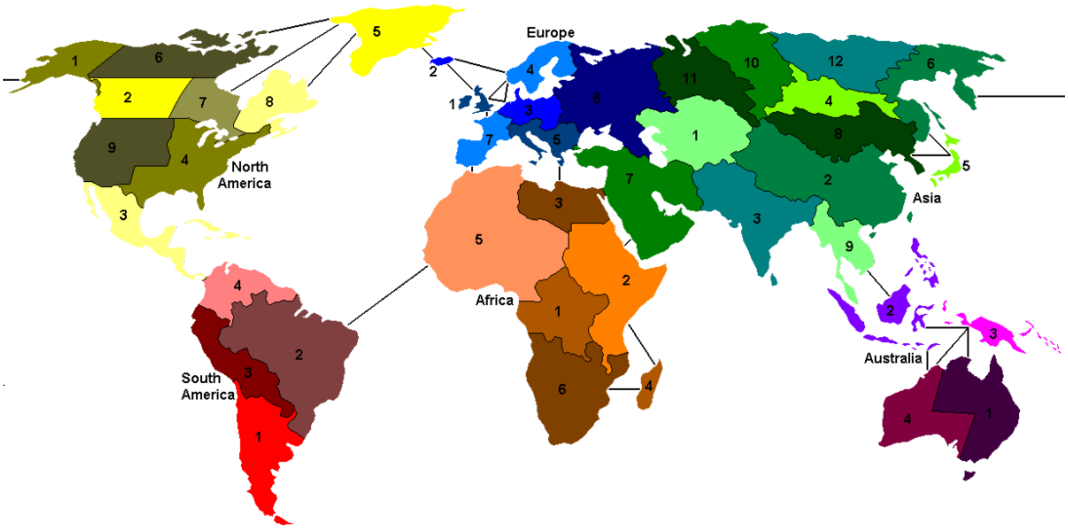
Software Engineering - Progetto Risiko
MVCGuru

Andrea Malnati
Roberto Negro
Daniele Giorgio Michele Romano
Lorenzo Radaelli
Lorenzo Persico

February 22, 2024



Contents

1 Prefazione

In questo documento presentiamo un'esplorazione approfondita del progetto DRisk, un'evoluzione digitale e altamente personalizzabile del classico gioco da tavolo Risk. Il nostro obiettivo è stato quello di superare le limitazioni della versione originale introducendo una maggiore flessibilità nella configurazione delle mappe e nella complessità del gioco, mantenendo al contempo l'essenza strategica del gioco originale

Il processo di sviluppo di DRisk è stato caratterizzato da un approccio iterativo, con lo scopo di affinare e adattare continuamente il progetto per rispondere meglio alle esigenze dei giocatori e alle possibilità offerte dalle tecnologie web moderne. Attraverso successive iterazioni, abbiamo progressivamente ampliato e migliorato le funzionalità del gioco, a partire da una solida base di gioco di base fino all'introduzione di elementi complessi come le opzioni di gioco scalabili.

Oltre a descrivere le fasi principali dello sviluppo, questo documento tocca anche il refactoring critico eseguito per integrare nuovi requisiti e migliorare l'architettura del software. Tale processo ha non solo facilitato l'aggiunta di nuove caratteristiche ma ha anche migliorato la manutenibilità e l'espandibilità del codice.

2 Introduzione

Il nostro approccio allo sviluppo del software è stato guidato dai principi del Unified Process, che abbiamo adottato per strutturare il lavoro in due iterazioni principali. La prima iterazione si è concentrata sull'analisi dei requisiti e sulla progettazione preliminare, coinvolgendo tutti i membri del team nella definizione delle specifiche e nella creazione dei diagrammi necessari a delineare l'architettura del sistema. La seconda iterazione ha spostato il focus sullo sviluppo del codice, mantenendo al contempo i diagrammi aggiornati e implementando una fase di testing e analisi del codice per assicurare la qualità e la stabilità dell'applicazione.

Nonostante la suddivisione dei compiti non sia stata rigidamente definita, il lavoro di squadra ha permesso di coprire efficacemente tutti gli aspetti del progetto. La fase di analisi e la parte backend hanno visto il contributo di tutti i membri del gruppo, mentre lo sviluppo del frontend è stato affidato principalmente a due componenti del team. Abbiamo inoltre prestato particolare attenzione all'utilizzo di design e architectural pattern, seguendo i principi SOLID e PHAME, che hanno guidato la nostra pratica di refactoring per ottenere un codice pulito e robusto.

