Progetto PPS-24-Risk
Documentazione

Arcese Gabriele, matr: 0001183007
Colì Diego, matr: 0001172691
Costantini Marco, matr: 0001189584
Meco Daniel, matr: 0001192138

Luglio 2025

Sommario

Questo progetto ha l'obiettivo di sviluppare una versione digitale e distribuita del gioco da tavolo Risiko, realizzata come applicazione desktop multipiattaforma utilizzando il linguaggio di programmazione Scala. Il sistema è progettato per supportare due modalità di gioco: una modalità locale, in cui l'utente può affrontare avversari controllati da bot senza la necessità di connessione di rete, e una modalità online, che consente a più client remoti di partecipare a partite distribuite attraverso un server centrale. L'architettura è strutturata in moduli distinti: un core indipendente che implementa l'intera logica di gioco, un client desktop con interfaccia grafica per l'interazione dell'utente e un server centralizzato responsabile della gestione delle partite online e della sincronizzazione dello stato tra i partecipanti. L'infrastruttura consente inoltre di integrare bot in modo dinamico sia localmente sia per completare partite online con un numero ridotto di giocatori umani. Particolare attenzione è rivolta alla modularità, alla scalabilità e alla riutilizzabilità del codice, garantendo un'esperienza coerente e fluida in entrambi gli scenari di utilizzo. Il progetto comprende anche un'analisi dettagliata dei requisiti funzionali e non funzionali, con un focus su robustezza, reattività e aderenza alle regole originali del gioco.

Indice

1 Processo di sviluppo adottato

In questo capitolo viene descritta la metodologia adottata per lo sviluppo del progetto, evidenziando gli elementi principali e gli strumenti utilizzati dal gruppo per definire gli obiettivi e organizzare il lavoro durante l'intero periodo di realizzazione. Il processo di sviluppo scelto si è ispirato al modello SCRUM, come indicato dalle specifiche del progetto, con l'intento di sperimentare un approccio iterativo alla gestione del lavoro di squadra.
I membri del gruppo hanno ricoperto sia il ruolo di team di sviluppo, sia quelli previsti dalla metodologia SCRUM, assumendo in particolare i ruoli di product owner (Colì Diego) e SCRUM master (Arcese Gabriele). Una valutazione più dettagliata del processo adottato sarà approfondita nella sezione successiva.

1.1 Definizione degli obiettivi

Nei primi incontri dedicati al progetto, il gruppo ha deciso di organizzare alcune riunioni intensive con l'obiettivo di definire i requisiti generali del sistema, stabilire gli obiettivi funzionali da raggiungere e abbozzare un primo modello rappresentativo del dominio.
Questa fase iniziale, che si è sviluppata durante la prima settimana di lavoro, ha visto i membri del gruppo immedesimarsi nel ruolo dei committenti per poter redigere una proposta progettuale sufficientemente dettagliata.
È stato necessario trovare un equilibrio tra la definizione di specifiche chiare, che permettessero di avviare lo sviluppo in tempi rapidi, e un certo grado di flessibilità, utile a gestire eventuali evoluzioni e ampliamenti emersi nel corso dello sviluppo.
Oltre all'identificazione dei requisiti funzionali, il gruppo ha individuato alcuni obiettivi trasversali da mantenere durante tutto il progetto, in particolare la volontà di rilasciare costantemente un prodotto eseguibile, partendo da una versione prototipale e arricchendola progressivamente con nuove funzionalità al termine di ogni sprint.

1.2 Pianificazione e organizzazione del lavoro

Il Development Team è così suddiviso:

- Sviluppatore 1: Arcese Gabriele
- Sviluppatore 2: Colì Diego
- Sviluppatore 3: Costantini Marco
- Sviluppatore 4: Meco Daniel

In totale lo sviluppo è stato suddiviso in 4 sprint. In particolare, ogni sprint è della durata di circa dieci giorni ciascuno. Si è scelto inoltre di mantenere il backlog di ogni sprint salvato sulla bacheca per avere una istantanea del lavoro svolto durante ogni ciclo. Solitamente la modalità di riunione è stata svolta principalmente in presenza.

1.3 Strumenti di controllo della qualità

Per ottimizzare i tempi di sviluppo si è deciso di dare spazio ad una prototipazione agevole e di non applicare la filosofia stringente del **TDD (Test Driven Developement)** affrontata durante il corso.
Tuttavia, sono stati realizzati unit tests di ogni componente autonomo del modulo del **Core**. I test, sia automatizzati che manuali, sono il principale strumento di controllo della qualità e della funzionalità del codice prodotto.
L'esecuzione dei test è stata automatizzata con sbt in quanto, essendo un progetto Scala, si è scelto di gestire sia le dipendenze che i test tramite questo strumento. Oltre al testing automatizzato, review e retrospettiva sono state particolarmente utili per valutare i risultati sia a livello qualitativo che a livello tecnico.

2 Analisi dei Requisiti

Nel capitolo che segue verranno illustrati i requisiti funzionali dell'applicazione sviluppata nella prima fase del progetto. Vanno a descrivere ciò che i vari componenti del gruppo si aspettano dal sistema sviluppato. Il progetto consiste nello sviluppo di una applicazione che ha come obiettivo lo sviluppo di una versione digitale e distribuita del gioco da tavolo Risiko (regole su https://risiko.it/wp-content/uploads/2017/10/Regolamento-Risiko.pdf) utilizzando il linguaggio di programmazione Scala. Inoltre, per l'implementazione dell'intelligenza artificiale dei giocatori Bot, è stato integrato anche il linguaggio logico Prolog.

2.1 Business

Il committente richiede la progettazione e implementazione di un videogioco digitale ispirato al celebre gioco da tavolo Risiko!, con l'obiettivo di riprodurre in modo fedele e interattivo l'esperienza strategica del gioco originale, mantenendo l'aspetto ludico e competitivo che lo ha reso popolare. Gli obiettivi principali del progetto sono:

- Ricreare fedelmente l'esperienza di Risiko in formato digitale, offrendo una modalità di gioco a turni tra più giocatori, con dinamiche di conquista, gestione delle truppe e obiettivi segreti.
- Incorporare una logica di gioco modulare, in cui la gestione delle regole (battaglie, turni, distribuzione delle carte, condizioni di vittoria) sia completamente parametrizzabile e testabile, con il potenziale di supportare varianti o nuove modalità di gioco.
- Permettere l'integrazione di giocatori automatici (Bot) tramite un'interfaccia ben definita, in modo da consentire lo sviluppo di agenti intelligenti capaci di competere con gli esseri umani. Tali Bot potranno essere sviluppati con logiche custom, anche tramite linguaggi dichiarativi come Prolog.
- Fornire una base di gioco coerente e bilanciata, che includa: una mappa divisa in continenti e territori fedeli a quella originale, un sistema di carte territorio e obiettivo, regole consolidate per la fase di setup, attacco, spostamento e scambio carte.

2.2 Utente

L'utente deve poter:

1. Interagire col sistema tramite GUI interattiva.
2. Creare una nuova lobby.
3. Accedere a lobby esistenti.

4. Avviare una partita da una lobby contenente almeno 2 giocatori.
5. Aggiungere Bot alla partita (fino a un massimo di 6 giocatori totali)
6. Giocare con altri utenti connessi tramite Server centrale.
7. Effettuare azioni di gioco.

2.3 Funzionali

Il sistema deve:

1. Offrire due modalità di gioco: locale e multiplayer.
   (a) Locale: su un singolo computer devono poter essere eseguiti diversi Client, anche senza connessione Internet.
   (b) Multiplayer: diversi computer connessi alla stessa rete devono potersi connettere allo stesso Server.
2. Avviare una partita al raggiungimento dei giocatori richiesti dal creatore della lobby.
3. Stabilire l'ordine dei giocatori, mischiando gli stessi prima dell'inizio della partita.
4. Assegnare ad ogni giocatore un obiettivo segreto.
5. Far eseguire il turno di ciascun giocatore a rotazione.
6. Assegnare ad ogni giocatore i territori iniziali, posizionando una truppa su ogni territorio.
7. Distribuire ad ogni giocatore un ammontare di truppe in base al numero totale di essi.
8. Avviare una SetupPhase al primo turno di ogni giocatore, durante la quale ognuno di essi posiziona sui propri territori tutte le truppe distribuite al RF precedente.
9. Avviare una MainPhase dal secondo turno di ogni giocatore.
10. Calcolare le nuove truppe a disposizione di ogni giocatore.
11. Consentire di giocare tris di carte territorio possedute da ogni giocatore (secondo combinazioni prestabilite), in cambio di truppe aggiuntive.
12. Consentire il posizionamento delle truppe (azione PlaceTroops).
13. Consentire al giocatore di turno di effettuare azioni di gioco:

(a) Attack: da un territorio posseduto dal giocatore, avente almeno 2 truppe, verso un territorio nemico confinante. Se il giocatore conquista almeno territorio, il sistema deve assegnargli una carta territorio, che potrà essere utilizzata per giocare un tris.

(b) Reinforce: spostamento di truppe tra due territori confinanti, posseduti dallo stesso giocatore. Tale azione comporta il passaggio del turno al giocatore successivo.

(c) End Turn: passaggio del turno al giocatore successivo.

14. Consentire al giocatore la visualizzazione delle carte territorio possedute
15. Consentire al giocatore la visualizzazione del suo obiettivo.
16. Consentire al giocatore la visualizzazione della tabella contenente informazioni su territori e relativi proprietari e truppe posizionate.
17. Consentire al giocatore di abbandonare la partita.
18. Inviare notifiche a tutti i giocatori sugli esiti degli attacchi effettuati, sull'abbandono della partita da parte di un giocatore e sull'eventuale vittoria.
19. Garantire una comunicazione tra Client e Server tramite WebSocket
20. Garantire che il Client possa inviare azioni e ricevere aggiornamenti sullo stato di avanzamento del gioco.
21. Garantire che il Server possa gestire le connessioni, sincronizzare lo stato del gioco e inviare aggiornamenti a tutti i Client.

2.4 Non Funzionali

1. Il sistema deve garantire tempi di risposta adeguati per un'esperienza di gioco fluida, sia in locale che online.
2. Il server deve supportare un numero simultaneo di partite indefinito, gestendo efficacemente le risorse disponibili.
3. L'interfaccia utente deve essere intuitiva e coerente, facilitando l'apprendimento e l'utilizzo da parte di utenti con diversi livelli di esperienza.
4. L'applicazione desktop deve essere eseguibile su Windows, macOS e Linux, con un processo di installazione semplice e documentato.

2.5 Implementativi

- Java 17+.
- Scala 3.
- SBT (Scala Build Tool).
- Framework Akka per concorrenza e comunicazione di rete.
- ScalaFX per l'interfaccia grafica.

3 Analisi del Dominio

3.1 Glossario

- **Gioco**: La partita di Risiko, che include tutte le fasi (setup, turni, vittoria) e raccoglie l'insieme degli elementi di gioco (giocatori, territori, carte, ecc.).
- **Giocatore**: Un partecipante che può essere umano o bot, dotato di un colore per l'esercito, territorio, armate e una carta Obiettivo segreta.
- **Territorio**: Una regione sulla mappa mondiale. Ogni territorio ha un proprietario, un numero di armate, confini definiti e, nelle varianti tradizionali, non possiede un punteggio vittoria speciale.
- **Continente**: Un raggruppamento geografico di territori. Il controllo completo di un continente garantisce bonus armate al giocatore.
- **Mappa**: L'insieme di territori e continenti che costituiscono il piano di gioco, inclusa la definizione delle adiacenze tra territori.
- **Turno**: La sequenza di azioni che ogni giocatore effettua, suddivisa in fasi: Rinforzo, Attacco e Spostamento Strategico.
- **Battaglia**: Il confronto tra un territorio attaccante e uno difensore, risolto attraverso il lancio di dadi e il confronto dei risultati.
- **Dado**: Lo strumento di randomizzazione usato per determinare l'esito dei combattimenti. L'attaccante e il difensore lanciano dadi (fino a 3 a seconda delle truppe disponibili) e i valori vengono confrontati in ordine decrescente, in caso di parità vince il difensore
- **Carta**: Le carte che rappresentano i territori presenti sulla mappa. Vengono distribuite in fase di setup e servono anche per ottenere rinforzi extra tramite la formazione di tris.
- **Obiettivo**: Una carta assegnata segretamente a ciascun giocatore, che definisce la condizione particolare (obiettivo segreto) da raggiungere per vincere la partita.
- **Piazzamento**: La fase di ogni turno in cui un giocatore riceve armate aggiuntive, calcolate sulla base dei territori posseduti, dei bonus continentali e, eventualmente, tramite l'uso di tris di carte.
- **Attacco**: La fase del turno in cui un giocatore tenta di conquistare territori avversari lanciando i dadi e confrontando i risultati.
- **Rinforzo**: La mossa finale del turno, che permette al giocatore di riorganizzare le proprie armate spostandole tra territori contigui di sua proprietà, con il vincolo di lasciare sempre almeno un'armata nel territorio di partenza

- **Tris di carte**: Una combinazione di tre carte Territorio (con simboli uguali o combinati, e possibili carte Jolly) che, se giocate, forniscono armate aggiuntive durante la fase di Rinforzo
- **Eliminazione** : La condizione in cui un giocatore perde il possesso di tutti i territori, ed eventualmente viene rimosso dalla partita; l'attaccante ottiene le carte dell'eliminato.
- **Vittoria**: Lo stato finale del gioco che si verifica quando un giocatore realizza il proprio obiettivo segreto, terminando la partita

3.2 Servizi del dominio

Le entità centrali dell'applicazione sono:

- **Player**: giocatore (umano o bot) che partecipa alla partita
- **Bot**: giocatore controllato dalla CPU che elabora le sue strategie e ciclicamente decide quale azione è la migliore
- **Strategy**: le diverse strategie che il bot può seguire per scegliere le azioni migliori, differenziate a seconda della "modalità" di bot scelta
- **Territory**: territorio che compone la mappa di gioco, sono presenti gruppi di territori che compongono i diversi continenti (6 in totale), ogni territorio ha un owner, se tutti i territori di un continente sono posseduti da un singolo giocatore, questo riceverà un bonus di truppe ad ogni turno
- **Finestre di dialogo**: finestre che si aprono per ogni relativa azione che il giocatore che sceglie di effettuare
- **Mappa**: mappa di gioco visualizzata attraverso una UI da tutti i giocatori connessi alla partita
- **Fase di gioco**: differenzia la primissima fase di gioco in cui tutti i giocatori possono solamente piazzare le truppe iniziali, dalla main phase in cui si svolgono tutte le azioni possibili
- **GameEngine**: motore di gioco che contiene tutte le regole e le condizioni per poter svolgere le diverse azioni
- **Setup**: effettua il setup (ordine di gioco, distribuzione armate e carte)
- **GameAction**: tutte le possibili azioni che un giocatore può effettuare
- **TurnManager**: gestisce la logica dei turni e l'ordine in cui svolgere le diverse azioni e il passaggio del turno al giocatore successivo
- **Battle**: implementa la meccanica di combattimento (lancio dadi, confronto) ed aggiorna lo stato della partita con il risultato, aggiornando il numero di truppe e il possedimento del territorio attaccato in caso venga conquistato

- **CardManager**: gestisce il mazzo di carte Territorio e Obiettivo
- **ObjectiveValidator**: verifica se un giocatore ha raggiunto l'Obiettivo segreto
- **BonusCalcolator**: regole per la gestione e il controllo dei tris con relativi calcoli per truppe bonus a seconda del tris presentato.
- **Test**: classi in cui vengono testate le diverse funzionalità implementate
- **Message**: contiene i messaggi che rappresentano tutte le diverse operazioni che si possono effettuare (es: creare partite, partecipare a partite già create ecc...) che vengono scambiati tra client e server quando vengono svolte
- **GameSession**: si occupa della singola partita, viene istanziato un diverso session per ogni partita che viene creata
- **GameManager**: si occupa di gestire tutte le partite che vengono create, memorizzando le informazioni di esse come l'identificativo, il numero di partecipanti suddivisi tra umani e bot ecc..

4 Design architetturale

Si è scelta un'architettura modulare che suddivide chiaramente le responsabilità del sistema in componenti distinti, in modo da garantire manutenibilità, scalabilità e semplicità nello sviluppo, così da organizzare il lavoro di ciascun componente del team nel modo più efficiente possibile.

