamento - **GameController.java**: Controller principale che orchestra il gioco - **GameStateManager.java**: Gestione centralizzata dello stato - **DeckManager.java**: Gestione del mazzo e distribuzione carte - **ScoreCalculator.java**: Calcolo dei punteggi e valutazione mani - **TurnManager.java**: Gestione dei turni e pause - **AIPlayer.java**: Intelligenza artificiale per i giocatori automatici - **GameView.java**: Interfaccia per le operazioni sulla view

`it.uniroma1.tresette.view`

Responsabilità: Interfaccia utente e presentazione

Sottopacchetti:

- **windows/**: Finestre principali
  - `MenuIniziale.java`: Menu principale del gioco
  - `Tresette.java`: Finestra di gioco principale
- **panels/**: Pannelli componenti dell'UI
  - `TopGamePanel.java`: Pannello superiore con punteggi
  - `PlayedCardsPanel.java`: Pannello delle carte giocate
  - `PlayerCardsPanel.java`: Pannello delle carte del giocatore
  - `SideControlPanel.java`: Pannello controlli laterali
- **components/**: Componenti UI riutilizzabili
  - `CardComponent.java`: Componente grafico per le carte
  - `UIComponentFactory.java`: Factory per componenti UI
- **dialogs/**: Finestre di dialogo
  - `DialogManager.java`: Gestione centralizzata dei dialoghi
- **icons/**: Gestione risorse grafiche
  - `IconFactory.java`: Factory per le icone
  - `CardImageLoader.java`: Caricamento ottimizzato delle immagini
- **layout/**: Gestione layout
  - `GameLayoutManager.java`: Manager per il layout di gioco

- **sound/**: Sistema audio
  - `SoundManager.java`: Gestione effetti sonori
- **utils/**: Utilità per la view
  - `PaletteColori.java`: Definizione palette colori

`it.uniroma1.tresette.util`

Responsabilità: Utilità generali - **StringUtils.java**: Utility per manipolazione stringhe

---

Dettaglio delle Classi

Classi del Model

`Carta.java`

```
public class Carta {
    private final int valore;        // 1-10
    private final Seme seme;         // COPPE, DENARI, SPADE, BASTONI
    private final String nome;       // A, 2-7, F, C, R
    private final int punti;         // Punti per il calcolo del punteggio
    private final String risorsaNome; // Nome file immagine
}
```

Responsabilità: - Rappresenta una carta da gioco con valore, seme e punti - Calcola automaticamente la forza per la presa - Gestisce il mapping verso le risorse grafiche

`Seme.java`

```
public enum Seme {
    COPPE("[coppe]", Color.RED),
    DENARI("[denari]", Color.RED),
    SPADE("[spade]", Color.BLACK),
    BASTONI("[bastoni]", Color.BLACK);
}
```

Responsabilità: - Definisce i quattro semi del tresette - Associa colori per la visualizzazione - Fornisce simboli testuali per l'UI

`Giocatore.java`

```
public class Giocatore {
    private String nome;
    private List<Carta> mano;
    private boolean umano;
    private StatisticheGiocatore statistiche;
}
```

Responsabilità: - Modella un giocatore (umano o AI) - Gestisce la mano di carte - Mantiene le statistiche di gioco

Classi del Controller

`GameController.java` **Classe centrale dell'architettura** - Coordina tutti i componenti:

```
public class GameController {
    // Componenti specializzati
    private final GameStateManager gameState;
    private final DeckManager deckManager;
    private final ScoreCalculator scoreCalculator;
    private final TurnManager turnManager;
```

```
    private final AIPlayer aiPlayer;

    // Sistema di osservazione
    private final GameStateObservable gameObservable;
    private final GameView view;
}
```

Responsabilità principali: 1. **Orchestrazione**: Coordina tutti i manager specializzati 2. **Gestione Eventi**: Risponde agli input del giocatore 3. **Logica di Gioco**: Implementa le regole del tresette 4. **Comunicazione**: Interfaccia tra Model e View

Metodi chiave: - `avviaNuovoGioco()`: Inizializza una nuova partita - `giocaCarta(int indice)`: Gestisce il gioco di una carta - `gestisciTurnoAI()`: Coordina i turni dell'intelligenza artificiale - `valutaMano()`: Valuta il vincitore di una mano