3 Analisi

3.1 Glossario delle Difficoltà di Gioco

Difficoltà Facile: Questa modalità prevede una mappa ridotta del 66% rispetto alla versione originale, arrivando a un totale di 16 territori. È ideale per partite rapide da 2 a 4 giocatori. In questa variante, i giocatori condividono un obiettivo comune; il primo che lo raggiunge vince la partita, aggiungendo un elemento competitivo ma unificato al gioco.

Difficoltà Media: La mappa per la difficoltà media è ridotta del 33% rispetto alla versione originale, offrendo un'esperienza di gioco su 28 territori. Questa configurazione è adatta per partite da 2 a 6 giocatori e fornisce un punto d'incontro tra la rapidità della difficoltà facile e la complessità della difficoltà difficile.

Difficoltà Difficile: In questa modalità, la mappa rimane invariata con 42 territori, consentendo il gioco completo da 2 a 6 giocatori. È la configurazione standard per chi cerca l'esperienza di gioco più classica.

3.2 Requisiti Extra Richiesti dal Progetto

Di seguito, descriviamo come abbiamo interpretato e implementato le richieste aggiuntive per il progetto DRisk:

- **Scalabilità automatica delle mappe**: Abbiamo implementato un sistema che permette alle mappe di essere adattate automaticamente in base alla complessità del gioco scelta (facile, medio, difficile) e al numero di giocatori.
- **Gestione Avanzata degli Obiettivi di Gioco**: In alternativa alle carte obiettivo standard di Risiko, abbiamo implementato la possibilità di un obiettivo comune per tutti i giocatori, come il controllo di due terzi del mondo, per partite con un approccio strategico differente; questo è disponibile in modalità facile.
- **Integrazione con Piattaforme Web**: All'avvio della applicazione partirà un server in locale, a cui sarà possibile connettersi tramite browser per giocare.

3.3 Documenti Prodotti

Caso D'Uso UC1: Gioca Turno

Attore primario: Giocatore

Table 1: Scenario principale

Passo	Azione
1	Il sistema calcola e assegna al giocatore le nuove truppe basandosi sul numero di territori posseduti diviso 3, più eventuali bonus per il controllo dei continenti.
2	Il giocatore può scegliere di utilizzare le carte Risk per ricevere ulteriori truppe prima di posizionarle.
3	Il giocatore posiziona le nuove truppe nei propri territori.
4	Il giocatore decide se attaccare e segue il processo di attacco che include la selezione del territorio, il lancio dei dadi, e l'aggiornamento del numero di truppe.
(4)	*Questi passi si ripetono finché il giocatore desidera continuare ad attaccare.*
5	Il giocatore decide se fare uno spostamento di truppe tra i propri territori al termine degli attacchi.
6	Se il giocatore ha conquistato almeno un territorio, il sistema assegna una carta Risk.
7	Il sistema aggiorna lo stato del gioco e passa al turno successivo.

Table 2: Scenari alternativi

Scenario	Azione
3.1	Il giocatore utilizza le carte Risk in modo non valido: il sistema visualizza un messaggio di errore e richiede una nuova selezione di carte.
4.1	Il giocatore sceglie di non attaccare dopo il posizionamento delle truppe: procede direttamente al passaggio 5.
4.2	Dopo un attacco riuscito e la conquista di un territorio, il sistema richiede di specificare il numero di truppe da spostare nel territorio appena conquistato. Il numero minimo di truppe è uguale al numero dei dadi lanciati durante l'attacco, mentre il numero massimo è pari al numero di truppe presenti nel territorio di partenza meno uno.
6.1	Il giocatore termina il turno senza aver attaccato: il sistema non assegna una carta Risk al giocatore.