4.1 Moduli principali

- **Core:** Contiene la logica di gioco, le regole (movimenti, attacchi, vittoria), i modelli dati (mappa, territori, carte, obiettivi, giocatori, stato della partita). È indipendente dalla rete e dall'interfaccia utente. Viene utilizzato sia dal Client che dal Server, in quanto è grazie ad esso se le diverse azioni che vengono compiute dal giocatore vengono accettate e producono i risultati desiderati.
- **Client desktop:** Implementa l'interfaccia utente utilizzando **ScalaFX**, gestisce l'interazione dell'utente e comunica con il Server scambiando informazioni inerenti alle azioni che il giocatore sceglie di compiere. Consente due **modalità di gioco**:
  - **Remota**: Connessione al Server via WebSocket.
  - **Locale**: Istanzia direttamente la logica di gioco.
- **Server:** Gestisce le partite online, le connessioni dei Client, la sincronizzazione dello stato del gioco e l'integrazione dei Bot nelle partite. Viene utilizzato solo in modalità distribuita, non necessario in locale, ma condivide la logica con il modulo Core.
- **Bot**: Pensato per fornire all'utente la possibilità di confrontarsi in partita con giocatori controllati dall'intelligenza artificiale. Quest'ultima è in grado di prendere decisioni automatizzate e aumentare la flessibilità delle modalità di gioco. È inoltre possibile estendere le strategie che il Bot può utilizzare.

4.2 Interazione tra moduli

Una volta implementati e realizzati i diversi moduli, ed accertatoci che con i dovuti test le funzionalità richieste fossero state correttamente implementate, è stato necessario lavorare per mettere in comunicazione i diversi moduli affinchè potessero funzionare in maniera ottimale, per fare questo abbiamo utilizzato un approccio **pair programming**, ossia divisi in gruppi da due/tre per effettuare un integrazione tra i vari moduli in maniera corretta, ed accertarci del corretto funzionamento dell'applicazione.

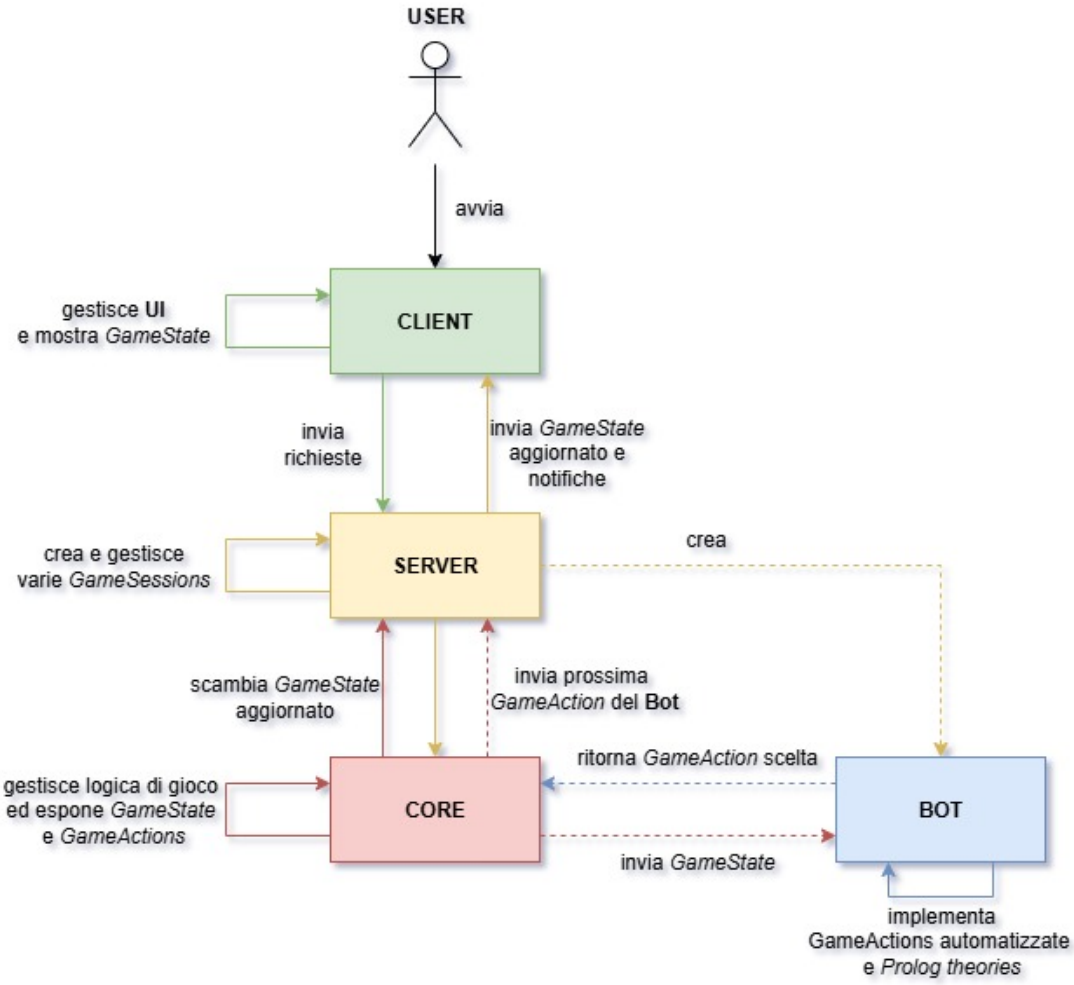

Figura 1: Architettura Applicazione

5 Design di dettaglio

5.1 Modulo Server

In questa parte analizziamo com'è stato realizzato il modulo Server e tutte le funzionalità che implementa.

- Il Server è stato progettato seguendo un'architettura reattiva e scalabile, basata sul framework **Akka**, il quale sfrutta Akka HTTP per la gestione delle connessioni web e Akka Actors per l'elaborazione concorrente e la gestione dello stato.
- L'architettura è stratificata per rispettare il più possibile il SRP, rendendo il sistema robusto, manutenibile e testabile.
- `RisikoServer` è il punto di ingresso del sistema: avvia un server HTTP e definisce gli endpoint. La comunicazione principale avviene tramite un endpoint WebSocket , gestito dal `WebSocketHandler`. Quest'ultimo adotta il pattern "un attore per sessione": per ogni Client che si connette, viene creato un `ConnectionActor` dedicato. Questo attore agisce da intermediario, traducendo i messaggi JSON del Client in comandi interni e inoltrandoli al `GameManager`.
- `GameManager` è un attore singleton che funge da orchestratore centrale. Non conosce le regole del gioco, ma gestisce il ciclo di vita di tutte le partite. Riceve richieste per creare nuove partite o unirsi a quelle esistenti e, di conseguenza, crea e supervisiona attori `GameSession` figli.
- Ogni `GameSession` incapsula la logica e lo stato di una singola partita. Funziona come una macchina a stati finiti, gestendo le fasi del gioco (attesa, setup, gioco, fine). Questo attore riceve le azioni di gioco dal `GameManager` e le delega al `GameEngine`, il componente cardine del Core (contenente pure logica), completamente disaccoppiato dal sistema di attori. Il `GameEngine` applica le regole di Risiko, calcola il nuovo stato del gioco e lo restituisce alla `GameSession`, che provvede poi a notificarlo a tutti i giocatori coinvolti.
- Questa architettura a più livelli, dal `ConnectionActor` al `GameManager` fino alla `GameSession` e al `GameEngine` del Core, garantisce indipendenza e definizione chiara di gestione della rete, gestione delle partite e della logica del gioco.

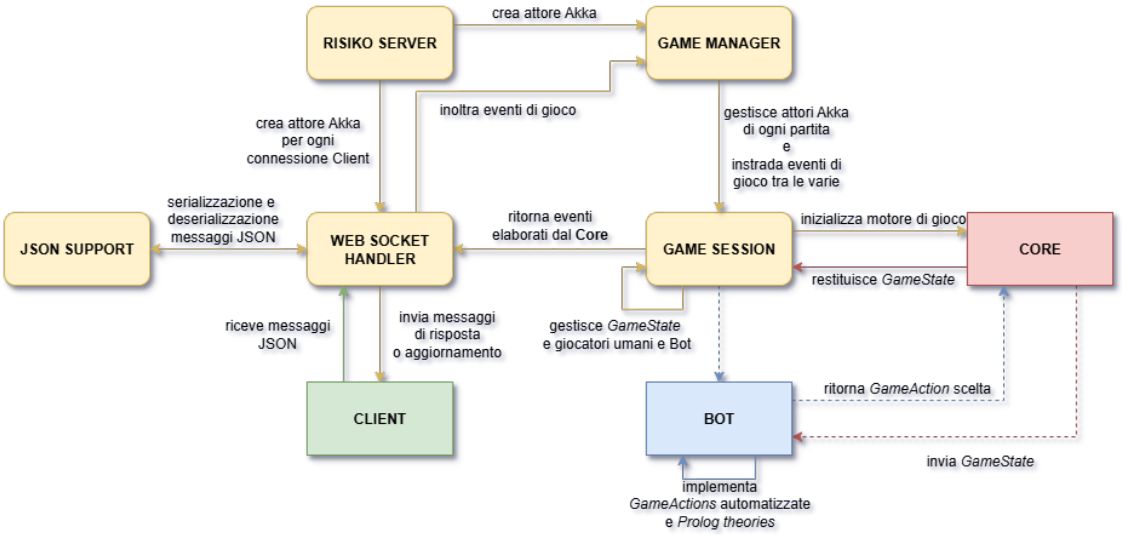

Figura 2: Design Modulo Server

5.2 Modulo Client

In questa parte analizziamo com'è stato realizzato il modulo Client e tutte le funzionalità che esso implementa.

- Il Client è stato progettato per offrire una UI reattiva e intuitiva, permettendo ai giocatori di interagire in tempo reale con il Server. L'architettura segue il SRP, separando la gestione della rete, dello stato locale e dell'interfaccia grafica.
- `ClientUI` rappresenta il punto di ingresso: avvia l'applicazione, gestisce la finestra di login e la connessione al Server tramite WebSocket. Dopo il login, viene mostrata la `LobbyWindow`, che consente di visualizzare le partite disponibili, crearne di nuove o unirsi a una partita esistente.
- La comunicazione con il Server avviene principalmente tramite WebSocket, gestito dal componente `ClientNetworkManager`. Client si occupa di stabilire la connessione, inviare messaggi (come azioni di gioco o richieste di lobby) e ricevere aggiornamenti in formato JSON. I messaggi ricevuti vengono decodificati tramite `ClientJsonSupport` e smistati ai vari componenti tramite un sistema di callback.
- La UI è realizzata con **ScalaFX** e organizzata in finestre e pannelli modulari:
  - `LobbyWindow` gestisce la lobby e la lista delle partite.

  - `GameWindow` rappresenta la finestra principale della partita, mostrando la mappa, le informazioni sui giocatori, le carte e le azioni disponibili.
  - Componenti come `TerritoryInfoPane`, `DiceDisplay`... si occupano di visualizzare e aggiornare le diverse parti dell'interfaccia in risposta agli eventi di gioco.

- Le azioni dell'utente vengono gestite tramite `GameActionHandler`, che costruisce i messaggi appropriati e li invia al Server tramite `ClientNetworkManager`. Ogni azione è asincrona e il Client attende la conferma dal Server prima di aggiornare lo stato locale.

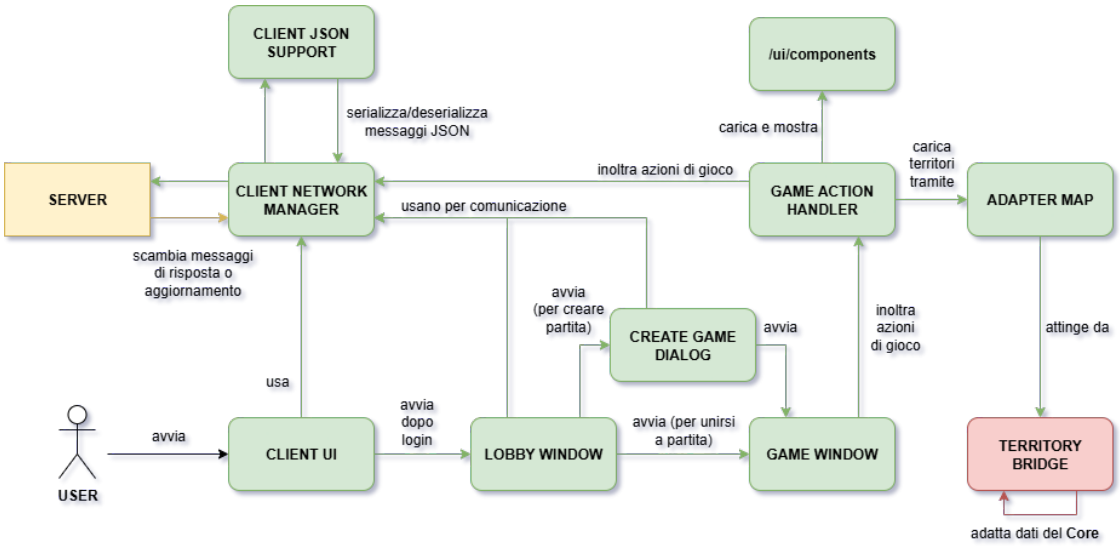

Figura 3: Design Modulo Client

5.3 Modulo Core

In questa parte analizziamo com'è stato realizzato il modulo Core e tutte le funzionalità che esso implementa.

- Rappresenta il cuore logico dell'applicazione: si occupa di implementare e far rispettare le regole del gioco Risiko, gestendo lo stato della partita, i turni, le azioni dei giocatori e la logica di battaglia. È stato progettato in maniera funzionale e puramente deterministica, seguendo il SRP e il principio dell'immutabilità.

- `GameEngine` è il componente principale: gestisce l'intera logica di gioco e funge da "collante" tra le varie componenti. Riceve azioni `(GameAction)` dai giocatori o dai Bot, le valida tramite il `TurnManager` e aggiorna il `GameState` di conseguenza. Gestisce anche la risoluzione delle battaglie, l'assegnazione delle carte, la verifica della condizione di vittoria e la rotazione dei turni.

- Tutte le azioni passano attraverso un ciclo chiaro: il `TurnManager` verifica se l'azione è valida rispetto alla fase corrente `(TurnPhase)`, dopodiché viene eseguita tramite `GameEngine`, che produce un nuovo **GameState** aggiornato.

- Quando è necessario risolvere una battaglia, il `GameEngine` invoca `Battle`, che implementa la logica di confronto tra attaccante e difensore, simulando i dadi (con `Dice`) e restituendo un `BattleRoundResult`.

- Questo modulo gestisce anche la distribuzione e il mescolamento delle carte tramite `DecksManager`, l'estrazione delle `(ObjectiveCard)` e la loro verifica (con `ObjectiveValidator`), e calcola i bonus di inizio turno o di scambio carte tramite `BonusCalculator`.

- `BotController` è l'interfaccia tra l'engine e i Bot, ossia giocatori automatizzati che possono decidere la propria mossa in base allo stato corrente.

- `GameEngine` fa della sua forza l'immutabilità del `GameState`, infatti ogni suo metodo lo riceve in input e ne restituisce uno aggiornato.

- L'interazione tra Core e Server avviene solo tramite scambio di `GameState` e `GameAction`, mantenendo così un forte disaccoppiamento tra logica di gioco e infrastruttura di rete.

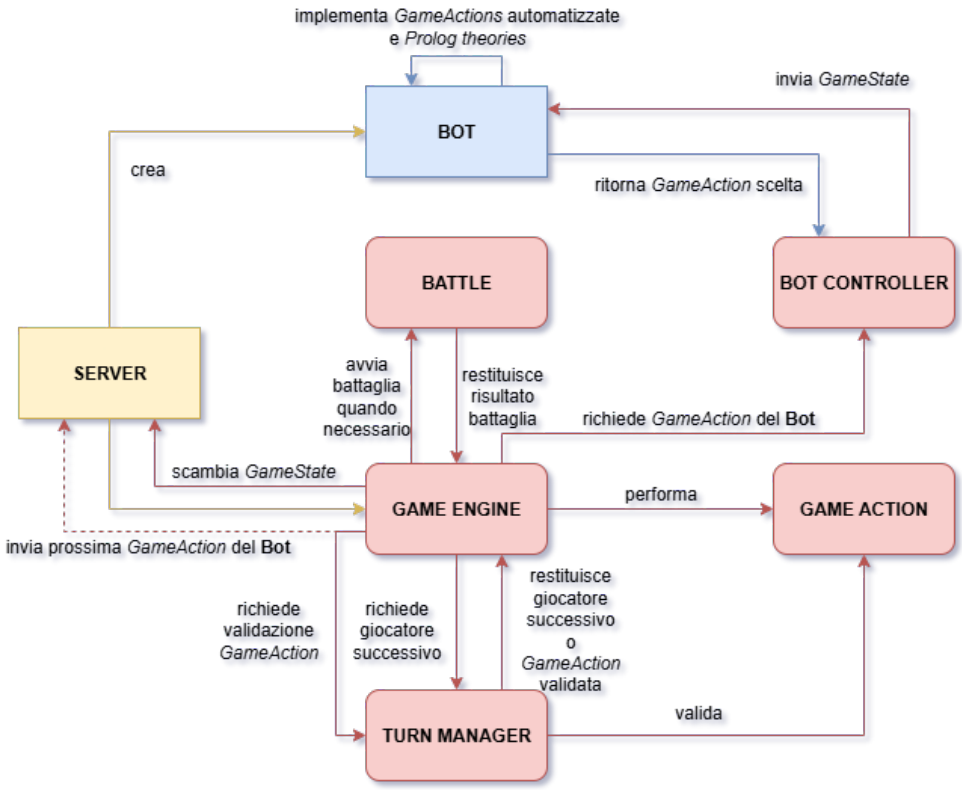

Figura 4: Design Modulo Core

5.4 Modulo Bot

In questa parte analizziamo come sono stati realizzati i Bot, e tutte le tecniche utilizzate per stabilire le strategie migliori.

