Relazione Magazzino

Andrea Mariotti e Abderrahman Khouili

1 Analisi:

Il Team si pone l'obbiettivo lo sviluppo della versione digitale di una partita di Briscola:

- Ci sono due giocatori che hanno 3 tipi di carte:
    1. **Briscola:** indica il seme predominante della partita (Es: Bastoni, Denari, Coppe, Spadi);
    2. **Carta con valore:** carta che un valore significativo in base alla figura presente sopra la carta:
        2.1.Fante = 2pt;
        2.2.Cavallo = 3pt;
        2.3.Re = 4pt;
        2.4.Asso = 11pt;
        2.5.Tre = 10pt;
    3. **Carta con valore zero:** carta che non ha valore significativo ma ha valore in base al numero inciso sulla carta. Quest'ultime partono dalla carta con il numero due che è la più bassa mentre la carta più alta di valore è il sette(esclusa la carta con inciso il numero tre);

1.1 Requisiti

I **requisiti funzionali** della nostra applicazione sono:

1. I giocatori avviano una partita;
2. A turno i giocatori selezionano la carta;
3. Stampare la briscola;
4. Stampare la carta utilizzata dal giocatore che interviene per primo in quel determinato momento;
5. I turni vengono decisi in base a chi ha vinto "l'ultima presa" seguendo le regole della Briscola;

6. Viene implementato un metodo che permette ai giocatori di scambiarsi tra di loro per giocare;
7. Stampare il nome del giocatore che deve giocare in quel determinato momento;

I **requisiti non funzionali** presenti nella nostra applicazione sono:

1. I giocatori scrivono i loro nomi per identificarsi tra i diversi turni.
2. L'applicazione calcola il punteggio ogni turno (un turno=entrambi i giocatori lanciano una carta);
3. I giocatori hanno un bottone “esci della partita” che permette loro di tornare alla schermata principale in qualsiasi momento della partita;
4. Nella schermata principale dell’applicazione è presente un bottone che termina l’applicazione;