3.4 Diagrammi Prodotti

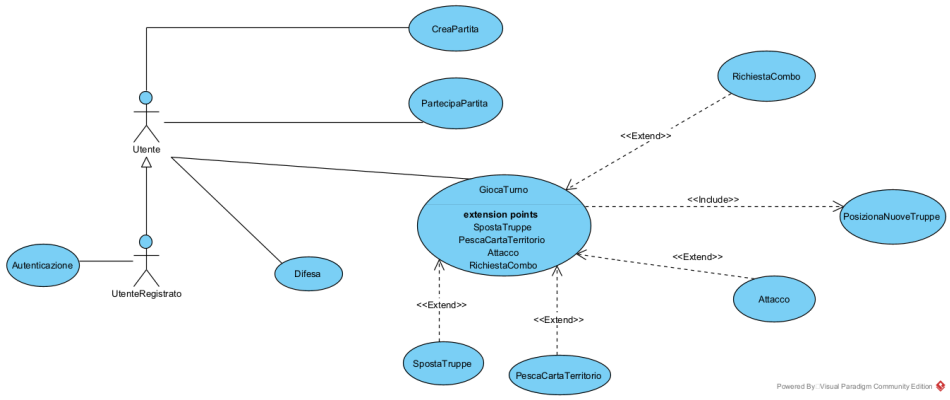


Figure 1: Diagramma dei casi d'uso

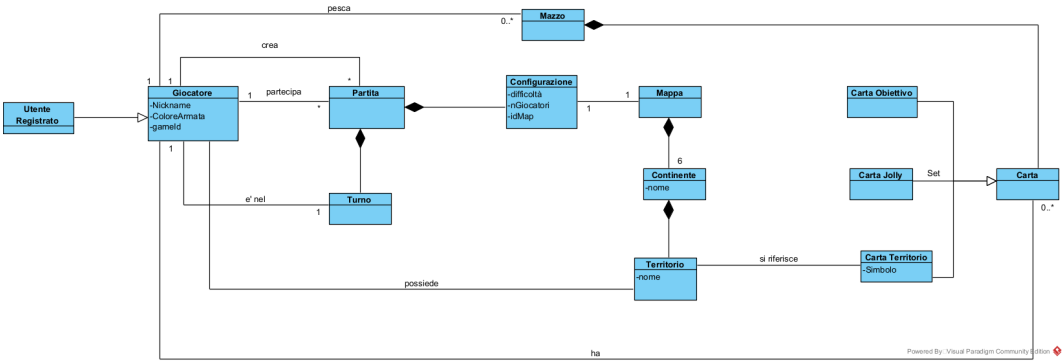


Figure 2: Modello di dominio

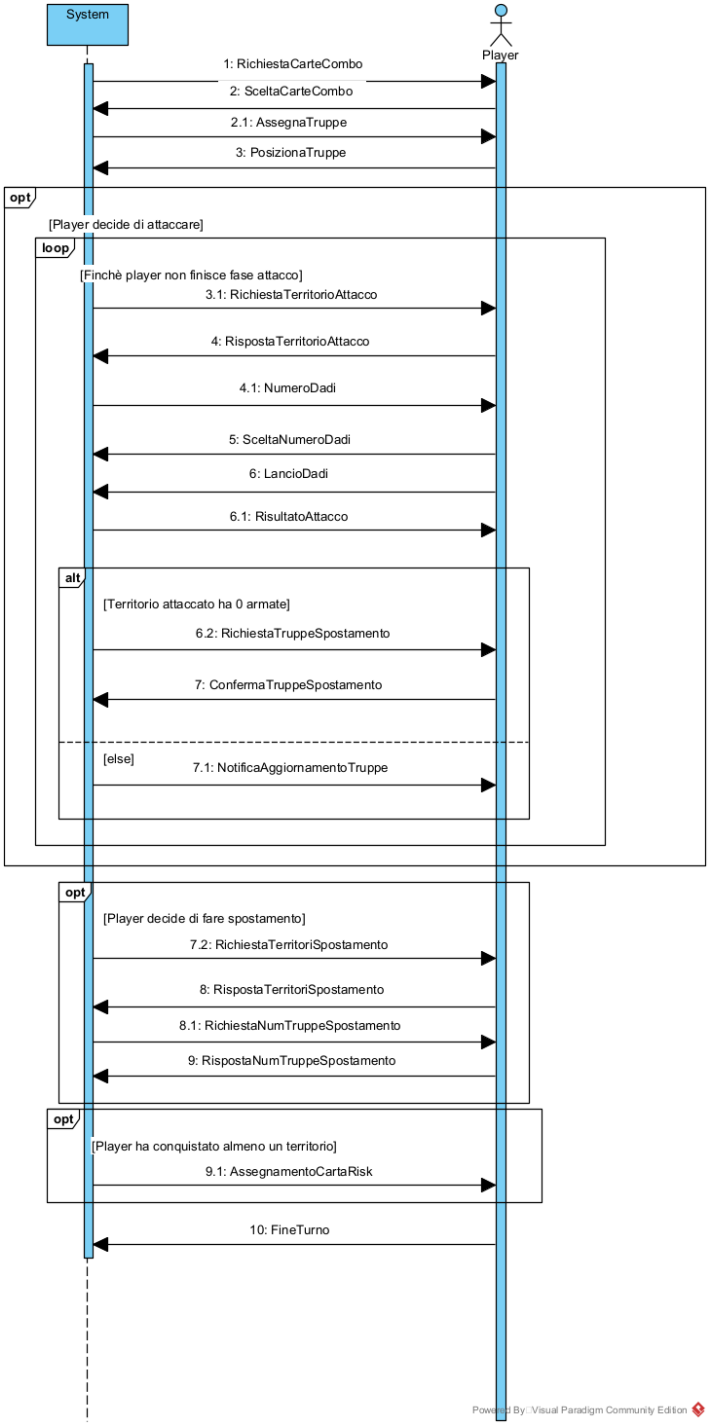


Figure 3: SSD Turno utente

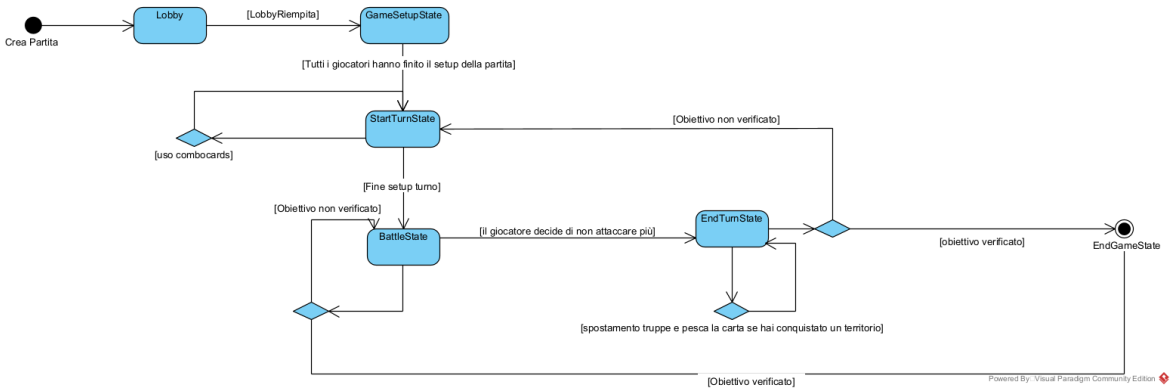


Figure 4: Diagramma stati - esecuzione partita

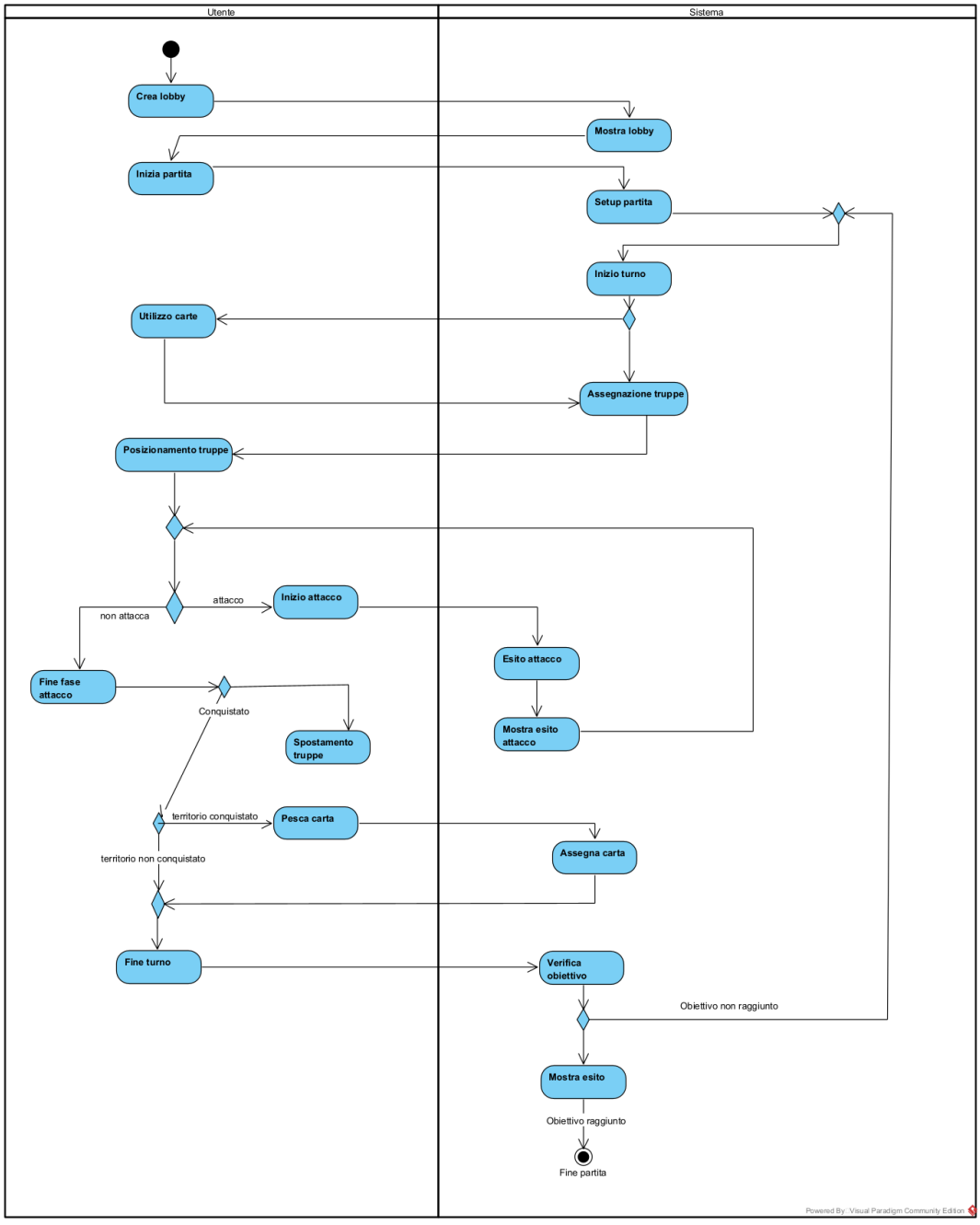


Figure 5: Diagramma delle attività - esecuzione partita

4 Progettazione

4.1 Diagrammi Prodotti

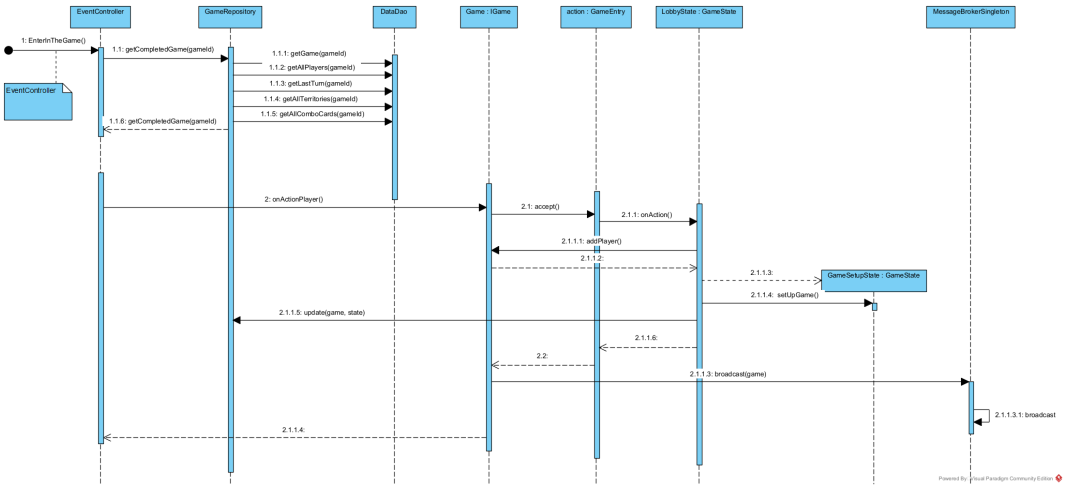


Figure 6: Diagamma di sequenza - entrata in lobby

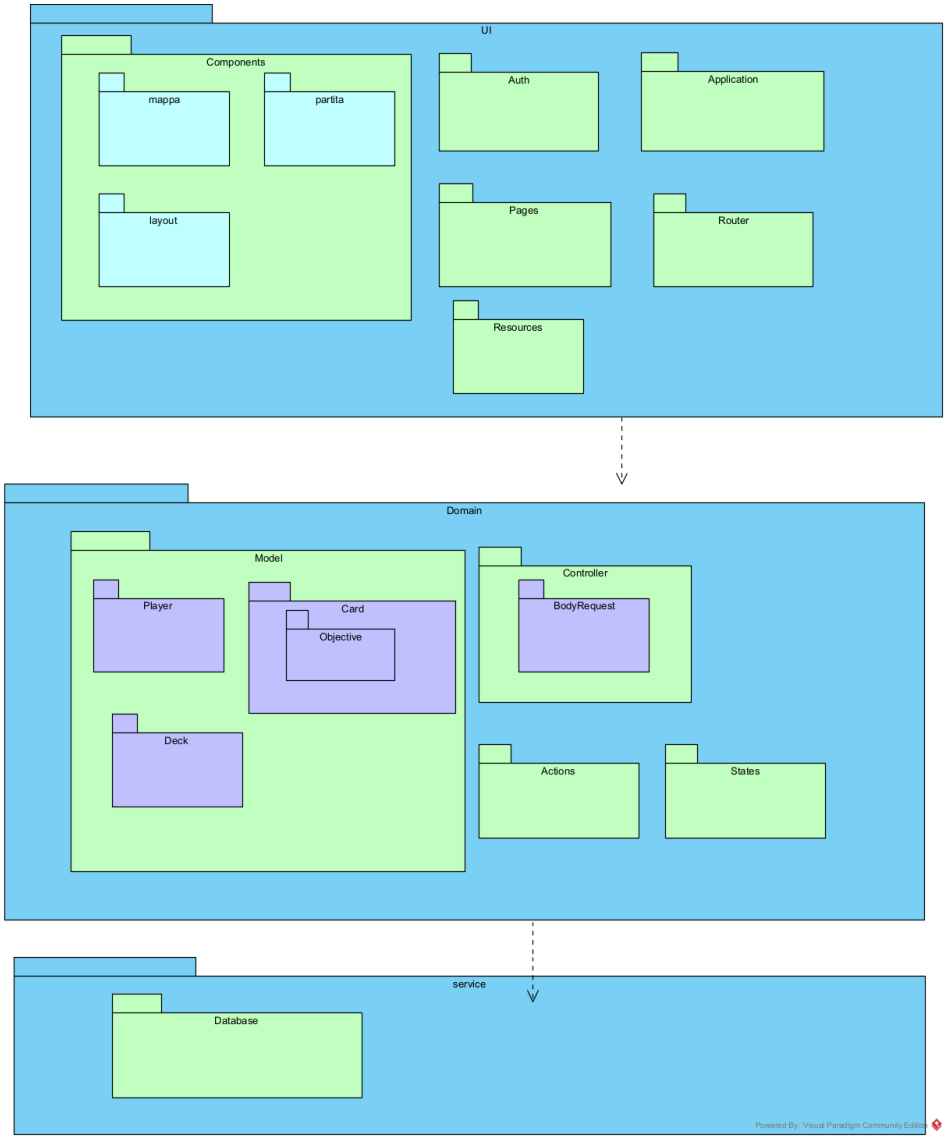


Figure 7: Diagramma dell'architettura software