`GameStateManager.java` **Centralizza tutto lo stato del gioco**:

```
public class GameStateManager {
    // Stati del gioco
    private boolean giocoInCorso, giocoInPausa, valutazioneInCorso;

    // Informazioni di gioco
    private int giocatoreCorrente, primoGiocatoreMano;
    private int mano, giocata, carteGiocateInMano;

    // Punteggi
    private double punteggioCoppia1Totale, punteggioCoppia2Totale;
    private double puntiBonus1, puntiBonus2;

    // Configurazione
    private final boolean modalitaDueGiocatori;
    private final double punteggioVittoria;
}
```

`DeckManager.java` **Gestisce il mazzo e la distribuzione delle carte**: - Creazione e mescolamento del mazzo - Distribuzione delle carte ai giocatori - Tracking delle carte giocate - Reset per nuove mani

`ScoreCalculator.java` **Calcola i punteggi secondo le regole del tresette**: - Valutazione del vincitore di ogni mano - Calcolo dei punti delle carte - Gestione dei bonus (ultima carta, ecc.) - Logica per determinare il seme richiesto

`TurnManager.java` **Gestisce la sequenza dei turni**: - Avanzamento del giocatore corrente - Gestione delle pause - Coordinamento con il sistema di AI - Gestione del primo giocatore di ogni mano

`AIPlayer.java` **Implementa l'intelligenza artificiale**: - Algoritmo di selezione delle carte - Logica strategica basata sul seme richiesto - Integrazione con il TurnManager per l'esecuzione asincrona

Classi della View

`Tresette.java` **Finestra principale del gioco**: - Implementa l'interfaccia `GameView` - Coordina tutti i pannelli dell'UI - Gestisce gli eventi dell'utente - Mantiene la referenza al GameController

Pannelli Specializzati `TopGamePanel.java`: Visualizza punteggi e informazioni di gioco `PlayedCardsPanel.java`: Mostra le carte giocate nella mano corrente `PlayerCardsPanel.java`: Gestisce l'interazione con le carte del giocatore `SideControlPanel.java`: Bottoni di controllo (pausa, nuovo gioco, ecc.)

Pattern Architetturali

1. Model-View-Controller (MVC)

Implementazione: - **Model**: Package `model` con entità di dominio - **View**: Package `view` con componenti UI - **Controller**: Package `controller` con logica di controllo

Vantaggi: - Separazione delle responsabilità - Facilità di testing - Manutenibilità del codice

2. Observer Pattern

Implementazione:

```
GameStateObservable --> notifica eventi
    |
    v
AudioObserver --> riproduce suoni
LoggingObserver --> log degli eventi
```

Utilizzo: - Notifiche di cambio stato - Eventi di gioco (carta giocata, mano finita) - Aggiornamenti dei punteggi

3. Factory Pattern

Implementazioni: - `UIComponentFactory`: Creazione componenti UI standardizzati - `IconFactory`: Gestione delle icone - `CardImageLoader`: Caricamento ottimizzato delle immagini

4. Strategy Pattern

Implementazione implicita: - `AIPlayer`: Diverse strategie in base al seme richiesto - `ScoreCalculator`: Diverse modalità di calcolo (2/4 giocatori)

5. Singleton Pattern

Implementazioni: - `SoundManager`: Gestione centralizzata dell'audio - `CardImageLoader`: Cache delle immagini con pattern di caricamento lazy

---

Flusso di Esecuzione

1. Avvio dell'Applicazione

```
Main --> MenuIniziale --> configurazione gioco --> Tresette(finestra principale)
```

2. Inizializzazione di una Partita

```
1. Tresette crea GameController
2. GameController inizializza tutti i manager:
   - GameStateManager (stato)
   - DeckManager (carte)
   - ScoreCalculator (punteggi)
   - TurnManager (turni)
   - AIPlayer (intelligenza artificiale)
3. Setup del sistema Observer
4. Creazione e distribuzione delle carte
5. Aggiornamento dell'interfaccia
```

3. Ciclo di Gioco Principale

```
LOOP per ogni mano:
  1. Determina il primo giocatore
  2. FOR ogni carta da giocare (10 carte per giocatore):
```

```
     a. Se turno umano:
        - Abilita interazione carte
        - Attende input utente
        - Valida carta selezionata
     b. Se turno AI:
        - AIPlayer seleziona carta
        - Esecuzione con timer (non bloccante)
     c. Aggiorna stato e interfaccia
     d. Passa al giocatore successivo
  3. Valuta vincitore mano
  4. Calcola e aggiorna punteggi
  5. Verifica condizione di vittoria
  6. Se partita non finita, nuova mano
```