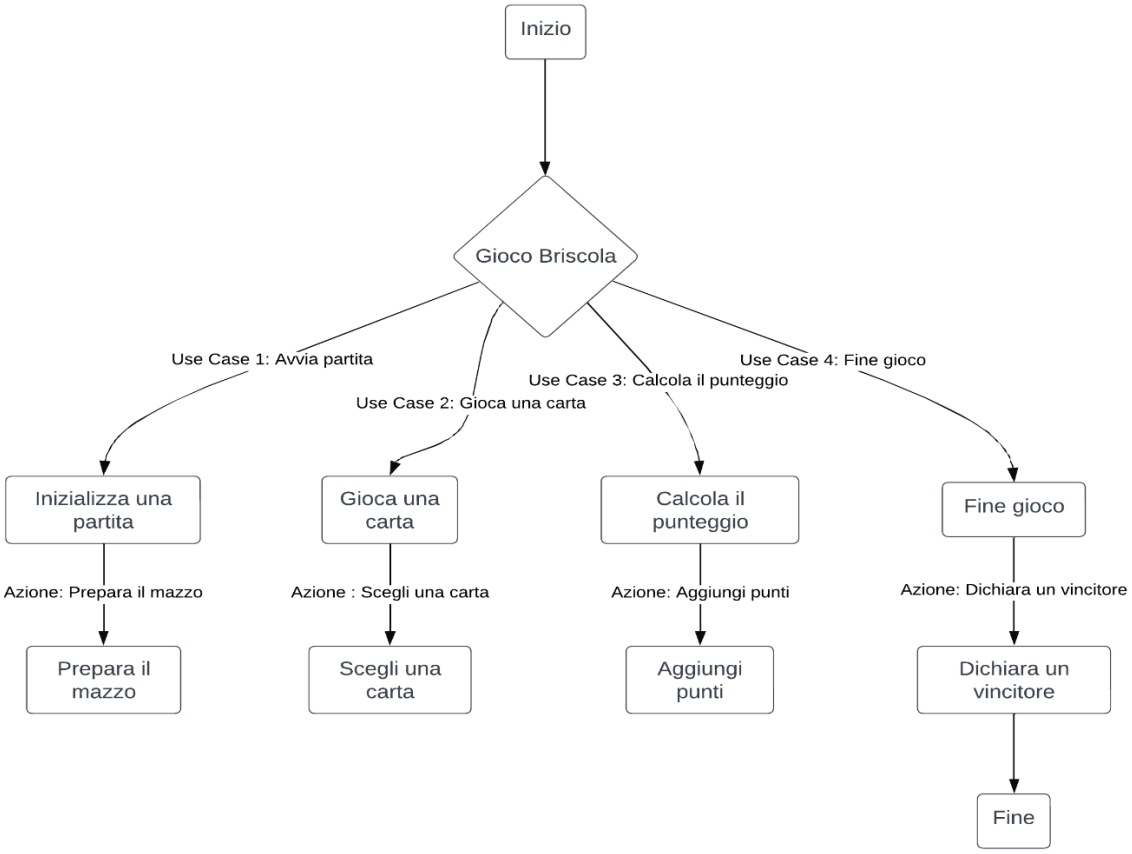


Figura 1: Use-case diagram

1.2 Analisi e modello del dominio

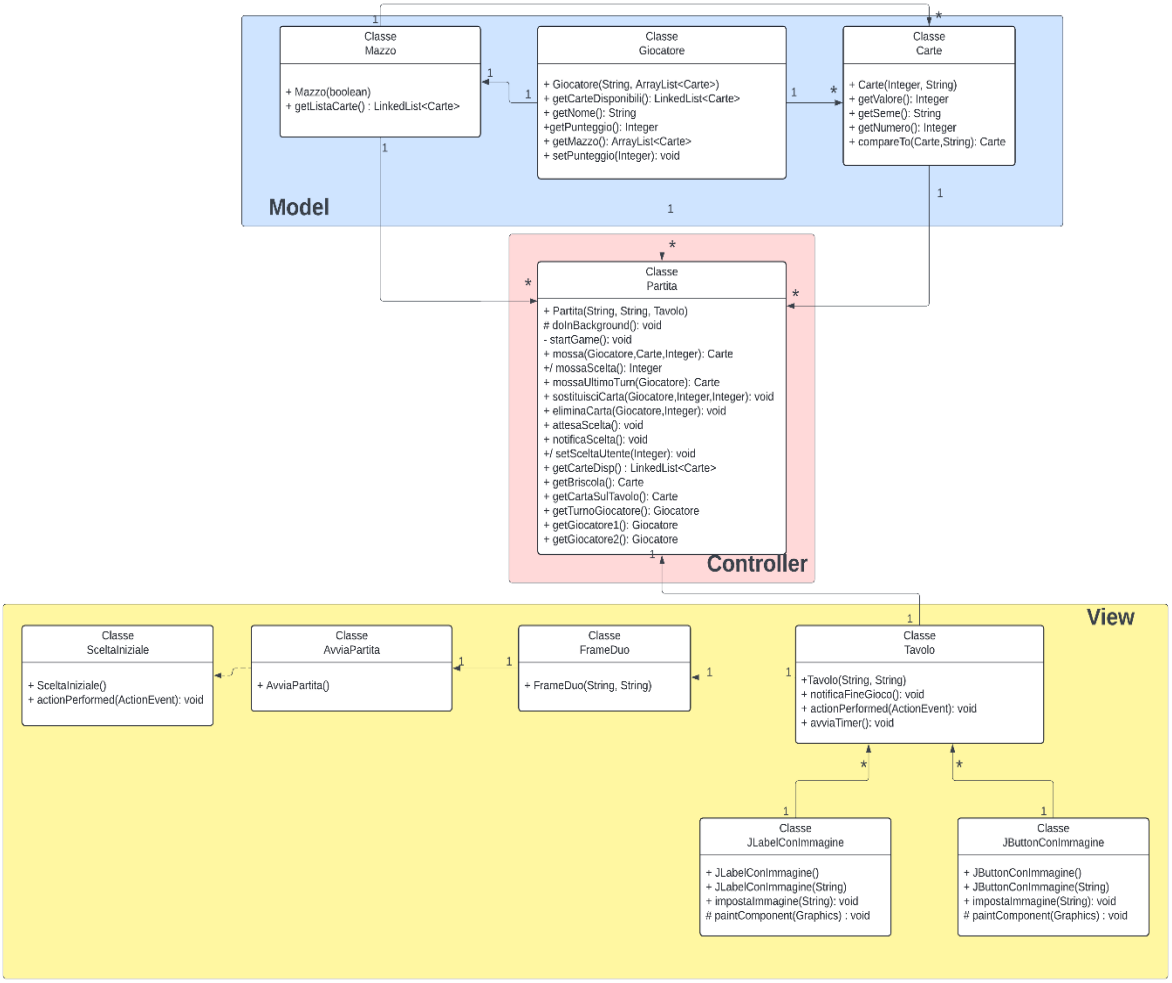


Figura 2: uml diagram completo

2 Design

2.1 Architettura

Per lo sviluppo dell'architettura si è scelto l'utilizzo del pattern Model View Controller in quanto consente una netta separazione della logica del programma dall'interfaccia grafica. Questo permetterà alle future modifiche nel design della GUI senza intaccare il funzionamento dell'intero sistema.

2.2 Model

Il model si occupa di gestire l'intero funzionamento del sistema implementando i requisiti funzionali in particolare, si occuperà di dare un valore, seme e numero ad ogni carta e fa un Sorting con Comparatori in base alle necessità di gioco, regolandone poi il punteggio dei due giocatori.

2.3 View

La view è la parte dell'applicazione con cui l'utente finale si interfaccerà direttamente. Si suddivide in tre interfacce grafiche, ognuna adibita a caricare o selezionare diverse opzioni: una per avviare o chiudere la partita, una per