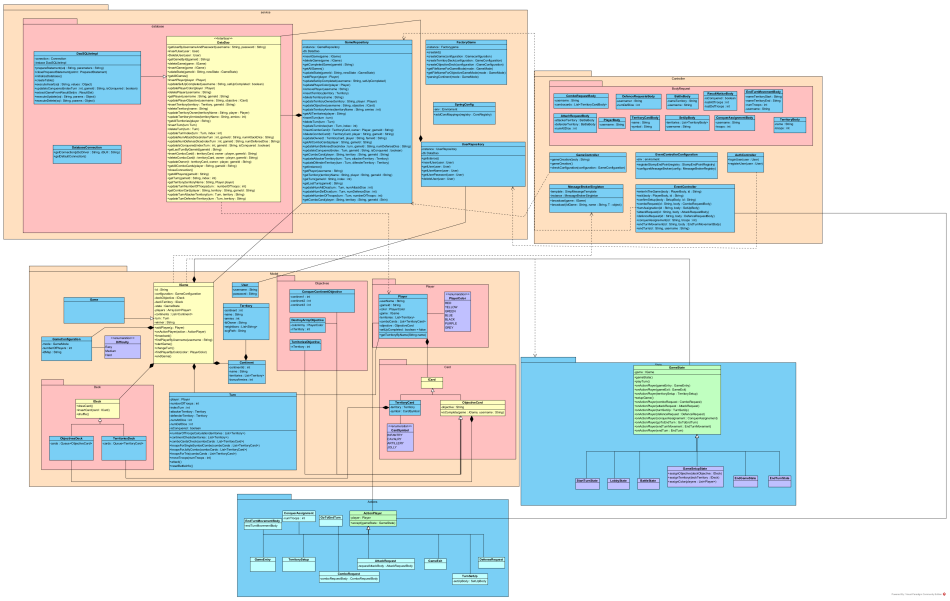


Figure 8: Diagramma delle classi software

4.2 Tecnologie Utilizzate

Di seguito sono elencate le principali tecnologie e strumenti adottati nello sviluppo del progetto:

- **Eclipse:** IDE polivalente adottato per lo sviluppo in Java, scelto per la sua familiarità tra i membri del team e la capacità di estensione tramite plugin.
- **Spring:** Framework selezionato per il backend, nonostante la novità per il team, grazie alla sua compatibilità con Java e il supporto al protocollo STOMP per la comunicazione in real-time.