- Un Bot è di fatto un giocatore controllato dalla CPU, il quale può effettuare le stesse identiche azioni di un giocatore reale.
  Proprio per questo, la classe `BotPlayer`, la quale definisce di fatto tutte le caratteristiche di un giocatore Bot, estende alcune delle sue proprietà dal trait `Player` del modulo Core.

- Al giocatore che crea la partita viene data la possibilità di scegliere quale strategia (offensiva o difensiva/conservativa) dovrà seguire ciascun Bot (se presenti), al fine di rendere la partita più imprevedibile. Tali strategie sono state realizzate a partire da regole **Prolog**, distinte in base alla strategia scelta dall'utente. Per far sì che le regole logiche potessero essere interpretate correttamente dal linguaggio Scala, è stato necessario costruire ad-hoc

le classi `PrologRule` e `PrologEngine`, le quali si occupano del parsing e della valutazione delle regole, con l'ausilio del singleton `StrategyRule`.

- Ad ogni turno del Bot, viene calcolato un punteggio per ogni singola regola applicabile e restituita al modulo Core la relativa azione di gioco avente punteggio più alto.

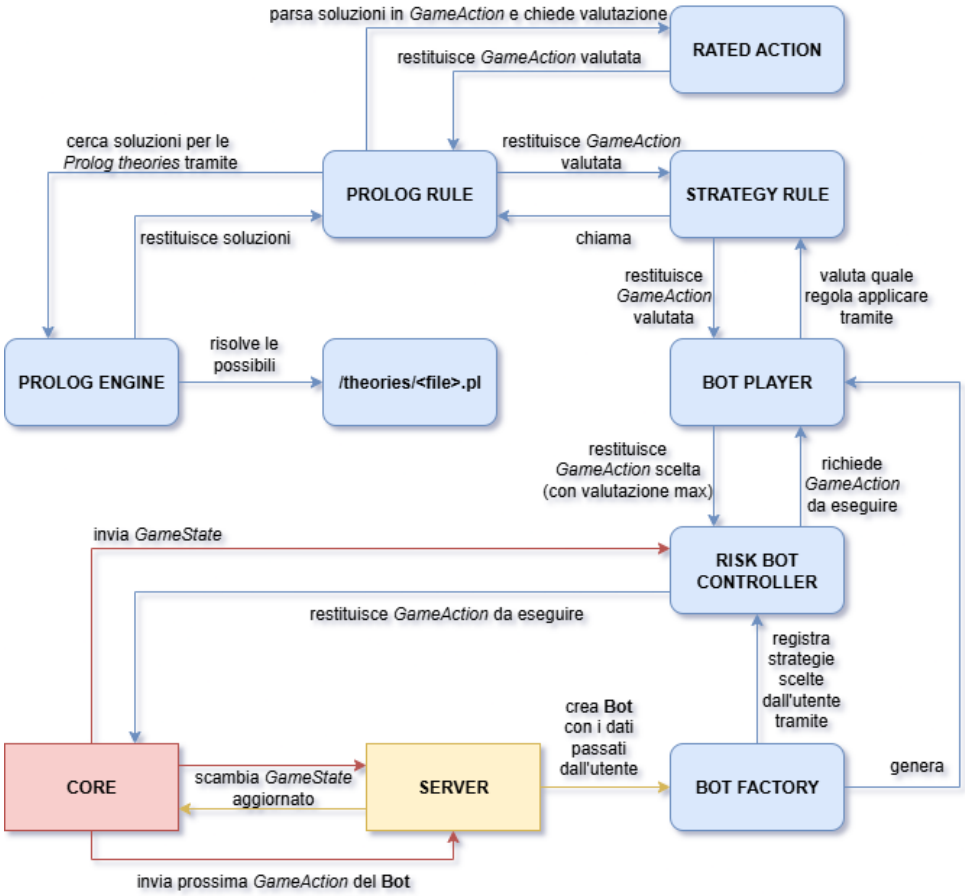

Figura 5: Design Modulo Bot

6 Implementazione

Di seguito viene descritto il lavoro compiuto da ciascun membro del team.

6.1 Arcese Gabriele

Per quanto riguarda il mio contributo all'implementazione, mi sono occupato principalmente dei moduli Bot, implementando la struttura base, Client, aggiungendo finestre di dialogo utilizzabili dall'utente, Core, effettuando modifiche alle classi già esistenti per permettere la comunicazione con il Client.

6.1.1 Implementazione Bot

Nel corso della costruzione di questo modulo, avvenuta in collaborazione con Colì Diego per unire le regole prolog alle strategie che il bot può seguire, gli elementi attribuibili al sottoscritto sono, in particolare:
Files di `/bot`;
Files di `/strategy`.

Esempio file di /bot: `BotFactory.scala`

```
object BotFactory:

  val controller = new RiskBotController()

  def createAggressiveBot(playerId: String, name: String, color:
      PlayerColor): (BotPlayer, RiskBotController) =
    val aggressiveRules = Set[StrategyRule](
      //tutte le regole prolog per il bot offensivo
    )
    val botPlayer = new BotPlayer(playerId, name, color, aggressiveRules)
    controller.registerStrategy(playerId, botPlayer)
    (botPlayer, controller)
```

Aspetti implementativi: questa classe si occupa della creazione di un giocatore di tipo Bot, il quale può essere di due tipologie: offensivo e difensivo. Una volta creato, viene registrato nel `RiskBotController` un set di `StrategyRule`, ossia un set regole che il Bot deve rispettare per seguire la sua strategia.
Tale `RiskBotController` si occupa di valutare la migliore azione possibile, in base però solo alle regole che possiede.
Il `BotFactory` è stato implementato seguendo il pattern factory object, in quanto incapsula la logica per costruire i `BotPlayer`, con strategie diverse ma evitando di duplicare nel Server il codice che si occupa della loro creazione. Inoltra isola la creazione delle regole ed il collegamento con il `RiskBotController`, consentendo eventuale estendibilità futura.
Segue inoltre l'OCP, in quanto se fosse necessario in futuro aggiungere un nuovo tipo di Bot (oltre ai già esistenti offensivo e difensivo), basterà creare un

nuovo metodo in `BotFactory`, senza dover modificare ulteriore implementazione esistente. Essendo che questa classe si occupa esclusivamente della creazione del Bot e della registrazione delle strategie, senza la gestione/valutazione delle azioni, rispetta l'SRP.

Esempio file di /bot: `RiskBotController.scala`

```
class RiskBotController extends BotController:

  private val botStrategies = mutable.Map[String, Strategy]()

  def registerStrategy(playerId: String, strategy: Strategy): Unit =
    botStrategies.put(playerId, strategy)

  override def nextAction(gameState: GameState, playerId: String):
      GameAction =
    botStrategies.get(playerId) match
      case Some(strategy) => strategy.decideMove(gameState)
      case None => lancio eccezione per nessuna strategy trovata
```

Aspetti implementativi: questa classe estende il trait `BotController` del Core, implementando il metodo `registerStrategy()`, il quale si occupa di registrare per ogni Bot quelle che sono le regole che possiederà, ed eseguendo l'override di `nextAction()`, che restituisce la GameAction da performare. Per l'implementazione è stato seguito il pattern strategy, che consente di incapsulare le strategie in maniera che siano intercambiabili e che possano essere utilizzate in modo flessibile.

6.1.2 Implementazione Client

La costruzione di questo modulo è avvenuta in collaborazione con Meco Daniel (principale sviluppatore del modulo) e con Costantini Marco (sviluppatore del modulo Core). L'obiettivo principale di questa implementazione è stato quello di far interfacciare correttamente l'UI del Client con la logica interna del Core. Gli elementi attribuibili al sottoscritto sono, in particolare:
Files di `/dialogs`;
Files di `/ui`.

Esempio file di file/ui: `GameWindow.scala`

```
class GameWindow(
  networkManager: ClientNetworkManager,
  gameId: String,
  gameName: String,
  initPlayers: List[String],
  myUsername: String,
  val myPlayerId: String,
  playerColors: Map[String, String] = Map.empty
) extends Stage {

    //parametri di configurazione della finestra di gioco

    private def registerCallbacks(): Unit = {
        networkManager.registerCallback("gameStarted", msg => {
            handleGameStarted(msg.asInstanceOf[GameStartedMessage])
        })
        networkManager.registerCallback("gameState", msg => {
            val gameState = msg.asInstanceOf[GameState]
            currentPhase = gameState.state.currentPhase
            handleGameState(gameState)
            })
         ...
    }

    private def setupUIEventHandlers(): Unit = {
        actionPane.attackButton.onAction = handle {
          val myTerritories = territories.filter(_.owner.value ==
              myPlayerId)
          if (myTerritories.isEmpty)
            showErrorAlert("Non hai territori da cui attaccare")

          val attackDialog = new AttackDialog(
            this,
            myTerritories,
            territories.filter(_.owner.value != myPlayerId)
          )
```

```
      val result = attackDialog.showAndWaitWithResult()
      result.foreach { attackInfo =>
        actionPane.attackButton.disable = true
        actionHandler.attack(...).onComplete {
          case scala.util.Success(true) =>
            Platform.runLater {
              actionPane.attackButton.disable = false
            }
          case _ =>
            Platform.runLater {
              actionPane.attackButton.disable = false
              showErrorAlert("Errore nell'esecuzione dell'attacco")
            }
        }(networkManager.executionContext)
      }
    }
      ...

  private def handleBattleResult(battleResult: BattleResultMessage):
      Unit = {
    Platform.runLater {
      try {
        // aggiorna dati attaccante
        diceDisplay.updateValues(battleResult.attackerDice,
            battleResult.defenderDice)

        val outcomeText = if (battleResult.conquered)
          s"Territorio ${battleResult.defenderTerritory} conquistato!"
        else
          s"Attacco a ${battleResult.defenderTerritory} respinto."

        val attackerId = territories.find(_.name ==
            battleResult.attackerTerritory)
          .map(_.owner.value).getOrElse("Sconosciuto")

        val attackerName = playersList.map(_.nameLabel.text.value)
          .find(_.contains(s"($attackerId)"))
          .getOrElse("Sconosciuto")

        val attackMessage = s"Attacco di ${attackerName} da
            ${battleResult.attackerTerritory} a
            ${battleResult.defenderTerritory}"
        val lossesMessage = s"$outcomeText\nPerdite: Attaccante
            ${battleResult.attackerLosses}, Difensore
            ${battleResult.defenderLosses}"

        val alert = new Alert(Alert.AlertType.Information) {
            initOwner(GameWindow.this)
            title = "Risultato Battaglia"
            headerText = attackMessage
```

```
            contentText = lossesMessage
          }
          alert.show()
        }
      }
}
```

Aspetti Implementativi: `GameWindow` è la classe principale che si occupa di far visualizzare al giocatore l'interfaccia con cui si interfaccerà per poter compiere azioni di gioco, in particolare è stata implementata seguendo il seguente flusso:

- **Inizializzazione e composizione UI**: Quando viene istanziata, `GameWindow` riceve i dati di sessione e crea tutti i componenti UI: mappa, sidebar, action panel, indicatori, bottoni, ecc.
  I territori sono caricati tramite `AdapterMap.loadTerritories()`, che restituisce una `ObservableBuffer[UITerritory]` con proprietà osservabili (immutabilità e reattività).
  I componenti come GameMapView e i vari pannelli sono composti in modo dichiarativo e funzionale (costruzione tramite parametri e funzioni).

- **Registrazione dei callback**: La funzione `registerCallbacks()` associa funzioni (callback) ai tipi di messaggi ricevuti dal server tramite il `ClientNetworkManager`. Ogni callback è una funzione pura che effetua pattern-matching sul messaggio e aggiorna lo stato/UI. Questo pattern favorisce la disaccoppiamento tra rete e logica di presentazione.

- **Gestione eventi asincroni**: Quando arriva un messaggio, il relativo callback viene invocato. I dati ricevuti sono trattati come immutabili e usati per aggiornare la UI tramite funzioni pure (es. `handleGameState`, `handleBattleResult`...). L'aggiornamento della UI avviene sempre su **thread JavaFX** tramite `Platform.runLater`, garantendo la sicurezza dei dati.

- **Propagazione dello stato**: I componenti UI osservano le proprietà dei territori e dei giocatori (`ObservableBuffer`, `StringProperty`...). Quando lo stato cambia (es. cambio proprietario di un territorio), la modifica si propaga automaticamente a tutti i giocatori connessi alla partita, così facendo interagiscono sempre con una versione consistente della partita. Questo pattern è tipico della programmazione reattiva/funzionale.

- **Gestione delle azioni utente**: I bottoni e le azioni UI compiute dal giocatore invocano metodi che costruiscono messaggi (attacco, rinforzo, fine turno...) tramite `GameActionHandler`, che usa funzioni pure per generare i dati. Le azioni sono inviate al server tramite il network manager, che gestisce la serializzazione/deserializzazione in modo funzionale (`ClientJsonSupport`).

- **Dialoghi e feedback**: I dialoghi (piazzamento truppe, attacco, obiettivi...) sono creati e mostrati in modo dichiarativo, spesso con callback di ritorno che gestiscono il risultato. L'uso di funzioni di ordine superiore e pattern matching per gestire i risultati dei dialoghi è un altro aspetto funzionale.

- **Interazione tra componenti**: `GameWindow` è il coordinatore: riceve eventi dal server, aggiorna lo stato e notifica i componenti.
  `AdapterMap` fornisce la conversione tra modello core e UI, garantendo che i dati siano sempre consistenti e osservabili.
  `GameActionHandler` incapsula la logica delle azioni di gioco, separando la costruzione dei messaggi dalla UI.
  `ClientNetworkManager` gestisce la comunicazione e la registrazione dei callback, permettendo una reazione funzionale agli eventi di rete.
  Componenti UI (map, sidebar, action pane, dialoghi) osservano lo stato e si aggiornano automaticamente grazie alle proprietà osservabili e ai buffer reattivi.

- **Pattern funzionali evidenziati**: Il codice sfrutta diversi pattern funzionali propri della programmazione in Scala, che favoriscono chiarezza, modularità e assenza di effetti collaterali indesiderati.

  - Immutabilità: i dati ricevuti dal server, come lo stato del gioco o le informazioni sui territori, non vengono mai modificati direttamente. Vengono invece utilizzati per generare nuovi stati o aggiornare la UI, mantenendo così il paradigma dell'immutabilità e riducendo il rischio di effetti collaterali.
  - Funzioni pure: molte delle funzioni utilizzate per elaborare i dati o costruire le azioni da inviare al server sono funzioni pure, ovvero producono un output deterministico dato un input, senza modificare lo stato globale o generare effetti collaterali.
  - Pattern matching: viene impiegato per distinguere i diversi tipi di messaggi ricevuti dal server. Questo approccio permette di scrivere codice più leggibile e sicuro, specialmente nella gestione di azioni come l'attacco (come mostrato nell esempio di codice) o la fine del turno.
  - Composizione: l'interfaccia utente è costruita in modo modulare, componendo componenti grafici (`VBox`, `HBox`, `Button`, `Label`) attraverso funzioni e parametri. Questo facilita la riusabilità del codice e l'organizzazione funzionale dell'interfaccia.
  - Programmazione reattiva: l'utilizzo di strutture come `ObservableBuffer` e proprietà osservabili consente un aggiornamento automatico e coerente della UI ogni volta che lo stato sottostante cambia, favorendo un modello di programmazione reattivo ed efficiente.

– Callback come funzioni: la gestione degli eventi dell'interfaccia grafica, così come le risposte del server, è basata sul passaggio di funzioni come parametri (ad esempio con `handle {...}` o `msg => ...`). Questo approccio è tipico della programmazione funzionale e rende il flusso degli eventi più chiaro e componibile.

6.1.3 Implementazione Core

Nel corso della costruzione di questo modulo, avvenuta in collaborazione con Costantini Marco, è stato necessario garantire la corretta comunicazione con il modulo Client, in modo tale che quest'ultimo potesse utilizzare e interpretare adeguatamente le classi del Core. Gli elementi attribuibili al sottoscritto sono, in particolare:
Files del package `/utils`;
Files del package `/engine`.