la scrittura dei nomi dei giocatori e una per il tavolo da gioco. Ognuna di queste view ha il compito di informare il controller di un'operazione che aggiornerà lo stato dell'applicazione. Questo permette di mantenere una separazione netta tra la view, il controller e il model in modo tale che se in futuro si vogliono apportare degli aggiornamenti nell'interfaccia grafica, non sarà necessario modificare nessuno degli altri due componenti dell'architettura.

2.4 controller

Il controller serve per fare da intermediario tra il model e la view nello specifico nella nostra applicazione la Partita (controller) gestisce tutto quello che succede tra quest'ultima e Mazzo, Carte e Giocatore. È stato implementato utilizzando il pattern Simple Factory che permette di costruire correttamente oggetti o famiglie di oggetti , il metodo non è statico e in una classe separata.

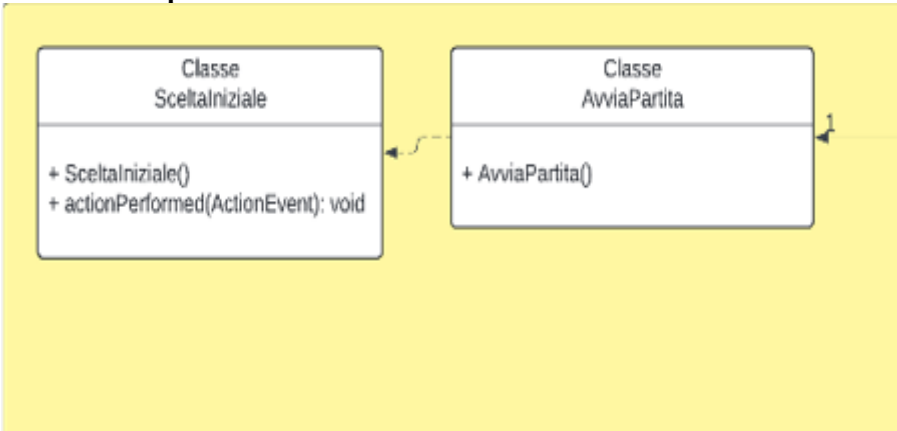
3 Sviluppo e Design dettagliato

Introduzione alla logica di pensiero e le varie scelte da noi prese per lo sviluppo della nostra applicazione.