- **Node e react:** abbiamo usato Node.js per gestire il nostro ambiente di sviluppo frontend, ottimizzando il lavoro con React per creare interfacce utente dinamiche. Questo approccio ha sfruttato la versatilità di Node.js oltre il lato server, facilitando il deployment e migliorando l'efficienza dello sviluppo grazie all'esperienza di un membro del team con React.
- **SQLite:** Scelto per la persistenza dei dati, favorito per la sua leggerezza e facilità d'integrazione.
- **Librerie aggiuntive:**
  - *Lombok:* Integrata per minimizzare il boilerplate code, fornendo annotazioni per la generazione automatizzata di getter, setter e altri metodi comuni.
  - *Jacoco:* Implementato per generare report dettagliati sulla copertura dei test.
  - *Jackson:* Utilizzato per la serializzazione e deserializzazione di oggetti Java in formato JSON, sfruttando la classe 'ObjectMapper'.
- **Git:** Impiegato per il versionamento e la collaborazione, con una strutturazione dei branch così definita:
  - **Branch main:** Contiene le release ufficiali, marcate con tag.
  - **Branch develop:** Dedicato allo sviluppo e aggiornamenti continui delle feature.
  - **Branch release:** Riservato per le versioni stabili del software, pronto per le release.
  - **Branch personali:** Destinati al lavoro individuale degli sviluppatori, i cui cambiamenti vengono integrati in 'develop' dopo la conclusione delle task.