Esempio file di /utils: `BonusCalculator.scala`

```
def calculateTradeBonus(cards: Seq[TerritoryCard]): Int =
  val imgs = cards.map(_.cardImg)
  if (imgs.size != 3) 0
  else if (imgs.forall(_ == CardImg.Artillery)) 4
  else if (imgs.forall(_ == CardImg.Infantry)) 6
  else if (imgs.forall(_ == CardImg.Cavalry)) 8
  else if (imgs.distinct.size == 3) 10
  else 0

def calculateStartTurnBonus(playerId: String, board: Board): Int =
  val territoriesOwned = board.territoriesOwnedBy(playerId).size
  val continentsOwned = board.continentsOwnedBy(playerId)
  val territoryBonus = math.max(3, territoriesOwned / 3)
  val continentBonus = continentsOwned.map(_.bonusTroops).sum
  territoryBonus + continentBonus

def calculateInitialTroops(players: List[PlayerImpl], playerStates:
    List[PlayerState], board: Board): List[PlayerState] =
  //metodo che si occupa di determinare le truppe iniziali con cui ogni
      giocatore inizia la partita a seconda di quanti giocatori ci sono
```

Aspetti implementativi: questa è la classe che viene utilizzata in fase di piazzamento delle truppe: oltre a calcolare il numero di truppe da fornire ai giocatori all'inizio del turno, gestisce la logica dei tris giocabili (per ogni combinazione di carte valida, viene fornito un numero di truppe bonus variabile, da aggiungere a quelle già calcolate).
`BonusCalculator` è stato implementato come un oggetto singleton, che contiene solo metodi puri e statici, ossia stateless e deterministici. Inoltre si può notare una centralizzazione del calcolo delle truppe bonus, seppur in situazioni diverse nel corso della partita. Possiamo quindi affermare che sono vengono il SRP e il

principio DRY (assenza di ripetizioni nel codice).

Metodo di GameEngine.scala: `tradeCardsAction()`

```
private def tradeCardsAction(playerId: String, cardNames: Set[String],
    gameState: GameState, state: EngineState): EngineState =
  val playerState = gameState.getPlayerState(playerId).get
  val cardNameCounts =
      cardNames.groupBy(identity).view.mapValues(_.size).toMap
  val cards: Seq[TerritoryCard] = playerState.territoryCards
    .groupBy(_.territory.name)
    .flatMap { case (name, cards) =>
      val count = cardNameCounts.getOrElse(name, 0)
      cards.take(count)
    }.toSeq
  val bonus = BonusCalculator.calculateTradeBonus(cards)
  val updatedPlayerState = playerState
    .removeTerritoryCards(cards.toSet)
    .copy(bonusTroops = playerState.bonusTroops + bonus)
  val updatedGameState = gameState.updatePlayerState(playerId,
      updatedPlayerState)
  state.copy(gameState = updatedGameState)
```

Aspetti implementativi: questo metodo viene chiamato all'interno di `performActions()` nel momento in cui il giocatore decide di giocare un tris di carte. Per far ciò, deve osservare il `PlayerState` del giocatore in questione, estrarre da esso la lista di `TerritoryCard` che egli possiede ed estrarre le tre carte che sono state selezionate, per poi passarle al `BonusCalculator`, il quale calcola il bonus di truppe che il giocatore guadagna. Una volta terminato questo processo, aggiorna il `PlayerState`, andando a rimuovere le `TerritoryCard` giocate e aggiungere le truppe bonus ottenute.
Infine, è necessario aggiornare anche il `GameState`, per far sì che tutti i giocatori utilizzino la stessa versione e visualizzare l'azione appena effettuata.

6.2 Colì Diego

Per quanto riguarda l'implementazione, mi sono occupato principalmente dei moduli Core e Bot, garantendo la corretta comunicazione tra di essi e con il modulo Server.

6.2.1 Implementazione Core

Nel corso della costruzione di questo modulo, avvenuta in collaborazione con Costantini Marco, gli elementi attribuibili al sottoscritto sono, in particolare:
Files di `/model` e `/player`;
Files `GameEngine`, `TurnManager`, `DecksManager` e `BotController` di `/engine`.
Mi sono occupato, inoltre, della creazione del file `Board.json`, contenente i dati relativi ai continenti, ai territori e alle truppe bonus.

Esempio file di /engine: `TurnManager.scala`

```
trait TurnManager:
    def currentPlayer: Player
    def nextPlayer(): TurnManager
    def currentPhase: TurnPhase
    def isValidAction(action: GameAction, gameState: GameState,
        engineState: EngineState): Boolean

case class TurnManagerImpl(
    players: List[Player],
    currentPlayerIndex: Int = 0,
    phase: TurnPhase = TurnPhase.SetupPhase
) extends TurnManager:

    def currentPlayer: Player = players match
        case Nil => throw InvalidPlayerException()
        case _ => players(currentPlayerIndex)

    def nextPlayer(): TurnManager = players match
        case Nil => throw InvalidPlayerException()
        case _ => // rotazione turno dei giocatori
            val nextIndex = (currentPlayerIndex + 1) % players.size
            if (phase == TurnPhase.SetupPhase && nextIndex != 0)
                copy(currentPlayerIndex = nextIndex, phase =
                    TurnPhase.SetupPhase)
            else
                copy(currentPlayerIndex = nextIndex, phase =
                    TurnPhase.MainPhase)

    def currentPhase: TurnPhase = phase
```

```
def isValidAction(action: GameAction, gameState: GameState,
    engineState: EngineState): Boolean =

  def getPlayerStateOrThrow(id: String) = ...
  def getTerritoryOrThrow(name: String) = ...

  (action, phase) match
    case (GameAction.PlaceTroops(playerId, troops, territoryName),
        _) => // validazione piazzamento truppe

    case (GameAction.Reinforce(playerId, from, to, numTroops),
        TurnPhase.MainPhase) => // validazione spostamento truppe

    case (GameAction.Attack(attackerId, defenderId, from, to,
        numTroops), TurnPhase.MainPhase) => // validazione attacco

    case (GameAction.TradeCards(playerId, cardNames),
        TurnPhase.MainPhase) => // validazione scambio carte

    case (GameAction.EndTurn | GameAction.EndSetup, _)
      if phase == TurnPhase.MainPhase || phase ==
          TurnPhase.SetupPhase => // validazione fine turno/setup

    case _ => false
```

Aspetti implementativi: questo file definisce uno componenti principali per gestire la corretta rotazione dei turni in partita e per la validazione delle azioni di gioco. È stato utilizzato il pattern trait + implementation, rispettando l'immutabilità dei campi e generandone di nuovi ogni qualora ce ne fosse bisogno, grazie al metodo `copy()` fornito di default dalla case class. È stato inoltre largamente impiegato il pattern matching, il quale ci permette di discriminare le varie combinazioni GameAction-TurnPhase. Infatti, come si può notare da `isValidAction()`, la validazione di un'azione di gioco viene effettuata solo se la fase del turno corrisponde a quella richiesta. Ad esempio, l'azione Attack può essere effettuata esclusivamente in MainPhase. Il fatto che la validazione di tali azioni avvenga solo all'interno di questa classe, ci consente di rispettare il SRP, oltre al principio DRY, grazie all'utilizzo di metodi di comodo (Es: `getPlayerStateOrThrow()`), che evitano ripetizioni inutili.

6.2.2 Implementazione Bot e regole Prolog

Nel corso della costruzione di questo modulo, avvenuta in collaborazione con Arcese Gabriele, gli elementi attribuibili al sottoscritto sono, in particolare:
Files di `/bot`, `/prolog` e `/strategy`;
Theories Prolog di `/resources/theories`.

Esempio file di /bot: `BotPlayer.scala`

```
class BotPlayer(
    override val id: String,
    override val name: String,
    override val color: PlayerColor,
    val strategyRules: Set[StrategyRule]
) extends Player, Strategy:

  override def playerType: PlayerType = PlayerType.Bot
  private val isOffensive: Boolean = ...
  override def decideMove(gameState: GameState): GameAction = ...
```

Aspetti implementativi: questa classe definisce un giocatore di tipo Bot, estendendo i traits `Player` e `Strategy`. Il primo viene utilizzato per tener traccia dei dati di ciascun giocatore, indipendentemente dalla tipologia Human/Bot. Il secondo, invece, viene utilizzato per definire solo il metodo che consente di decidere la prossima azione di gioco. Questa scelta implementativa consente di separare la logica strategica automatizzata del Bot dal resto dell'implementazione di un qualsiasi giocatore, al fine di rispettare il SRP e garantire estendibilità.

Esempio files di /prolog: `PrologRule.scala` | `OffensiveBotAttackRule.scala`

```
trait PrologRule(val theoryName: String) extends StrategyRule:

  private val engine: PrologEngine = PrologEngine("/theories/" +
      theoryName + ".pl")

  override def evaluateAction(gameState: GameState, playerId: String):
      Set[RatedAction] =
    val (territoriesStr, neighborStr) = encodeGameState(gameState)
    ...
    val goal = s"${theoryName.toLowerCase}($territoriesStr,
        $neighborStr, '$phase', '$playerId', $actionString,
        $scoreString, $descString)"
    val solutions = engine.solveAll(goal, ...)
    val actions = solutions.map(...).toSet
    actions

  protected def encodeGameState(gameState: GameState): (String, String) =
    ...
    (territoriesStr, neighborStr)
```

```
  private def escapeName(name: String): String = name.replace("'", "\\'")

  protected def parseAction(gameState: GameState, actionTerm: Term,
      playerId: String): GameAction =

    if (actionTerm.toString.startsWith("place_troops"))
      ...
      GameAction.PlaceTroops(playerId, troops, territoryName)
    else if (actionTerm.toString.startsWith("reinforce"))
      ...
      GameAction.Reinforce(playerId, from, to, troops)
    else if (actionTerm.toString.startsWith("attack"))
      ...
      GameAction.Attack(playerId, defenderId, from, to, troops)
    else if (actionTerm.toString.startsWith("end_turn")) then
        GameAction.EndTurn
    else throw new InvalidActionException()

  private def extractArgs(term: Term): Array[String] = ...
```

```
class OffensiveBotAttackRule() extends PrologRule("OffensiveBotAttack")
// "OffensiveBotAttack" = nome della theory Prolog da risolvere
```

Aspetti implementativi: questo trait è solo uno degli elementi che compongono il pattern strategy ed il pattern trait + implementation + companion object, il quale è stato molto utile per suddividere le responsabilità (dunque per rispettare il SRP) e offrire modularità, testabilità e possibilità di riutilizzo, con diversi Bot che possono avere sia strategie esclusive che in comune.
Un esempio calzante è `BotSetupPlaceTroopsRule`, la quale viene attualmente condivisa da tutti i Bot (sia offensivi che difensivi), a differenza di strategie come `OffensiveBotAttackRule` che, invece, sono esclusive.
Il metodo utilitario `encodeGameState()` ci permette di semplificare significativamente le theories Prolog, dal momento che si occupa di convertire lo stato di gioco in liste di predicati `territory/3` e `neighbor/3`. Tali liste vengono utilizzate da `evaluateAction()` per "costruire" il goal da risolvere tramite `PrologEngine`.
L'intero flusso di dati che permette la valutazione di una theory Prolog risolta, la scelta dell'azione da eseguire e il passaggio di informazioni tra Bot e Core, è rappresentato schematicamente al paragrafo 4.1.4.

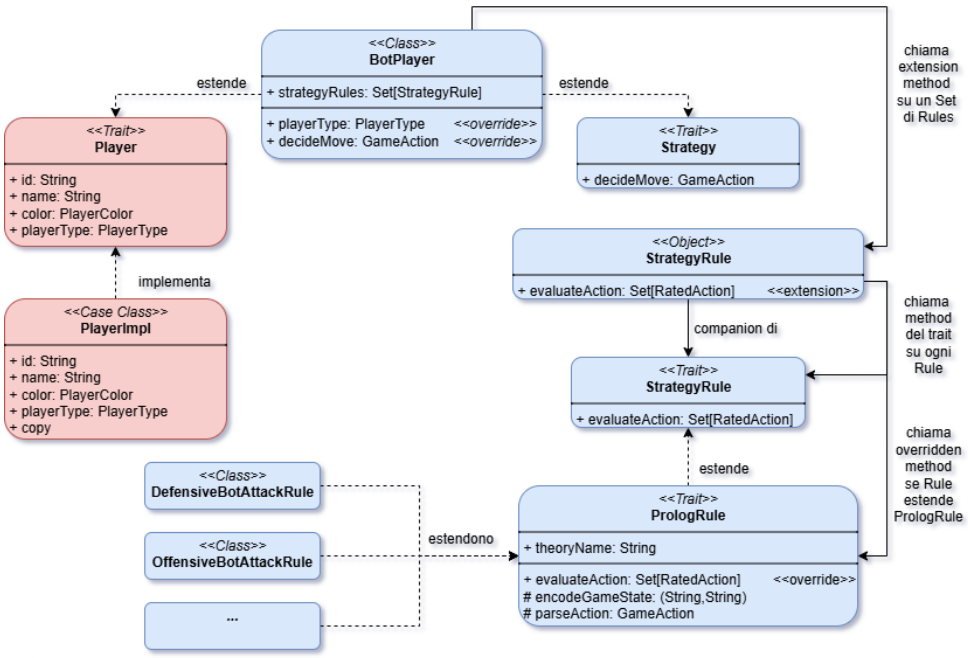

Figura 6: Diagramma Classi Bot-Player

Esempio theory Prolog: `OffensiveBotAttack.pl`

```
offensivebotattack(Territories, Neighbors, 'MainPhase', PlayerId,
    Action, Score, Description) :-
    member(territory(From, PlayerId, Troops), Territories),
    Troops > 5,
    member(neighbor(From, To, EnemyId), Neighbors),
    EnemyId \= PlayerId,
    EnemyId \= none,
    member(territory(To, EnemyId, DefTroops), Territories),
    TroopsToUse = 3,
    Action = attack(From, To, TroopsToUse),
    Score is Troops / (DefTroops + 0.1),
    Description = 'Attack enemy neighbor with strong territory'.
```

Logica: la theory in questione è stata progettata al fine di consentire al Bot (avente strategia offensiva) di avviare un attacco solo se tutte le condizioni all'interno di essa vengono rispettate, quindi se la sostituzione con tutti i sub-goals restituiscono un risultato positivo.

Le condizioni, dunque, che questa theory nello specifico deve soddisfare sono:

- La fase di gioco passata come argomento alla theory deve matchare *MainPhase*.
- Il territorio *From* deve far parte della lista *Territories*. Grazie al predicato `member/2` ci è possibile inoltre sfruttare per le condizioni successive i dati di tale territorio.
- Il territorio *From* deve possedere più di 5 *Troops*.
- Il territorio *To* deve confinare col territorio *From* e far parte della lista *Neighbors*.
- L'*EnemyId*, ossia l'id del proprietario del territorio *To*, deve essere diverso da 'none' e da *PlayerId*, ossia l'id del proprietario del territorio *From*.
- Controllo ulteriore: il territorio *To* deve far parte della lista *Territories*.

Se tutte queste condizioni vengono rispettate, la theory:

- Setta a 3 le *TroopsToUse* per l'attacco.
- Restituisce l'azione da effettuare (Attack con i corretti parametri)
- Restituisce un punteggio dinamico, che verrà valutato rispetto agli altri attacchi possibili.
- Restituisce una descrizione sommaria di ciò che avviene.

Ricordiamo che i predicati `territory/3` e `neighbor/3` sono stati resi disponibili dal metodo `encodeGameState()`.

6.2.3 Collegamento Bot-Core-Server

Dopo aver lavorato su `GameEngine` in **pair programming** con Constantini Marco, è stato possibile far comunicare i moduli sopra citati grazie alla collaborazione di Meco Daniel, al fine di garantire il corretto utilizzo delle azioni di gioco automatizzate in presenza di Bot.