3.1 Classe AvviaPartita.

Breve spiegazione dei metodi e classi presenti in AvviaPartita.

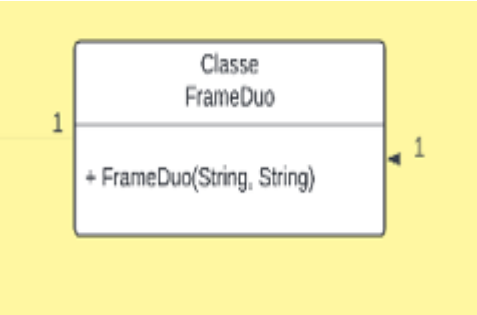
1. **Sceltainiziale**: permette di avviare l'applicazione mostrando il titolo di quest'ultima e ne permette la chiusura; detto questo sempre nella classe AvviaPartita abbiamo un JButton (avvia partita) che permette di intraprendere una partita.
2. **Metodo actionPerformed**: facendo l'override del metodo ActionPerfomed permette di utilizzare i JButton presenti nella classe SceltaIniziale.



3.2 Classe FrameDuo.

Breve spiegazione dei metodi e classi presenti in FrameDuo.

1. **FrameDuo**: inizializza la finestra per la classe Tavolo.



3.3 Classe Tavolo.

Concisa panoramica dei metodi e delle classi presenti in Tavolo.

1. **Tavolo**: regola lo sfondo, posiziona le carte dell'avversario coperte, la briscola e la carta utilizzata dal giocatore che parte per primo in quel determinato turno.
2. **notificaFineGioco**: avvisa i giocatori che la partita è finita.
3. **Metodo actionPerformed:** anche qui usiamo il metodo actionPerformed per gestire le scelte fatte dai giocatori.
4. **avviaTimer**: avvia un timer per permettere ai giocatori di scambiarsi tra di loro.

3.3.1 Classe JButtonConImmagine.

Panoramica breve sui metodi e sulle classi presenti in JButtonConImmagine.

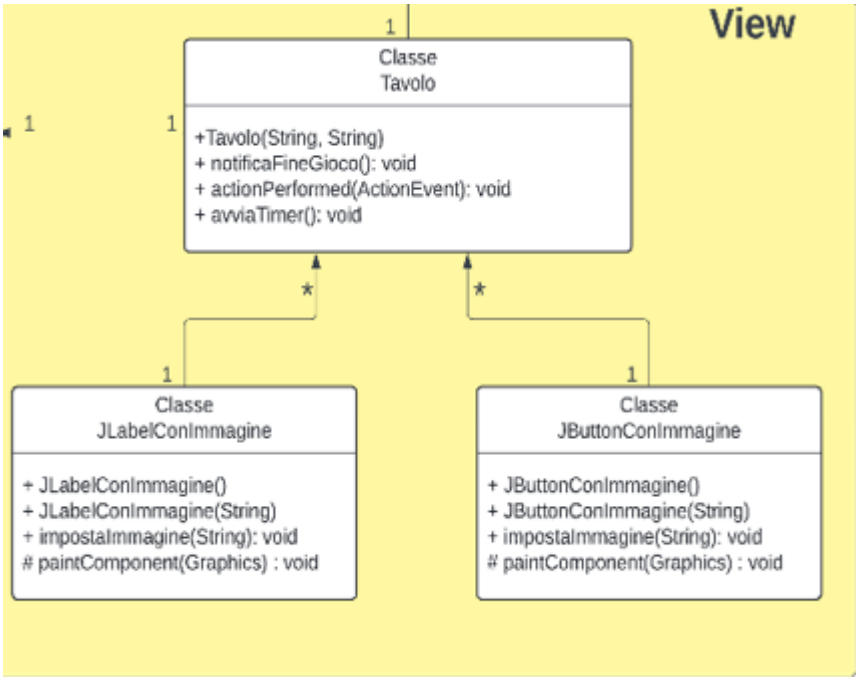
1. **impostaImmagine:** assegna un percorso immagine al JButton da usare come sfondo.