- **Understand:** Strumento per l'analisi statica del codice e identificazione di code smells.
- **SonarQube e SonarCloud:** Utilizzati per l'analisi qualitativa del codice sia in locale che in remoto.

4.3 Principi SOLID

- **Single Responsibility Principle (S):** diverse classi sono state progettate per avere una sola responsabilità o compito, migliorando così la testabilità e la manutenibilità del codice
- **Open/Closed Principle (O):** il nostro sistema è stato ideato per essere estendibile, permettendo l'aggiunta di nuove funzionalità senza alterare il codice esistente. Ciò si ottiene attraverso la possibilità di ridefinire metodi specifici per la gestione di eventi, senza necessità di modificare altre parti del software.
- **Liskov Substitution Principle (L):** abbiamo assicurato che le sottoclassi possano essere utilizzate al posto delle loro classi base senza compromettere l'integrità del sistema. Questo principio garantisce che il nostro design sia corretto dal punto di vista dell'ereditarietà.
- **Interface Segregation Principle (I):** abbiamo progettato interfacce specifiche per i diversi clienti del nostro sistema, evitando di costringere le classi a implementare interfacce che non utilizzano. Questo principio contribuisce a mantenere il codice pulito e focalizzato.
- **Dependency Inversion Principle (D):** per ridurre l'accoppiamento tra le classi concrete e promuovere una maggiore flessibilità, abbiamo introdotto astrazioni come *IDeck*, *IGame*, *ICard*. Questo approccio ci permette di dipendere dalle astrazioni piuttosto che dalle implementazioni concrete.

4.4 Principi PHAME

- **Polymorphism :** Il principio del polimorfismo è applicato efficacemente per esempio attraverso le interazioni tra la classe `ObjectiveCard` e le sue sottoclassi. In questo contesto, `ObjectiveCard` funge da classe base per una serie di sottoclassi specifiche, ognuna delle quali implementa un comportamento unico pur aderendo all'interfaccia definita dalla classe padre.

- **Abstraction :** l'astrazione viene esemplificata nel progetto attraverso l'uso della classe `GameState` e del suo metodo `onActionPlayer()`. Questa classe astratta fornisce una definizione generale di un'azione che il giocatore può compiere, lasciando alle classi derivate il compito di implementare il metodo in modi specifici che rispecchiano le regole e le dinamiche di ciascuno stato del gioco. Attraverso l'astrazione, il progetto nasconde la complessità sottostante di ciascuno stato del gioco dietro un'interfaccia semplice, permettendo così di gestire facilmente diversi comportamenti del giocatore in contesti vari senza compromettere la flessibilità o l'estendibilità del codice.

- **Encapsulation :** L'incapsulamento è un principio ampiamente utilizzato nel progetto per proteggere lo stato interno degli oggetti. Attraverso l'incapsulamento, i dettagli implementativi di come un oggetto esegue una determinata funzionalità sono nascosti, esponendo solo le operazioni sicure e necessarie all'esterno. Questo approccio non solo previene accessi o modifiche indesiderate allo stato interno degli oggetti ma promuove anche una maggiore modularità e manutenibilità del codice.

4.5 Design patterns utilizzati

- **Singleton** utile per salvare informazioni a cui si può accedere da ogni parte del programma ottenendo sempre la stessa istanza. In particolare è stato usato LazySingleton, perchè l'istanza viene creata solo quando viene chiamato il metodo getIstance().
  Esempi di classi : FactoryGame, MessageBrokerSingleton, DaoSQLiteImpl

- **Monitor Object:** usato per gestire l'accesso concorrente alle risorse condivise in modo sicuro e efficiente, tramite la keyword synchronized

- **State :** è stato utilizzato perchè il comportamento di ogni partita dipende dallo stato in cui si trova la partita stessa
  Classe interessate : GameState, GameSetupState, BattleState, LobbyState, StartTurnState

- **Facade :** nel contesto del nostro progetto, il pattern **Facade** è illustrato efficacemente dall'implementazione della classe `DaoSQLiteImpl`. Questa classe funge da interfaccia semplificata per l'interazione con il database SQLite, offrendo un'astrazione che permette alle altre parti dell'applicazione di effettuare operazioni di database come la lettura, scrittura, aggiornamento e cancellazione dei dati.