Metodo di GameEngine.scala: `executeBotTurn()`

```
def executeBotTurn(): GameState =
  ...
  if currentPlayer.playerType != PlayerType.Bot || botController.isEmpty
      then throw new InvalidPlayerException()
  engineState.gameState.turnManager.currentPhase match
    case TurnPhase.SetupPhase => executeSetupTurn()
    case TurnPhase.MainPhase => executeMainTurn()
  engineState.gameState

private def executeSetupTurn(): Unit =
  ...
  if botTerritories.isEmpty then return

  while currentPlayerState.bonusTroops > 0 do
    try {
      val action = botController.get.nextAction(...)
      engineState = performActions(engineState, action)
      ...
    } catch {
      case e: Exception => ... // errore, performa direttamente EndSetup
        ...
        return
    }
  try { // termina setup se tutte bonusTroops piazzate
     engineState = performActions(engineState, GameAction.EndSetup)
  } catch {
    case e: Exception => ... // errore
  }

private def executeMainTurn(): Unit =
  ... // esegue in ordine: piazzamento, attacco, spostamento
  if playerState.bonusTroops > 0 then placeAllBonusTroops()
  executeAllAttacks()
  executeReinforceOrEnd()

private def placeAllBonusTroops(): Unit =
  ... // esegue piazzamento bonusTroops
  while currentPlayerState.bonusTroops > 0 do
    val action = botController.get.nextAction(...)
    engineState = performActions(engineState, action)
    ...
```

```
private def executeAllAttacks(): Unit =
  ... // esegue molteplici attacchi finche' possibile
  while canContinueAttacking && attackCount < maxAttacks do
    try {
      val action = botController.get.nextAction(...)
      action match
        case attack: GameAction.Attack =>
          engineState = performActions(engineState, action)
          ...
          updatedState.lastBattleResult match
            case Some(battleResult) =>
              battleResultCallback.foreach { callback =>
                callback(...) // passa dati a callback per notificare
                    agli altri giocatori i risultati della Battle
              }
            case None => canContinueAttacking = false
        case _ => canContinueAttacking = false
    } catch {
      case e: Exception => ... // errore
      }

private def executeReinforceOrEnd(): Unit =
  try {
    val action = botController.get.nextAction(...)
    action match
      // se possibile, spostamento e fine turno obbligatorio
      case _: GameAction.Reinforce =>
        engineState = performActions(engineState, action)
        engineState = performActions(engineState, GameAction.EndTurn)
      // se spostamento non possibile, fine turno
      case GameAction.EndTurn =>
        engineState = performActions(engineState, action)
      case _ =>
        engineState = performActions(engineState, GameAction.EndTurn)
  } catch {
      case e: Exception => ... // errore, performa direttamente EndTurn
        ...
  }
```

Aspetti implementativi: in questo metodo si è fatto ampio uso del pattern matching per filtrare quali azioni di gioco possono essere eseguite da un Bot, in base alla fase del turno in cui si trova. Nel caso SetupPhase, i giocatori possono esclusivamente posizionare le proprie truppe fino all'esaurimento di esse. Nel caso MainPhase, invece, questo metodo si assicura il `botController` restituisca, già valutati, tutti gli attacchi possibili e gli spostamenti di truppe. Vi è dunque una chiara suddivisione di responsabilità tra i vari metodi privati, al fine di evitare un unico blocco di codice, difficilmente leggibile ed estendibile. Vengono, inoltre, utilizzati gli opzionali per verificare l'esistenza di un risultato

di battaglia, per poter permettere al Bot di performare tutti gli attacchi possibili, fino a un numero massimo di azioni prestabilito. Per quanto riguarda lo spostamento truppe, invece, viene valutata la soluzione che ha ottenuto score migliore e quindi performata. A seguito dello spostamento, il passaggio del turno è obbligatorio. Per far si che questo metodo possa essere concretamente utilizzato durante lo svolgimento della partita, viene opportunamente chiamato in `GameSession.scala` (modulo Server).

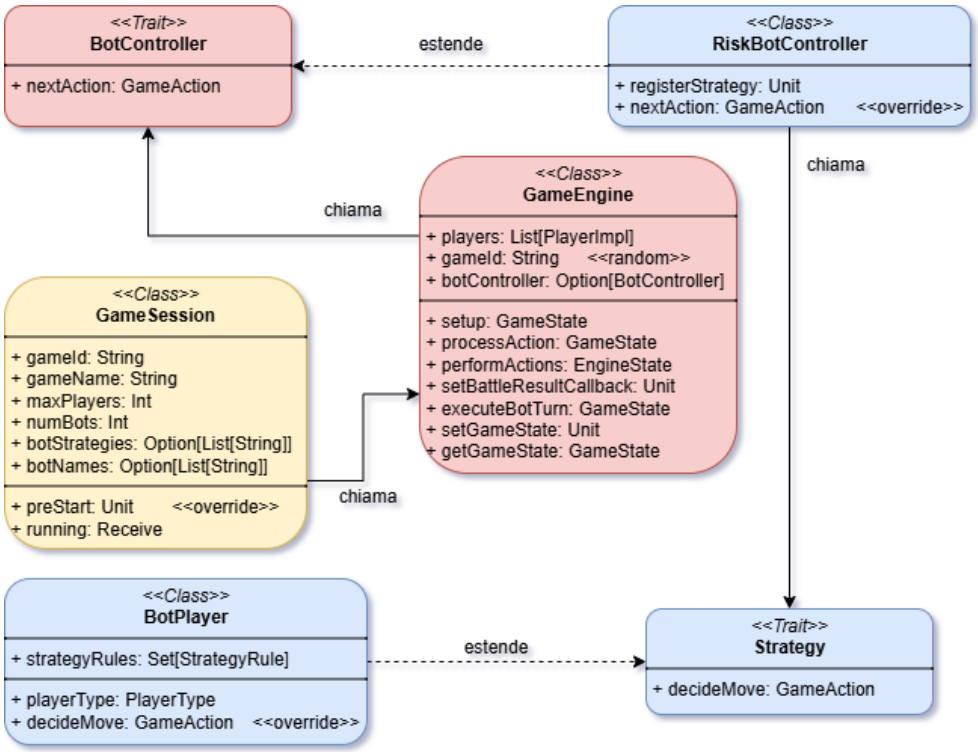

Figura 7: Diagramma Classi Bot-Core-Server

6.2.4 Interfaccia utente

Il lavoro che ho effettuato, inerente allo sviluppo dell'UI, è stato abbastanza limitato, essendomi occupato principalmente del lato backend del sistema (logica Core e Bot). Ho comunque contribuito creando l'immagine della mappa di gioco e la finestra `CreateBotDialog` per la creazione di Bot, assicurandomi che i dati passati al Server fossero coerenti e consistenti.

6.3 Costantini Marco

Per quanto riguarda l'implementazione, mi sono occupato principalmente dei moduli Core e Client.

6.3.1 Implementazione Core

Nel corso della costruzione di questo modulo, avvenuta in collaborazione con Colì Diego, gli elementi attribuibili al sottoscritto sono, in particolare:
Files di `/board`;
Files `Battle`, `GameEngine`, `CardsBuilder` e `ObjectiveValidator` di `/engine`.
Mi sono occupato, inoltre, del file `Objectives.json`, contenente gli obiettivi che i giocatori dovranno raggiungere.

Aspetti implementativi di: `Battle.scala`
Ho implementato la logica della battaglia tra territori, la quale è stata già descritta nei requisiti, gestendo il confronto tra attaccante e difensore, il calcolo delle perdite e la risoluzione dei turni di battaglia. L'intero flusso è modellato secondo i principi della programmazione funzionale.

```
enum BattleResult:
  case AttackerWins, DefenderWins, BattleContinues
```

Utilizzo di Algebraic Data Types (ADT), garantendo l'esaustività del pattern matching.

```
case class BattleState( ... )
case class BattleRoundResult( ... )
```

Utilizzo di Case Class immutabili con supporto a metodi come `copy()`, per aggiornare lo stato senza mutarlo.

```
def validateBattle(...): Either[String, Unit]
```

Utilizzo di `Either` per la gestione funzionale degli errori: `Either[String, Unit]` per validare condizioni prima della battaglia. Gli errori vengono rappresentati con valori (`Left`) e i successi con `Right`.

```
for
  _ <- validateBattle(...)
  ...
yield BattleRoundResult(...)
```

Utilizzo del costrutto monadico `for-yield` per rendere la sequenza di operazioni leggibile e gestibile.

```
battleRound(...)(using rollDice: Int => Seq[Int])
...
given defaultRollDice: (Int => Seq[Int]) = Dice.rollMany
```

Utilizzo di `using`: la funzione `battleRound()` prende come parametro implicito una funzione per tirare i dadi, seguendo il pattern dependency injection funzionale, mantenendo la funzione pura.

```
val result = ...
val (updatedAttacker, updatedDefender) = result match
  case AttackerWins => ...
  case DefenderWins => ...
  case BattleContinues => ...
```

Utilizzo di pattern matching su ADT per gestire tutti i possibili risultati della battaglia.

Aspetti implementativi di: `CardsBuilder.scala`
Ho sviluppato la classe responsabile della costruzione dei mazzi di carte, composti da `TerritoryCard`, le quali contengono un territorio e una figura (per il funzionamento dello scambio carte già descritto in precedenza), e da `ObjectiveCard`, le quali descrivono l'obiettivo del giocatore per permettere di vincere la partita. Questa classe lavora a partire dai file JSON che descrivono la mappa (`Board.json`) e gli obiettivi (`Objectives.json`).

```
private def createInitialTerritories(continentsJson: Seq[JsValue]):
    Map[String, Territory] =
  continentsJson.flatMap { continentJson =>
    (continentJson \ "territories").as[JsArray].value.map {
        territoryJson =>
      val name = (territoryJson \ "name").as[String]
      name -> Territory(name)
    }}.toMap
```

Utilizzo di `flatMap()` per la lista di territori e `map()` per trasformare i dati JSON in oggetti del dominio.

```
private def addNeighborsToTerritories(
  continentsJson: Seq[JsValue],
  territoriesMap: Map[String, Territory]
): Map[String, Territory] =
  continentsJson.foldLeft(territoriesMap) { (accMap, continentJson) =>
    (continentJson \ "territories").as[JsArray].value.foldLeft(accMap) {
        (mapAcc, territoryJson) =>
      val name = (territoryJson \ "name").as[String]
      val neighborsNames = (territoryJson \
          "neighbors").as[JsArray].value.map(_.as[String])
      val neighbors = neighborsNames.flatMap(n => mapAcc.get(n)).toSet
```

```
      val updatedTerritory = mapAcc(name).copy(neighbors = neighbors)
      mapAcc + (name -> updatedTerritory)
    }
  }
```

Utilizzo di `foldLeft()` per accumulare lo stato in modo funzionale e `copy()` per mantenere l'immutabilità degli oggetti.
Ho composto inoltre varie funzioni tra loro per costruire la logica di parsing e creazione degli oggetti.

Aspetti implementativi di: `GameEngine.scala`
Questo file è stato creato in collaborazione con Colì Diego, a seconda delle funzionalità che abbiamo sviluppato. L'engine rappresenta il cuore logico del gioco, dalla fase di setup iniziale fino alla conclusione con la vittoria di un giocatore. Incapsula lo stato del gioco e coordina l'interazione tra giocatori, Bot, carte, board, fasi di gioco e azioni. Le componenti principali sono:

- `EngineState`: contiene lo stato attuale del gioco (`GameState`) e tiene traccia di eventi particolari, ad esempio l'eventuale conquista di un territorio da parte di un giocatore durante il suo turno.
- `GameState`: rappresenta l'istantanea del gioco in un determinato momento e include: la mappa di territori/continenti (`Board`), gli stati dei giocatori (`PlayerState`) e il gestore di turni (`TurnManager`) e mazzi di carte (`DecksManager`).
- `GameEngine`: è la classe che elabora le azioni di gioco (`GameAction`) e aggiorna di conseguenza il `GameState`.

La mia responsabilità è stata quella della gestione della battaglia, descritta precedentemente. Nel metodo `attackAction()`, dopo la chiamata a `battleRound()` di `Battle`, il quale simula una battaglia tra un territorio attaccante e uno difensore, i territori coinvolti vengono aggiornati e, se l'attaccante ne conquista uno:

- `GameState` viene aggiornato tramite `updatedTerritory()`.
- A fine turno (metodo `endAction()`), il giocatore che ha conquistato il territorio viene ricompensato con una `TerritoryCard` attraverso un controllo di una variabile flag all'interno della case class `EngineState` che viene utilizzata in questo metodo attraverso una nuova istanza con il metodo **copy** (essendo la variabile flag all'interno di una case class, quindi immutabile).

```
private def attackAction(...): EngineState = {
    val attackerTerritoryOpt =
        gameState.getTerritoryByName(from).flatMap(t =>
        t.owner.map(owner => (owner, t)))
```

```
  val defenderTerritoryOpt =
      gameState.getTerritoryByName(to).flatMap(t => t.owner.map(owner
      => (owner, t)))

  (attackerTerritoryOpt, defenderTerritoryOpt) match {
    case (Some((attacker, attackerTerritory)), Some((defender,
        defenderTerritory))) =>
      Battle.battleRound(attacker, defender, attackerTerritory,
          defenderTerritory, troops) match
        case Left(error) => // errore
        case Right(battleResult) =>
          val updatedBoard = gameState.board
            .updatedTerritory(battleResult.attackerTerritory)
            .updatedTerritory(battleResult.defenderTerritory)
          val updatedGameState = gameState
            .updateBoard(updatedBoard)
            .updateLastBattleResult(battleResult)
          val conquered = battleResult.result ==
              BattleResult.AttackerWins
          state.copy(
            gameState = updatedGameState,
            territoryConqueredThisTurn =
                state.territoryConqueredThisTurn || conquered
          )
    case _ => // errore
  }
}
```

Al momento della creazione dell'engine da parte del sistema, in `setup()` vengono automaticamente generati i mazzi di carte e distribuite quest'ultime tra i vari giocatori. Infine, ho implementato la logica per il controllo della condizione di vittoria. Il metodo `checkVictoryCondition()` utilizza `ObjectiveValidator`, il quale analizza l'obiettivo e controlla che le condizioni siano soddisfatte rispetto all'attuale `GameState`.

Aspetti implementativi di: `ObjectiveValidator.scala`
Questa classe rappresenta un oggetto di utilità che ha il compito di verificare se un giocatore ha completato il proprio obiettivo, secondo la logica definita dalle `ObjectiveCard` e sulla base dell'attuale `(GameState)`.

```
def done(objective: Option[ObjectiveCard], gameState: GameState,
    playerState: PlayerState): Boolean =
  objective.exists(card => evaluateObjective(card, gameState,
      playerState))
```

Questa è di fatto l'unica funzione esposta. Essa riceve la carta obiettivo, lo stato globale del gioco, lo stato del giocatore e chiama il metodo privato `evaluateObjective()` per verificarne il completamento. Quest'ultimo utilizza un pattern matching su `ObjectiveCard` per distinguere tra i tre tipi di obiettivi:

- `ConquerTerritories`: possedere un certo numero di territori con un numero minimo di truppe.
- `ConquerContinents`: conquistare specifici continenti.
- `ConquerNContinents`: conquistare un numero arbitrario di continenti.

Nel flusso del gioco, ogni volta che l'engine termina un'azione potenzialmente decisiva (ad esempio una conquista), viene chiamato `checkWinCondition()` di `GameState`, che sfrutta `ObjectiveValidator` per determinare se un giocatore ha raggiunto l'obiettivo.
`ObjectiveValidator` ha il vantaggio di essere facilmente estendibile, infatti offre la possibilità di aggiungere nuovi tipi di obiettivi e gestirli con nuovi cases. Inoltre tutta la logica di validazione è centralizzata qui, garantendo `SRP`.
Il codice di `ObjectiveValidator` segue un approccio funzionale puro, coerente con i principi fondamentali della programmazione funzionale in Scala, ad esempio immutabilità, pattern matching su ADT e l'impiego di funzioni pure (`done()` ed `evaluateObjective()`, che non causano side-effects).

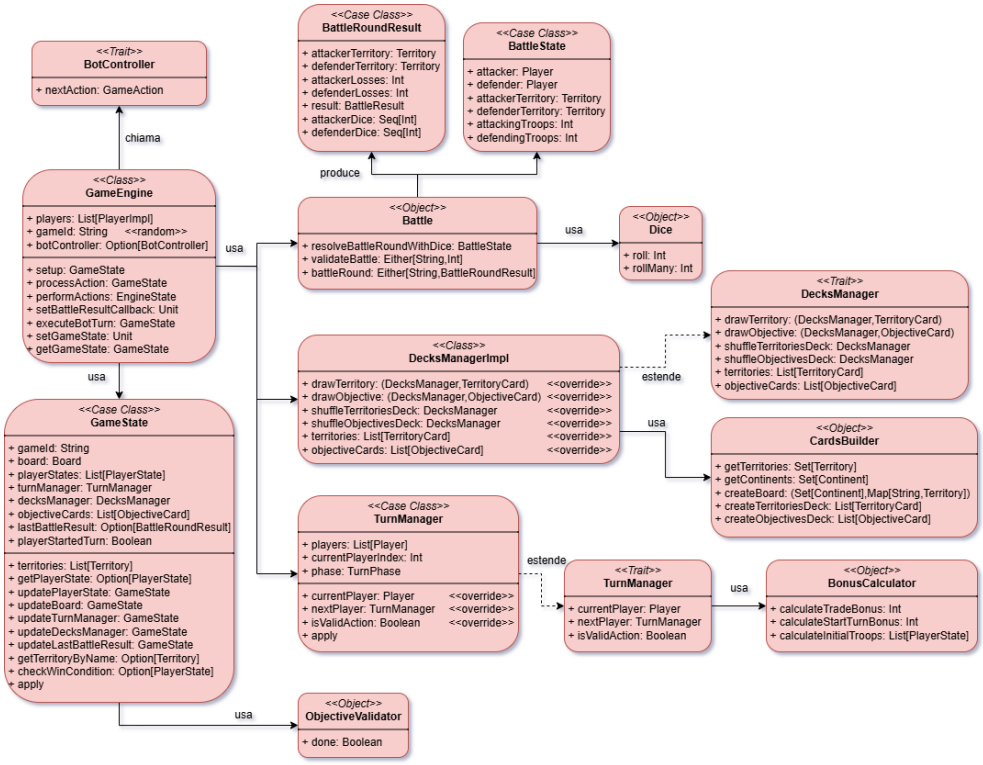

Figura 8: Diagramma Classi Core

Per completare la comprensione del diagramma, è importante evidenziare che quasi tutte le componenti utilizzano:

- Le enumerazioni `TurnPhase e GameAction`, le quali rappresentano rispettivamente la fase del turno di ogni giocatore e le azioni di gioco.
- Le classi di `/model/board`, `/model/cards` e `model/player`.