4. Gestione Eventi UI

```
Evento utente --> Tresette --> GameController --> Manager appropriato --> Notifica Observer --> Aggiornamer
```

Interazioni tra Componenti

1. Interazione Controller-Model

```java
// GameController coordina i manager
public boolean giocaCarta(int indiceCarta) {
    // 1. Verifica stato
    if (!gameState.isGiocoInCorso()) return false;

    // 2. Valida carta
    Carta carta = giocatori[0].getMano().get(indiceCarta);
    if (!scoreCalculator.isCartaGiocabile(carta)) return false;

    // 3. Esegue azione
    deckManager.giocaCarta(carta, 0);

    // 4. Aggiorna stato
    gameState.avanzaTurno();

    // 5. Notifica observer
    gameObservable.notifyCartaGiocata(carta, "Giocatore");

    return true;
}
```

2. Interazione Model-View (via Observer)

```java
// AudioObserver risponde agli eventi
public void onCartaGiocata(Carta carta, String nomeGiocatore) {
    if (audioAbilitato) {
        SoundManager.riproduci("carta_giocata.wav");
    }
}

// View aggiorna interfaccia
public void onTurnoCambiato(String nome, int indice) {
    topPanel.aggiornaGiocatoreCorrente(nome);
```

```
    playerCardsPanel.abilitaInterazione(indice == 0);
}
```

3. Interazione AI-Controller

```
// TurnManager coordina l'AI
public void eseguiTurnoAI() {
    Timer timer = new Timer(AI_DELAY, e -> {
        int indice = aiPlayer.selezionaCarta(giocatoreCorrente);
        gameController.giocaCartaAI(giocatoreCorrente, indice);
    });
    timer.setRepeats(false);
    timer.start();
}
```

4. Gestione dello Stato Centralizzato

```
// Tutti i componenti accedono allo stato via GameStateManager
public class GameController {
    public boolean isGiocoInPausa() {
        return gameState.isGiocoInPausa(); // Stato centralizzato
    }

    public void mettiInPausa() {
        gameState.setGiocoInPausa(true);
        turnManager.pausaTurno();
        view.aggiornaPulsantiPausa();
        gameObservable.notifyPausaToggled(true);
    }
}
```

Gestione degli Errori e Robustezza

1. Validazione Input

- Verifica carte giocabili secondo le regole
- Controllo stati del gioco prima delle azioni
- Validazione parametri dei costruttori

2. Gestione Risorse

- Cache delle immagini per evitare ricaricamenti
- Gestione corretta dei timer dell'AI
- Cleanup automatico delle risorse audio

3. Stati Consistenti

- Stato centralizzato in GameStateManager
- Transizioni atomiche tra stati
- Rollback in caso di errori

Estensibilità e Manutenibilità

1. Aggiunta di Nuove Modalità di Gioco

- Estendere GameStateManager per nuovi stati

- Implementare nuove strategie in AIPlayer
- Aggiungere pannelli specifici nella view

2. Miglioramento dell'AI

- AIPlayer è completamente modulare
- Possibilità di implementare diversi livelli di difficoltà
- Integrazione con sistemi di machine learning

3. Personalizzazione dell'UI

- PaletteColori centralizza tutti i colori
- Factory pattern per componenti riutilizzabili
- Layout manager flessibili

Metriche del Codice

- **Linee di codice totali**: ~3500 LOC
- **Numero di classi**: 31 classi
- **Copertura Javadoc**: 100% (0 warning)
- **Complessità ciclomatica**: Bassa (max 10 per metodo)
- **Accoppiamento**: Basso grazie ai pattern utilizzati

Conclusioni

L'architettura del gioco Tresette rappresenta un esempio di **clean architecture** con:

1. **Separazione chiara delle responsabilità** tra layer
2. **Pattern ben consolidati** (MVC, Observer, Factory)
3. **Codice manutenibile** e facilmente estensibile
4. **Gestione robusta dello stato** centralizzata
5. **Interfaccia utente reattiva** e user-friendly

Il sistema è progettato per essere **scalabile** e **manutenibile**, con una struttura che facilita l'aggiunta di nuove funzionalità e la modifica di quelle esistenti senza impattare altri componenti.