2. **paintComponent:** facendo l'override del metodo paintComponent possiamo disegnare il nostro sfondo sul JButton.

3.3.2 Classe JLabelConImmagine.

Sintesi relativa ai metodi e alle classi presenti in JLabelConImmagine.

1. **impostaImmagine:** assegna un percorso immagine al JLabel da usare come sfondo.
2. **paintComponent:** facendo l'override del metodo paintComponent possiamo disegnare il nostro sfondo sul JLabel.



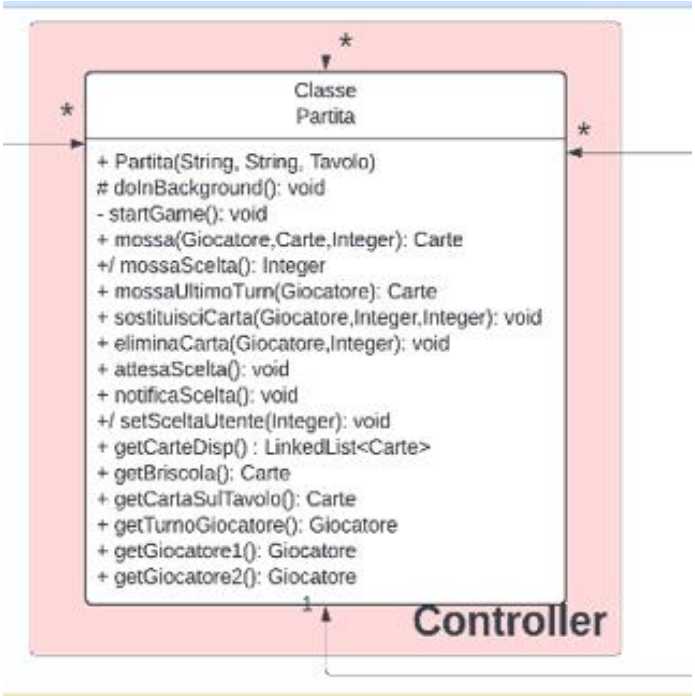
3.4 Classe Partita.

Rapida esposizione dei metodi e delle classi presenti in Partita.

1. **Partita:** inizializza la partita assegnando i due Giocatori e il tavolo.

2. **doInBackground:** questo nuovo metodo permette di creare una partita tramite il metodo **startGame** e mantenerla attiva in attesa della scelta del giocatore.
3. **mossa:** restituisce una carta (fra le 3 presenti in mano) in base alla scelta fatta del giocatore; in questo metodo vengono utilizzati quattro metodi aggiuntivi:
    - **mossaScelta:** conferma la mossa scelta dal giocatore però prima di intervenire passa tramite il metodo **attesaScelta** lo stato di Partita cambia in wait.
    - **setSceltaUtente:** imposta la scelta dell'utente a numero e modifica lo stato di Partita rendendolo di nuovo operativo, tramite il metodo **notificaScelta.**
4. **mossaUltimoTurn:** è un metodo che controlla che negli ultimi due turni se il giocatore ha solo una o due carte disponibili, esso può giocare solo la prima o la seconda (in base al turno in cui si trova).
5. **sostituisciCarta:** è il metodo che permette al giocatore di pescare una nuova carta.
6. **eliminaCarta:** questo metodo elimina la carta giocata cosicché non venga di nuovo pescata da uno dei due giocatori.
7. **getCarteDisp:** questo metodo serve per vedere la mano del giocatore in quel determinato turno.
8. **getBriscola:** questo metodo restituisce la briscola (che è un oggetto di tipo carte).
9. **getCartaSulTavolo:** mostra la prima carta giocata dal primo giocatore che la utilizza nel turno corrente.
10. **getTurnoGiocatore:** questo metodo restituisce il giocatore di quel turno.