Queste classi fungono da modello del dominio e vengono usate in ogni parte della logica di gioco, garantendo consistenza, immutabilità e forte tipizzazione.

6.3.2 Scambio messaggi Client-Server-Core

In collaborazione con Gabriele Arcese e Daniel Meco che mi ha spiegato il modo in cui viene gestito lo scambio dei messaggi, ho gestito questo scambio tra client-server-core per la battaglia e per le carte disponibili per ogni giocatore, assicurando la corretta trasmissione e ricezione dei risultati.

Messaggi di battaglia

L'utente seleziona territori e truppe tramite la UI `(AttackDialog)`, viene creato un messaggio di attacco **(GameActionMessage con action "attack")** e inviato al server tramite il `GameActionHandler`.

```
// client/GameActionHandler.scala
def attack(gameId: String, fromTerritory: String, toTerritory: String,
    troops: Int, defenderId: String): Future[Boolean] = {
  val action = GameActionMessage(...)
  networkManager.sendMessage(action)
}
```

Il messaggio viene ricevuto dal server e convertito in `GameAction` e lo passa al `GameEngine`. Il core **(GameEngine)** esegue la logica della battaglia tramite la funzione **attackAction**. Il server invia il risultato della battaglia a tutti i client tramite un messaggio **BattleResult**. Il client aggiorna la UI (dadi, territori, alert) in base al risultato ricevuto.

Messaggi carte

Quando il client chiede di vedere le carte, il server invia lo stato aggiornato del giocatore, inclusa la lista delle carte **(territoryCards)**. Le carte sono serializzate in DTO **(TerritoryCardDto)** nel server e inviate nel messaggio di stato. Il client riceve la lista delle carte e le mostra tramite la UI **(ShowCardsDialog)**.

6.3.3 Interfaccia utente

Ho collaborato attivamente con i miei compagni allo sviluppo dell'interfaccia utente, contribuendo alla progettazione e realizzazione delle finestre di gioco, dei dialoghi di attacco, rinforzo e gestione carte, e all'integrazione con la logica di gioco e la comunicazione client-server. L'interfaccia è molto **intuitiva e reattiva**, facilitando l'interazione degli utenti con tutte le funzionalità principali del gioco.

6.4 Daniel Meco

L'implementazione del server per il gioco di Risiko è stata progettata per robusta e scalabile, capace di gestire più partite e giocatori contemporaneamente. Per raggiungere questi obiettivi, la tecnologia centrale scelta è stata il toolkit Akka, un framework open-source per la costruzione di applicazioni concorrenti e distribuite, che insieme al protocollo applicativo WebSocket, sono stati il pilastro per la gestione e il mantenimento dei giocatori e delle partite.

6.4.1 Implementazione Server

Il sistema si basa su tre pilastri fondamentali di Akka:

- Akka HTTP gestisce il server web e le comunicazioni esterne in modo asincrono, supportando molte connessioni contemporanee.
- Akka Streams permette la comunicazione in tempo reale via WebSocket, controllando il flusso dei dati per evitare sovraccarichi.
- Akka Actors è il cuore del sistema: ogni attore gestisce uno stato isolato e comunica tramite messaggi asincroni, semplificando la gestione della concorrenza.

Questo approccio semplifica enormemente la gestione della concorrenza e dello stato. Invece di usare complessi meccanismi di lock e sincronizzazione, lo stato di ogni partita e di ogni connessione è incapsulato all'interno di un attore, garantendo che le operazioni siano eseguite in modo sicuro e senza conflitti.

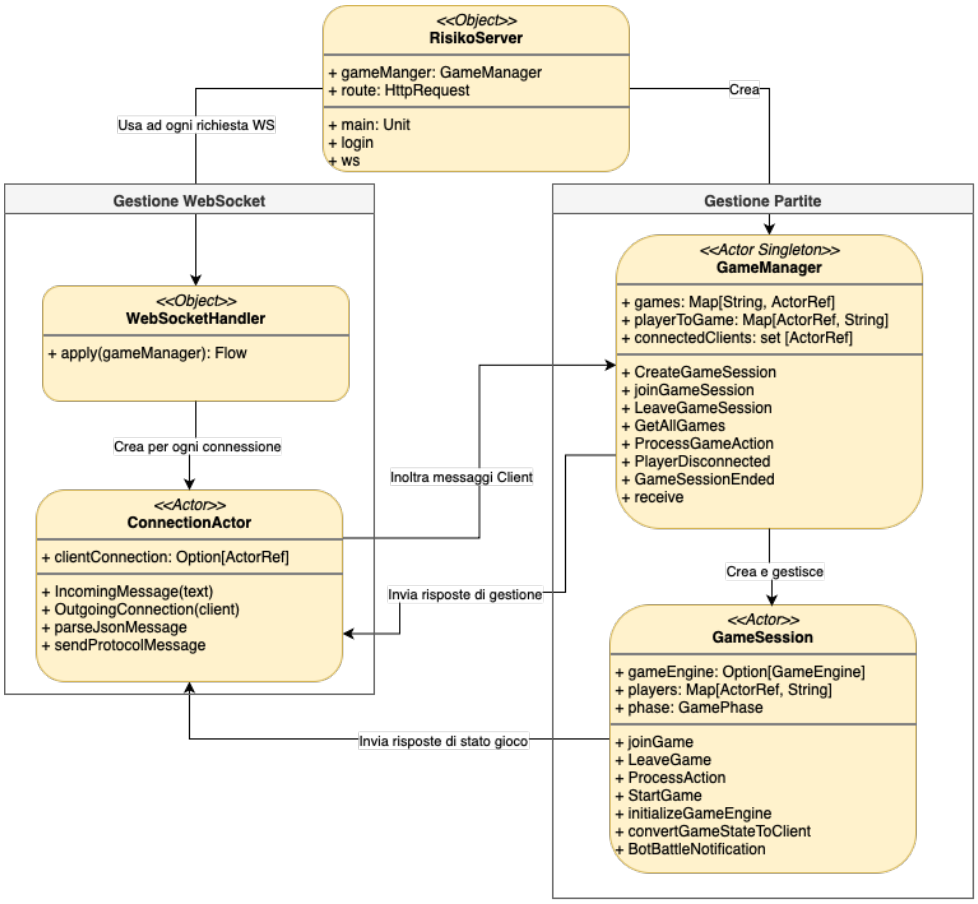

Figura 9: Diagramma Classi Server

6.4.2 Punto di ingresso nel server

Il file RisikoServer rappresenta il cuore dell'avvio dell'applicazione. Il suo compito primario è inizializzare il sistema, definire le interfacce di comunicazione e avviare il server. Esso espone le sue funzionalità attraverso una serie di endpoint HTTP, gestiti da Akka HTTP:

```
val route =
  concat(
    path("health") { ... },
    path("ws"){
      handleWebSocketMessages(WebSocketHandler(gameManager))
    },
    pathPrefix("api") { ... } )
```

All'interno della definizione delle rotte del server, la sezione path("ws") è dedicata a gestire le richieste che arrivano all'URL http://<host>:<port>/ws. Questo endpoint è il punto di ingresso per tutte le comunicazioni in tempo reale del gioco. Il metodo handleWebSocketMessages è una direttiva di Akka HTTP che gestisce la "promozione" di una connessione da HTTP a WebSocket.
Accetta come parametro un Flow di Akka Streams, che definisce come processare i messaggi in entrata e in uscita per quella specifica connessione.
La chiamata WebSocketHandler(gameManager) è un metodo factory che costruisce e restituisce questo Flow. Passa il gameManager a WebSocketHandler in modo che il Flow (e l'attore ConnectionActor che crea al suo interno) sappia a quale attore principale inoltrare i messaggi di gioco.

```
def apply(gameManager: ActorRef)(implicit system: ActorSystem):
    Flow[Message, Message, NotUsed] =

  val connectionId = java.util.UUID.randomUUID().toString.take(8)
  val handler = system.actorOf(Props(new
      ConnectionActor(gameManager)), s"connection-$connectionId")
  val incoming = Flow[Message]
    .collect {
      case TextMessage.Strict(text) => IncomingMessage(text)
    }
    .to(Sink.actorRef(
      ref = handler,
      onCompleteMessage = akka.actor.PoisonPill,
      onFailureMessage = { case _ => akka.actor.PoisonPill }
    ))
  val outgoing = Source.actorRef(
    PartialFunction.empty,
    PartialFunction.empty,
    16,
    OverflowStrategy.dropHead
  )
  Flow.fromSinkAndSourceMat(incoming, outgoing) { (_, outActor) =>
    handler ! OutgoingConnection(outActor)
    NotUsed
  }
```

Nel WebSocketHandler, il punto di ingresso è il metodo apply. Questo metodo agisce come una factory: viene invocato una volta per ogni nuova connessione WebSocket stabilita e ha il compito di costruire il Flow di Akka Streams. Tale Flow modella la pipeline di comunicazione bidirezionale per quel singolo client.

```
private class ConnectionActor(gameManager: ActorRef) extends Actor:
  // Stato interno dell'attore
  var clientConnection: Option[ActorRef] = None
  private var pingScheduler: Option[akka.actor.Cancellable] = None
```

```
// Gestione del ciclo di vita dell'attore
override def preStart(): Unit = {
  // Avvia un timer per inviare ping periodici
  // ... implementazione ...
}

override def postStop(): Unit = {
  // Cancella il timer e notifica il GameManager della disconnessione
  // ... implementazione ...
}

// Gestore principale dei messaggi
def receive: Receive = {
  // Mantiene viva la connessione
  case SendPing => // ... invia un messaggio di Ping al client

  // Memorizza il canale di uscita per rispondere al client
  case OutgoingConnection(client) => clientConnection = Some(client)

  // Gestisce i messaggi in arrivo dal client
  case IncomingMessage(text) =>
    parseJsonMessage(text) match {
      case Success(msg) => handleClientMessage(msg) // Inoltra alla
          logica
      case Failure(ex) => // ... gestisce errore di parsing
    }

  // Gestisce i messaggi di risposta dal sistema di gioco da inviare
      al client
  case msg: ProtocolMessage => // ... serializza e invia al client

  case _ => // ... gestisce messaggi non riconosciuti
}

// Inoltra le richieste del client al GameManager
private def handleClientMessage(msg: ProtocolMessage): Unit =
  msg match {
    case protocol.ClientMessages.CreateGame(...) =>
      gameManager ! GameManager.CreateGameSession(...)

    case protocol.ClientMessages.JoinGame(...) =>
      gameManager ! GameManager.JoinGameSession(...)

    case action: protocol.ClientMessages.GameAction =>
      gameManager ! GameManager.ProcessGameAction(...)

    case protocol.ClientMessages.LeaveGame(...) =>
      gameManager ! GameManager.LeaveGameSession(...)

    case _ => // ... gestisce altri tipi di messaggi client
```

```
    }

    private def parseJsonMessage(text: String): Try[ProtocolMessage] =
        Try {
      // ... implementazione della conversione da JSON a oggetto Message
    }
//
```

La classe privata ConnectionActor è il vero motore logico della comunicazione per un singolo client. Le sue responsabilità sono:

- gestire autonomamente il proprio ciclo di vita. Al suo avvio (preStart), pianifica l'invio periodico di messaggi di Ping per mantenere attiva la connessione e prevenire timeout. Alla sua terminazione (postStop), che avviene quando il client si disconnette, l'attore notifica al GameManager l'evento di disconnessione (PlayerDisconnected)
- agire da traduttore e router. Quando riceve un messaggio testuale dal client (IncomingMessage), lo deserializza da JSON a una case class del protocollo di comunicazione. Successivamente, invece di implementare la logica di gioco, inoltra la richiesta all'attore GameManager sotto forma di comando interno. Ad esempio, un ClientMessages.CreateGame viene trasformato in un GameManager.CreateGameSession. Questo disaccoppiamento è un punto di forza dell'architettura: il ConnectionActor si occupa esclusivamente della comunicazione e della traduzione, mentre la logica orchestrativa è delegata al GameManager.
- inoltrare le risposte ricevute dal GameManager o da una GameSession (ad esempio, un aggiornamento dello stato del gioco o il risultato di una battaglia), lo serializza in formato JSON e lo invia al client attraverso il canale di uscita che ha memorizzato.

6.4.3 Gestione delle partite

Il GameManager è un attore Akka che agisce come orchestratore centrale per tutte le sessioni di gioco. In quanto attore "singleton" (ne esiste una sola istanza per l'intero server), ha la responsabilità esclusiva di gestire il ciclo di vita delle partite, l'associazione dei giocatori alle sessioni e l'instradamento delle azioni di gioco. Non conosce le regole specifiche di Risiko, ma sa chi sta giocando, dove sta giocando e come instradare le comunicazioni.

L'implementazione del GameManager sfrutta un potente pattern di Akka per la gestione dello stato: la macchina a stati finiti (FSM) implementata tramite il metodo context.become. L'attore non mantiene lo stato in variabili mutabili (var), ma lo passa come argomento a un metodo che rappresenta lo stato corrente. Questo approccio promuove l'immutabilità e rende la gestione della concorrenza più sicura e prevedibile.
Lo stato dell'attore è definito dalla firma del suo metodo principale, running:

```
def running(
    games: Map[String, ActorRef],      // Mappa: ID Partita -> Attore
        GameSession
    connectedClients: Set[ActorRef],   // Set di tutti i client connessi
    playerToGame: Map[ActorRef, String], // Mappa: Attore Client -> ID
        Partita
    gameNameMap: Map[String, String]   // Mappa: ID Partita -> Nome
        Partita
): Receive =
    // ... gestione dei messaggi ...
```

Ogni volta che lo stato deve cambiare (ad esempio, quando una nuova partita viene creata), l'attore invoca context.become con una nuova chiamata a running, passando le mappe e i set aggiornati. Quindi, il GameManager gestisce un insieme definito di comandi che riceve dai ConnectionActor. Analizziamo i più significativi.

```
case CreateGameSession(gameName, maxPlayers, creator, ...) =>
    val gameId = UUID.randomUUID().toString().take(6)
    val gameSession = context.actorOf(
        GameSession.props(gameId, ...),
        s"GameSession-$gameId"
    )
    val updatedGames = games + (gameId -> gameSession)
    // ... aggiorna le altre mappe ...
    creator ! ServerMessages.GameCreated(gameId, gameName, ...)
    gameSession ! GameSession.JoinGame(...)
    context.become(running(updatedGames, ...))
```

Quando un giocatore richiede di creare una nuova partita, il GameManager genera un ID univoco per la nuova partita. Crea un nuovo attore GameSession come figlio, delegandogli la gestione della logica di quella specifica partita. Aggiorna il suo stato interno, aggiungendo la nuova partita alla mappa games. Invia un messaggio di conferma (GameCreated) al creatore e gli fa eseguire l'unione alla sessione appena creata.

```
case ProcessGameAction(gameId, playerId, action) =>
          games.get(gameId) match
              case Some(gameSession) =>
                  log.info(s"Forwarding game action '${action.action}'
                      from player $playerId to game $gameId")
                  gameSession ! GameSession.ProcessAction(playerId,
                      action)
              case None =>
                  log.warning(s"Game action processing failed: Game
                      session $gameId not found")
                  sender() ! ServerMessages.Error(s"Game session $gameId
                      not found")
```

Durante il gioco, tutte le azioni del client (es. attaccare, spostare armate) vengono inviate al GameManager incapsulate in un messaggio ProcessGameAction. Il GameManager agisce da puro router: estrae il gameId dal messaggio, cerca l'attore GameSession corrispondente e gli inoltra l'azione. Questo disaccoppia completamente la logica di rete dalla logica di gioco.

Quindi, il GameManager è un componente architetturale cruciale che funge da controllore del traffico e gestore di stato a livello globale, permettendo a più sessioni di gioco di coesistere in modo isolato e coordinato all'interno dello stesso sistema.

6.4.4 Protocollo di comunicazione

Il package "Protocollo e Messaggi" nel diagramma UML sotto raffigurato, rappresenta il cuore del dialogo tra il client e il server. Questo componente è fondamentale perché definisce un vocabolario comune e strutturato per tutte le interazioni. Al vertice della gerarchia si trova il trait Message, che agisce come un contratto base, garantendo che ogni comunicazione sia un tipo di messaggio riconosciuto dal sistema. I messaggi sono poi logicamente suddivisi in due namespace distinti:

- ClientMessages: Raggruppa tutte le azioni e le richieste che un client può inviare al server, come CreateGame o GameAction.
- ServerMessages: Contiene tutte le risposte, notifiche e aggiornamenti di stato che il server può inviare ai client, come GameCreated, GameState o BattleResult.

Questa struttura, implementata tramite case class in Scala, organizza il codice in modo pulito. Gli attori Akka utilizzano questi oggetti messaggio immutabili per comunicare in modo sicuro e asincrono. Inoltre, il componente JsonSupport si basa su queste definizioni per serializzare i messaggi in formato JSON per la trasmissione sulla rete e per deserializzarli al loro arrivo, agendo da ponte tra la logica interna del server e il mondo esterno.

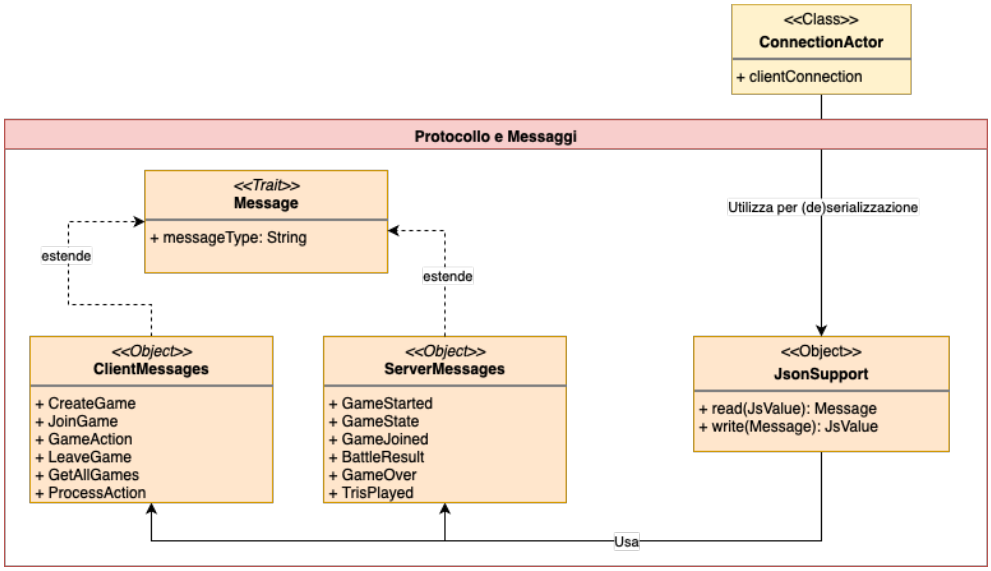

Figura 10: Diagramma Protocollo di comunicazione

6.4.5 Gestione del gioco

L'attore GameSession è il componente architetturale che incapsula la logica e lo stato di una singola partita di Risiko. Ogni istanza di GameSession è un mondo a sé stante, responsabile della gestione dei giocatori partecipanti, delle fasi di gioco e dell'interazione con il motore di gioco (GameEngine). Creato dal GameManager su richiesta di un utente, questo attore vive per la durata di una partita e viene terminato alla sua conclusione.

Similmente al GameManager, anche GameSession adotta un approccio funzionale per gestire la sua evoluzione, utilizzando context.become. Lo stato della sessione non è mantenuto in variabili mutabili, ma è incapsulato nei parametri del metodo running, che definisce il comportamento dell'attore.

```
def running(
    players: Map[String, ActorRef], //Mappa: ID Giocatore -> Attore
        Client
    playerData: Map[String, Player], // Dati anagrafici dei giocatori
    phase: GamePhase,             // Fase corrente del gioco
    currentPlayer: Option[String], // ID del giocatore di turno
    gameState: Map[String, Any]   //Stato del gioco da inviare ai client
): Receive =
    //gestione dei messaggi in base allo stato corrente
```

Le fasi principali del ciclo di vita di una partita sono modellate come stati distinti:

- WaitingForPlayers: La partita è stata creata ma non ha ancora raggiunto il numero di giocatori necessario per iniziare.
- Setup: La partita è al completo e si sta inizializzando il tabellone e distribuendo le risorse iniziali.
- Playing: La partita è in corso e i giocatori eseguono i loro turni.
- Finished: La partita è terminata con la vittoria di un giocatore.

L'attore GameSession gestisce un insieme di comandi che riceve principalmente dal GameManager o da se stesso per orchestrare le transizioni di stato.
Nella fase WaitingForPlayers, l'attore accetta nuovi giocatori.
Quando un giocatore si unisce, l'attore lo aggiunge alla mappa players. Controlla se il numero massimo di giocatori è stato raggiunto. In tal caso, avvia la transizione verso la fase di gioco.
Inizializza il motore di gioco attraverso il metodo initializeGameEngine. Questo metodo inizializza il GameEngine, agendo da ponte tra la GameSession e la logica pura del gioco. Traduce i giocatori nel formato richiesto dal motore, istanzia i bot con le loro strategie e crea l'oggetto GameEngine. Infine, imposta una callback per permettere al motore, un componente non-attore, di notificare l'attore su eventi specifici come le battaglie dei bot.

```
private def initializeGameEngine(players: List[Player]): Unit = {
  //1.Traduce i giocatori umani dal formato della sessione
      (GameSession.Player) a quello del motore di gioco
      (CorePlayerImpl)
  val humanPlayers = players.filter(!_.id.startsWith("bot-")).map {
      player =>
    CorePlayerImpl(
      player.id,
      player.username,
      generatePlayerColor(player.id),
      model.player.PlayerType.Human
    )
  }

  var botController: Option[bot.RiskBotController] = None

  // 2.Crea i giocatori bot in base ai parametri della partita
      (numero e strategie).
  val botPlayers =
    if (numBots > 0) {
      val existingBots = players.filter(_.id.startsWith("bot-"))
      val botStrategiesList =
          this.botStrategies.getOrElse(List.fill(numBots)("Offensivo"))

      //Per ogni bot, crea l'istanza usando la BotFactory in base
          alla strategia specificata.
      existingBots.zipWithIndex.map { case (existingBot, index) =>
        val botColor = generatePlayerColor(existingBot.id)
        val botStrategy = if (index < botStrategiesList.length)
            botStrategiesList(index) else "Offensivo"

        //La BotFactory restituisce sia il giocatore che il suo
            controller.
        val (botPlayer, controller) = botStrategy match {
          case "Offensivo" =>
              bot.BotFactory.createAggressiveBot(existingBot.id,
              existingBot.username, botColor)
          case "Difensivo" =>
              bot.BotFactory.createDefensiveBot(existingBot.id,
              existingBot.username, botColor)
          case _ =>
              bot.BotFactory.createAggressiveBot(existingBot.id,
              existingBot.username, botColor)
        }
        botController = Some(controller)

        //Traduce anche il bot nel formato del motore di gioco.
        CorePlayerImpl(
```

```
            existingBot.id,
            existingBot.username,
            botColor,
            model.player.PlayerType.Bot
          )
        }.toList
      } else {
        List.empty
      }

    //3.Unisce le liste di giocatori umani e bot.
    val allPlayers = humanPlayers ++ botPlayers

    try {
      // 4. Istanzia il GameEngine, passando la lista completa dei
          giocatori e il controller dei bot.
      gameEngine = Some(new GameEngine(allPlayers, gameId,
          botController))

      // 5.Imposta una callback per permettere al motore di notificare
          la sessione di eventi specifici.

      gameEngine.foreach { engine =>
        engine.setBattleResultCallback { (from, to, attackerId,
            defenderId, battleResult) =>
          //Quando il motore invoca la callback, l'attore invia un
              messaggio a se stesso.
          self ! BotBattleNotification(from, to, attackerId,
              defenderId, battleResult)
        }
      }
    } catch {
      case ex: Exception =>
        log.error(s"Error initializing game engine: ${ex.getMessage}")
        gameEngine = None
    }
  }
```

Dopodichè invia un messaggio StartGame a se stesso per procedere con l'avvio effettivo. Notifica ai client che il setup è iniziato.

```
case JoinGame(playerId, playerRef, username, ...) =>
   // ... logica per aggiungere il giocatore ...
   val newPhase = (finalPlayers.size >= maxPlayers, phase) match
       case(true, WaitingForPlayers) =>
          log.info(s"Game $gameId has reached max players, starting
              setup")
          initializeGameEngine(...)
          self ! StartGame
```

```
          Setup
        case _ => phase
    context.become(running(..., newPhase, ...))
```

Nella fase di Playing, l'attore agisce come un controllore che media tra le richieste del client e il motore di gioco.

- Riceve un'azione dal GameManager (originata da un ConnectionActor).
- Esegue controlli (validazione) preliminari (es. il giocatore è nella partita? La partita è nella fase corretta?).
- Traduce l'azione dal formato del protocollo client (ClientMessages.GameAction) a un'azione comprensibile dal motore di gioco (engine.GameAction) tramite il metodo convertToGameAction.
- Inoltra l'azione al GameEngine con engine.processAction(coreAction). Il motore, che contiene le regole di Risiko, elabora l'azione e restituisce il nuovo stato del gioco.
- Converte il nuovo stato del gioco in un formato DTO (Data Transfer Object) e lo trasmette a tutti i giocatori nella partita, garantendo che le loro interfacce siano sincronizzate.

```
case ProcessAction(playerId, action) =>
   // ... controlli di validazione ...
   case (Some(_), Playing, Some(engine)) =>
       try {
         val coreAction = convertToGameAction(action, playerId)
         val updatedGameState = engine.processAction(coreAction)

         //logica per notificare risultati specifici (es. battaglia)

         val clientState = convertGameStateToClient(updatedGameState)
         // Invia il nuovo stato a tutti i giocatori umani
         humanPlayers.values.foreach(_ ! ServerMessages.GameState(...))

         //logica per gestire il turno del prossimo giocatore (se e' un
             bot) ...
       } catch {
          case e: exceptions.GameOverException =>
          case ex: Exception =>
       }
```

Una caratteristica sofisticata della GameSession è la sua capacità di gestire i turni dei bot. Quando il turno passa a un giocatore controllato dal computer: Il metodo executeBotTurnsUntilHuman viene invocato. Questo metodo esegue in modo sequenziale i turni dei bot (engine.executeBotTurn()) fino a quando il turno non passa a un giocatore umano.

```
private def executeBotTurnsUntilHuman(
      engine: GameEngine,
      currentBotId: String,
      humanPlayers: Map[String, ActorRef],
      playersList: List[String]
  ): (GameState, String) = {
    println(s"[DEBUG] Esecuzione turno bot $currentBotId")
    try {
      val botGameState = engine.executeBotTurn()
      val nextPlayerId = botGameState.turnManager.currentPlayer.id
      println(s"[DEBUG] Bot $currentBotId ha completato il turno.
          Prossimo giocatore: $nextPlayerId")
      val intermediateClientState =
          convertGameStateToClient(botGameState)
      humanPlayers.values.foreach(player =>
        player ! ServerMessages.GameState(
          gameId,
          playersList,
          nextPlayerId,
          scala.collection.immutable.Map("gameStateDto" ->
              intermediateClientState)
        )
      )
      Thread.sleep(2000)
      val isNextPlayerBot =
          botGameState.turnManager.currentPlayer.playerType ==
          model.player.PlayerType.Bot
      if (isNextPlayerBot) then return
          executeBotTurnsUntilHuman(engine, nextPlayerId,
          humanPlayers, playersList)
      else return (botGameState, nextPlayerId)
    } catch {
      case e: exceptions.GameOverException =>
        println(s"[DEBUG] Game over durante il turno del bot
            $currentBotId: ${e.getMessage}")
        throw e
      case ex: Exception =>
        log.error(s"Errore durante l'esecuzione del turno del bot
            $currentBotId: ${ex.getMessage}")
        ex.printStackTrace()
        return (engine.getGameState,
            engine.getGameState.turnManager.currentPlayer.id)
    }
  }
```

Il suo funzionamento è ricorsivo, in particolare, esegue il turno completo di un singolo bot chiamando engine.executeBotTurn(). Dopo il turno, invia lo stato aggiornato del gioco a tutti i client umani. Questo permette ai giocatori umani

di osservare le azioni del bot in modo comprensibile. Verifica chi è il prossimo giocatore. Se anche il prossimo giocatore è un bot, il metodo richiama se stesso, passando il nuovo stato del gioco e l'ID del bot successivo. Se il prossimo giocatore è umano, la ricorsione termina e il metodo restituisce il controllo, lasciando il gioco in attesa dell'azione del giocatore umano.

Un principio architetturale chiave adottato all'interno della GameSession è la netta separazione tra il modello di dati interno del motore di gioco (engine.GameState) e il modello di dati inviato ai client. Il modello interno è complesso, ricco di oggetti e relazioni, e ottimizzato per l'esecuzione delle regole di gioco. Al contrario, il client necessita di una rappresentazione più semplice e serializzabile in JSON. Questa separazione è implementata attraverso il pattern Data Transfer Object come accennato prima. I DTO sono classi semplici, spesso case class, il cui unico scopo è trasferire dati. L'attore GameSession si fa carico di questa traduzione, agendo da intermediario tra il motore e i client.

```
case class TerritoryDto(name: String, owner: String, troops: String)
   case class TerritoryCardDto(id: String, territoryName: String,
       cardType: String)
   case class MissionCardDto(id: String, description: String,
       targetType: String, targetValue: String)
   case class PlayerStateDto(
     playerId: String,
     cards: String,
     bonusTroops: String,
     territoryCards: List[TerritoryCardDto] = List(),
     missionCard: Option[MissionCardDto] = None
   )
   case class GameStateDto(
     gameId: String,
     currentPlayer: String,
     currentPhase: String,
     territories: List[TerritoryDto],
     playerStates: List[PlayerStateDto],
     playerStartedTurn: String
   )
```

Il metodo convertGameStateToClient è il cuore di questo processo di traduzione. Ogni volta che un'azione viene elaborata e lo stato del gioco cambia, questo metodo viene invocato per creare una "fotografia" dello stato attuale, epurata e semplificata per la trasmissione.

```
private def convertGameStateToClient(gameState: engine.GameState):
    GameStateDto = {
  GameStateDto(
    gameId = gameState.gameId,
    currentPlayer = gameState.turnManager.currentPlayer.id,
```

```
    currentPhase = gameState.turnManager.currentPhase match {
      case TurnPhase.SetupPhase => "SetupPhase"
      case TurnPhase.MainPhase => "MainPhase"
    },
    territories = gameState.board.territories.map { territory =>
      TerritoryDto(
        name = territory.name,
        owner = territory.owner.map(_.id).getOrElse(""),
        troops = territory.troops.toString
      )
    }.toList,
    playerStates = gameState.playerStates.map { playerState =>
      PlayerStateDto(
        playerId = playerState.playerId,
        cards = playerState.territoryCards.size.toString,
        bonusTroops = playerState.bonusTroops.toString,
        territoryCards = // ... conversione delle carte ...
        missionCard = // ... conversione della missione ...
      )
    }.toList,
    playerStartedTurn = gameState.playerStartedTurn.toString
  )
}
```

In modo speculare, quando un'azione arriva dal client, deve essere tradotta dal formato del protocollo di comunicazione a un comando che il motore di gioco possa comprendere. Questo è il compito del metodo convertToGameAction.

```
private def convertToGameAction(
    clientAction: ClientMessages.GameAction,
    playerId: String
): engine.GameAction = {
  clientAction.action match {
    case "attack" =>
      //estrae i parametri dalla mappa...
      engine.GameAction.Attack(...)

    case "place_troops" =>
      //estrae i parametri ...
      engine.GameAction.PlaceTroops(...)

    //altri casi ...
  }
}
```

Questo metodo riceve un ClientMessages.GameAction, che contiene il tipo di azione come stringa e i parametri in una mappa generica Map[String, String]. Lo analizza e costruisce l'oggetto engine.GameAction corrispondente, fortemente tipizzato, che il GameEngine si aspetta di ricevere.

Inoltre il GameSession gestisce l'abbandono e la fine della partita (LeaveGame, GameOverException)
Se un giocatore abbandona, viene rimosso dalle mappe. Se la partita rimane vuota, l'attore notifica al GameManager la sua terminazione (GameSessionEnded) e si arresta.
Il GameEngine segnala la fine della partita lanciando una GameOverException. La GameSession la intercetta, notifica a tutti i giocatori il vincitore, passa allo stato Finished e comunica la sua terminazione al GameManager.