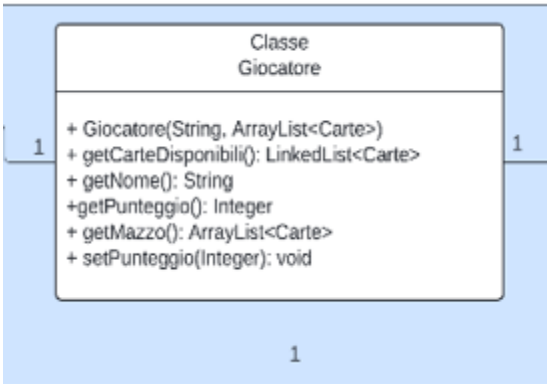
11. **getGiocatore1:** questo metodo restituisce il primo giocatore.
12. **getGiocatore2:** questo metodo restituisce il secondo giocatore.



3.5 Classe Giocatore.

Conciso resoconto dei metodi e delle classi presenti in Giocatore.

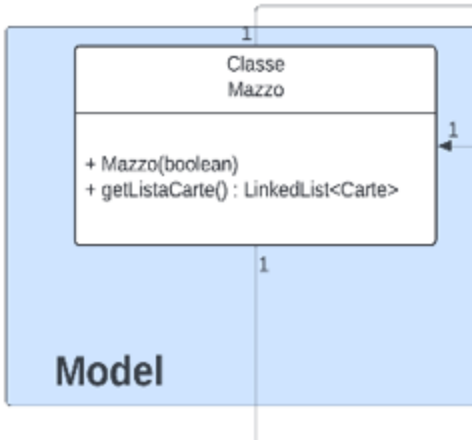
1. **Giocatore:** è il costruttore della classe e assegna nome, punteggio e mazzo al giocatore.
2. **getCarteDisponibili:** restituisce le carte utilizzabili dal giocatore.
3. **getNome:** restituisce il nome del giocatore.
4. **getPunteggio:** restituisce il punteggio del giocatore.
5. **getMazzo:** restituisce il mazzo del giocatore.
6. **setPunteggio:** assegna il punteggio al giocatore.



3.6 Classe Mazzo.

Esposizione del metodo e del costruttore all'interno della Classe Mazzo.

1. **Mazzo:** viene generato un mazzo contenenti le 40 carte della briscola e viene mescolato.
2. **getListaCarte:** restituisce la lista delle carte.

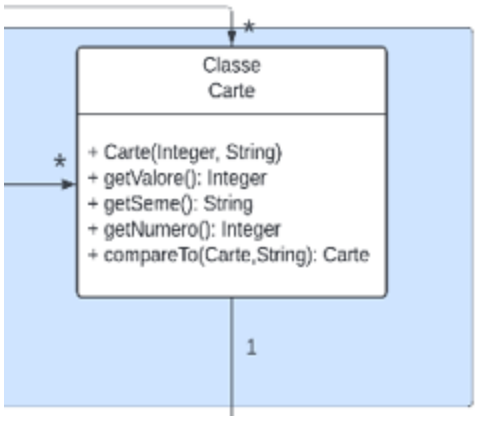


3.7 Classe Carte.

Conciso resoconto dei metodi e delle classi presenti in Carte.

1. **Carte:** inizializza il valore delle carte in base alle regole della briscola.
2. **getValore:** restituisce il valore della carta.
3. **getSeme:** restituisce il seme della carta.
4. **getNumero:** restituisce il numero della carta.

5. **compareTo:** seguendo le regole della briscola esegue una comparazione tra la prima carta giocata e la seconda.



4 Metodologia di lavoro

La fase iniziale è stata la fase di analisi dove abbiamo lavorato insieme nella creazione del class diagram. Successivamente nella parte del model abbiamo aggiornato tutti gli eventuali progressi effettuati di giorno in giorno, di conseguenza, vale anche per la parte del controller, della view e della relazione finale.

Version control Come DVCS (Distributed Version Control System) è stato utilizzato Git. È stato utilizzato un solo branch cioè il main dove sono state fatte tutte le release e in caso di collisioni con versioni precedenti veniva usato il merge.