6.4.6 Integrazione col Client

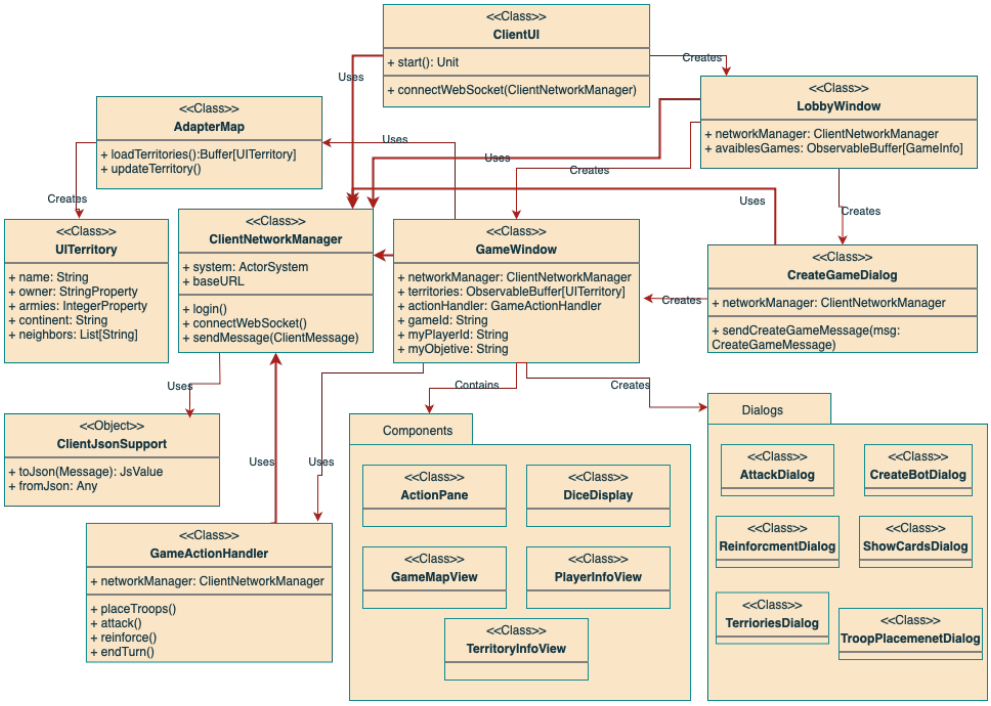

Figura 11: Diagramma Classi Client

L'applicazione client di Risiko è progettata per essere un'interfaccia utente reattiva e disaccoppiata dalla logica di rete. L'architettura si basa su una chiara separazione delle responsabilità tra l'interfaccia grafica (UI), la gestione della comunicazione di rete e la logica di invio delle azioni di gioco. Questa separazione è orchestrata principalmente da tre componenti chiave: ClientUI (e le sue finestre figlie), il ClientNetworkManager e il GameActionHandler.
L'interazione tra client e server segue un ciclo di vita ben definito, che inizia con una richiesta HTTP standard e transita verso una comunicazione persistente tramite WebSocket.
La prima operazione non avviene tramite WebSocket, ma con una richiesta POST standard all'endpoint /api/login. Questo approccio è convenzionale e sicuro per lo scambio iniziale di dati. Il ClientNetworkManager gestisce questa fase.

```
def login(username: String): Future[Boolean] =
  val request = HttpRequest(
    method = HttpMethods.POST,
    uri = s"$baseUrl/api/login",
    entity = HttpEntity(ContentTypes.‘application/json‘,
        s"""{"username": "$username"}""")
```

```
  )

  Http().singleRequest(request).flatMap { response =>
    response.status match
      case StatusCodes.OK =>
        Unmarshal(response.entity).to[String].map { jsonString =>
          // ... parsing della risposta e salvataggio del playerId ...
          true
        }
      case _ => Future.successful(false)
  }.recover { /* ... gestione errori ... */ }
```

Solo dopo un login iniziale andato a buon fine, il client tenta di stabilire la connessione WebSocket persistente. Il ClientNetworkManager utilizza Akka HTTP e Akka Streams per creare un Flow bidirezionale che gestirà tutti i messaggi futuri tra client e server.

```
def connectWebSocket(): Future[Boolean] = {
  // ...
  val incomingSink = Sink.foreach[Message] { message =>
    // Logica per gestire i messaggi in arrivo dal server
  }

  val (queue, outgoingSource) = Source
    .queue[String](100, OverflowStrategy.dropHead)
    // ...

  val webSocketFlow = Flow.fromSinkAndSource(incomingSink,
      outgoingSource)

  val (upgradeResponse, _) = Http().singleWebSocketRequest(
    WebSocketRequest(uri = wsUrl),
    webSocketFlow
  )

  upgradeResponse.map { upgrade =>
    if (upgrade.response.status == StatusCodes.SwitchingProtocols) {
      // Connessione riuscita
      true
    } else {
      // Connessione fallita
      false
    }
  }
}
```

Questa classe è il fulcro di tutte le operazioni di rete. Altre sue responsabilità sono:

- Fornisce un metodo sendMessage che prende un oggetto Scala, lo serializza

in JSON e lo accoda per l'invio tramite WebSocket.

- Ascolta i messaggi in arrivo sul Sink del WebSocket. Per ogni messaggio, lo deserializza e invoca la callback appropriata.
- Mantiene una mappa di messageCallbacks. L'interfaccia utente può registrare una funzione da eseguire quando arriva un messaggio di un certo tipo (es. gameState, gameJoined).

```
//Mappa delle callback registrate
private var messageCallbacks: Map[String, Any => Unit] = Map.empty

// Metodo per la registrazione
def registerCallback(messageType: String, callback: Any => Unit): Unit =
    {
  messageCallbacks = messageCallbacks + (messageType -> callback)
}

// Esempio di utilizzo nel gestore dei messaggi in arrivo
// ... nel Sink.foreach
val parsedMessage = ClientJsonSupport.fromJson(text)
parsedMessage match {
  case msg: GameState =>
    // ...
    messageCallbacks.get("gameState").foreach(_(msg))
}
```

L'oggetto ClientJsonSupport è il componente responsabile della traduzione bidirezionale tra gli oggetti case class di Scala e le stringhe JSON. Per gestire i messaggi in arrivo dal server, utilizza un approccio a "dispatch table" basato su una mappa chiamata messageHandlers.

```
/**
 * Map of message type handlers for deserializing incoming messages.
 * Each handler takes a map of JSON fields and returns the appropriate
     message object.
 */
private val messageHandlers: Map[String, Map[String, JsValue] => Any] =
    Map(
  "gameJoined" -> { fields =>
    GameJoinedMessage(
      extractString(fields, "gameId"),
      extractStringList(fields, "players"),
      extractString(fields, "gameName"),
      extractOption(fields, "playerColors")(_.convertTo[Map[String,
          String]])
    )
  },
```

```
  "gameState" -> { fields =>
    val gameId = extractString(fields, "gameId")
    val players = extractStringList(fields, "players")
    val currentPlayer = extractString(fields, "currentPlayer")

    val stateJson = fields.getOrElse("state", JsObject.empty).asJsObject
    val gameStateData = stateJson
        .fields("gameStateDto")
        .convertTo[GameStateData] // Usa un formato personalizzato

    GameState(gameId, players, currentPlayer, gameStateData)
  },

  "battleResult" -> { fields =>
    BattleResultMessage(
      extractString(fields, "gameId"),
        ...
    )
  },

  "error" -> { fields =>
    ErrorMessage(extractString(fields, "message", "Errore sconosciuto"))
  }
  // e cosi via per tutti gli altri tipi di messaggio
)
```

Per mantenere l'interfaccia utente pulita, la logica per costruire i messaggi di azione di gioco è incapsulata nel GameActionHandler. Questa classe fornisce metodi chiari come attack, placeTroops, endTurn, che prendono i parametri necessari, costruiscono l'oggetto GameActionMessage e lo passano al ClientNetworkManager per l'invio.

```
class GameActionHandler(networkManager: ClientNetworkManager)(implicit
    ec: ExecutionContext) {
  def attack(
    gameId: String,
    fromTerritory: String,
    toTerritory: String,
    troops: Int,
    defenderId: String
  ): Future[Boolean] = {
    val action = GameActionMessage(
      gameId = gameId,
      action = "attack",
      parameters = Map(...)
    )
    networkManager.sendMessage(action)
  }//altri metodi per le azioni ...
}
```

7 Testing

Aspetti implementativi dei files in: `/core/test`
A seconda della parte sviluppata nel Core, gli sviluppatori Costantini Marco e Colì Diego si sono occupati di implementare i vari test per verificare il corretto comportamento delle singole componenti. Si è fatto uso dei seguenti costrutti e librerie: `AnyFunSuite` (per scrivere test come unit test classici), `Matchers` (per asserzioni espressive, come `shouldBe`), `Either` (per gestire facilmente `Left/Right` e validare errori attesi). Il tutto al fine di garantire affidabilità della logica, prevenzione di regressioni (eventuali modifiche al codice possono essere validate velocemente tramite i test) e semplicità di refactor. Di seguito si riporta un esempio di codice.
Esempio file di /test: `BattleTest.scala`

```
test("Defender should lose troops if he loses the battle"):
  val resultEither = Battle.battleRound(...)
  resultEither match
    case Right(result)
      if (result.result == BattleResult.AttackerWins) =>
        result.defenderTerritory.owner should contain (attacker)
        result.attackerTerritory.troops shouldBe
            (attackerTerritory.troops - 3)
        result.defenderTerritory.troops shouldBe 3
    case Right(_) => succeed
    case Left(err) => fail(s"Unexpected error: $err")
```

Esempio file di /test: `DecksManagerTest.scala`

```
val emptyDecksManager = DecksManagerImpl(List(), List())
test("DecksManager should throw an exception if territories deck is
    empty"):
   assertThrows[NoTerritoriesCardsException]:
       emptyDecksManager.drawTerritory()

test("DecksManager should keep the same amount of cards after
    shuffling"):
   val newManager1 = tDecksManager.shuffleTerritoriesDeck()
   val newManager2 = oDecksManager.shuffleObjectivesDeck()
   newManager1.territoriesCards.size should be (2)
   newManager2.objectiveCards.size should be (4)
```

Inoltre, abbiamo configurato un sistema di **Continuous Integration (CI)** tramite **GitHub Actions**, il quale esegue autonomamente la suite di test ad ogni push. Nonostante non sia stato seguita tassativamente la metodologia **TDD (Test Driven Development)**, si è sempre riuscito a mantenere sotto controllo il corretto funzionamento del codice, testando ogni modifica significativa.

8 Retrospettiva

Durante lo sviluppo del progetto, il team ha adottato una metodologia di lavoro ispirata ai principi agili, organizzando le attività in sprint regolari e pianificando ciclicamente le priorità in base alla complessità delle componenti da implementare. Ogni sprint ha visto la realizzazione incrementale di funzionalità testabili, mantenendo un ritmo costante di avanzamento e integrazione tra i componenti. L'approccio modulare e test-driven adottato nella fase di sviluppo del Core si è dimostrato particolarmente efficace nel garantire robustezza e affidabilità della logica di gioco. Tuttavia, l'ultimo sprint ha subito un rallentamento imprevisto a causa dell'assenza per malattia di un membro del gruppo, il quale era responsabile di alcune parti fondamentali del modulo Core. Questo ha temporaneamente influito sull'avanzamento complessivo e sulla capacità del team di completare in tempo tutte le funzionalità previste. Nonostante la difficoltà, il gruppo ha reagito con spirito collaborativo, suddividendo il carico rimanente e concentrandosi sulle priorità più critiche per consegnare comunque una versione stabile e coerente del sistema. In generale, il lavoro di squadra, la comunicazione costante e la buona organizzazione delle responsabilità hanno permesso al progetto di evolvere in modo efficace, dimostrando la capacità del team di adattarsi ai cambiamenti e di affrontare gli imprevisti con flessibilità.

TASKS				SPRINT 1									
Descrizione	**Membro team**	**Modulo**	**Effort**										
Analisi requisiti	Team		**3**	3	2	1	0	0	0	0	0	0	0
Creazione struttura moduli	Team		**2**	2	2	2	2	1	1	0	0	0	0
Creazione componenti */model /player /card /utils*	Coli Costantini	Core	**3**	3	2	2	1	1	1	0	0	0	0
Impl. base *Battle* e *ObjectiveValidator*	Costantini	Core	**4**	4	3	3	2	2	1	1	0	0	0
Creazione componenti */bot /prolog /strategy*	Coli Arcese	Bot	**4**	4	4	4	3	2	2	2	1	0	0
Impl. base *RisikoServer*	Meco	Server	**2**	2	2	1	1	0	0	0	0	0	0
Impl. base *WebSocketHandler*	Meco	Server	**4**	4	4	3	3	2	1	1	1	1	0
Test connessione via terminale	Meco	Client	**2**	2	1	0	0	0	0	0	0	0	0

Figura 12: Sprint 1

TASKS				SPRINT 2									
Descrizione	**Membro team**	**Modulo**	**Effort**										
Impl. base classi */engine*	Coli Costantini	Core	**4**	4	4	3	3	2	2	1	1	0	0
Refactor *Battle*	Costantini	Core	**3**	3	2	2	1	0	0	0	0	0	0
Creazione *Objectives.json*	Costantini	Core	**1**	1	0	0	0	0	0	0	0	0	0
Creazione *Board.json*	Coli	Core	**2**	2	1	1	0	0	0	0	0	0	0
Creazione test componenti	Coli	Core	**3**	3	3	2	2	2	1	1	0	0	0
Refactor *ObjectiveValidator*	Costantini	Core	**3**	3	3	2	1	0	0	0	0	0	0
Impl. base GUI	Meco	Client	**3**	3	2	2	1	1	1	0	0	0	0
Impl. base *GameManager* e *GameSession*	Meco	Server	**5**	5	5	4	4	3	3	2	2	1	0
Refactor *WebSocketHandler*	Meco	Server	**2**	2	1	1	0	0	0	0	0	0	0
Supporto messaggi json	Meco	Server	**4**	4	4	4	4	4	3	3	2	1	0
Integrazione **Client**	Meco	Server	**3**	3	2	2	1	1	0	0	0	0	0

Figura 13: Sprint 2

TASKS				SPRINT 3									
Descrizione	**Membro team**	**Modulo**	**Effort**										
Miglioramento *GameEngine* e *Battle*	Coli Costantini	Core	**5**	5	4	4	3	3	3	2	2	1	0
Riformulazione fasi di gioco	Arcese Costantini	Core	**4**	4	3	3	2	2	1	1	0	0	0
Miglioramento validazione azioni in *TurnManager*	Coli	Core	**3**	3	3	2	2	1	0	0	0	0	0
Aggiornamento test	Coli	Core	**3**	3	2	2	1	1	1	0	0	0	0
Bridge con **Client**	Meco	Core	**2**	2	2	1	0	0	0	0	0	0	0
Miglioramento *GameWindow* e GUI	Meco Arcese	Client	**4**	4	4	3	3	2	2	1	0	0	0
Creazione protocolli scambio messaggi con **Server**	Meco	Client	**3**	3	3	2	2	1	1	0	0	0	0
Gestione eventi generati da **Client**	Meco	Server	**4**	4	4	3	3	2	2	1	1	0	0
Miglioramento interpretazione regole	Coli Arcese	Bot	**4**	4	4	3	2	2	1	1	0	0	0
Creazione theories Prolog	Coli	Bot	**4**	4	3	3	2	2	1	0	0	0	0
Dialog di creazione **Bot**	Coli	Client	**2**	2	1	0	0	0	0	0	0	0	0

Figura 14: Sprint 3

TASKS				SPRINT 4									
Descrizione	**Membro team**	**Modulo**	**Effort**										
Gestione turno e azioni **Bot**	Coli	Core	**4**	4	3	3	2	2	1	0	0	0	0
Refactor BonusCalculator e tradeCards()	Arcese	Core	**3**	3	2	2	1	0	0	0	0	0	0
Refactor *ObjectiveValidator*, *Battle* e *CardsBuilder*	Costantini	Core	**5**	5	4	4	3	3	2	1	0	0	0
Aggiornamento *scala.yml* per test	Meco		**1**	1	0	0	0	0	0	0	0	0	0
Integrazione **Core**	Meco	Server	**5**	5	4	4	3	3	2	1	1	0	0
Integrazione **Bot**	Meco Coli	Server	**4**	4	4	3	3	2	2	1	1	0	0
Miglioramento integrazione **Client**	Meco	Server	**3**	3	2	1	0	0	0	0	0	0	0
Gestione eventi generati da **Core**	Meco	Server	**3**	3	2	1	0	0	0	0	0	0	0
Miglioramento parsing json	Meco	Client	**2**	2	1	1	0	0	0	0	0	0	0
Miglioramento theories Prolog	Coli	Bot	**4**	4	3	3	2	1	0	0	0	0	0

Figura 15: Sprint 4