- **Observer :** utilizzato nel frontend che viene notificato da messageBroker usato nel backend, per gestire il cambio stato degli oggetti e aggiornarli dopo la notifica. *Classe interessata :* PartitaObserverSingleton
- **Subscriber :** utilizzato col protocollo STOMP per una messaggistica in tempo reale scalabile ed efficiente
- **Factory Method :** fornisce un'interfaccia per creare un oggetto in una classe, ma permette alle sottoclassi di modificare il tipo di oggetto che viene istanziato, un esmepio è *FactoryGame*.
- **Crud :** "Create Read Update Delete" questo pattern definisce le operazioni utilizzate sul nostro database
- **Controller :** l'utilizzo di questo pattern fornisce una separazione tra logica di business e l'interfaccia utente, il controller, nel nostro progetto `EventController` per esempio, gestisce l'input dell'utente manipola i dati attraverso il modello e determina la vista da mostrare all'utente. *Classi interessate :* DataDao, DaoSQLiteImpl
- **Builder :** usato mediante la libreria Lombok di Maven evita l'utilizzo esplicito di costruttori e di conseguenza errori di coerenza nell'instanziare gli oggetti

4.6 Architectural patterns utilizzati

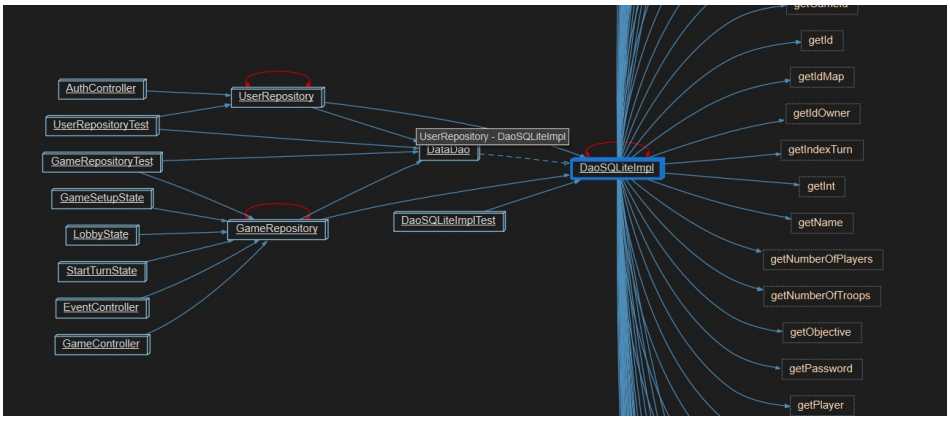
- **Repository e Data Access Object (DAO):** Questi pattern sono stati impiegati per disaccoppiare la logica di business dalla logica di accesso ai dati. Ciò è particolarmente evidente nel trattamento dei dati relativi agli utenti registrati e alle partite. Le classi che incarnano questi pattern includono `UserRepository`, `GameRepository` per il layer di Repository, e `DataDao`, `DaoSQLiteImpl` per il DAO, assicurando un'interfaccia uniforme per l'accesso ai dati, indipendentemente dal meccanismo di persistenza sottostante.
- **Model View Controller (MVC):** Abbiamo strutturato il sistema seguendo il pattern MVC, con una netta distinzione tra le componenti. Il `EventController` funge da controller, gestendo la logica di risposta agli eventi generati dal frontend, particolarmente tramite `StompController`. Il backend agisce come modello, gestendo lo stato dell'applicazione, che viene aggiornato e poi notificato alle viste attraverso il metodo `broadcast()`. Quest'ultimo sfrutta `PartitaObserverSingleton` per informare le viste dei cambiamenti, seguendo il principio dell'observer per mantenere sincronizzate le viste con lo stato attuale del modello.

- **Message Channel:** Per gestire le sessioni di gioco e assicurare che tutti i giocatori rimangano aggiornati con lo stato corrente della partita, è stato implementato un pattern Message Channel. Questo approccio prevede l'uso di un message broker, al quale i client si abbonano per ricevere aggiornamenti. Tale meccanismo facilita la comunicazione asincrona e la distribuzione degli aggiornamenti di stato a tutti i giocatori connessi, garantendo che l'esperienza di gioco coerente e sincronizzata per tutti i partecipanti.

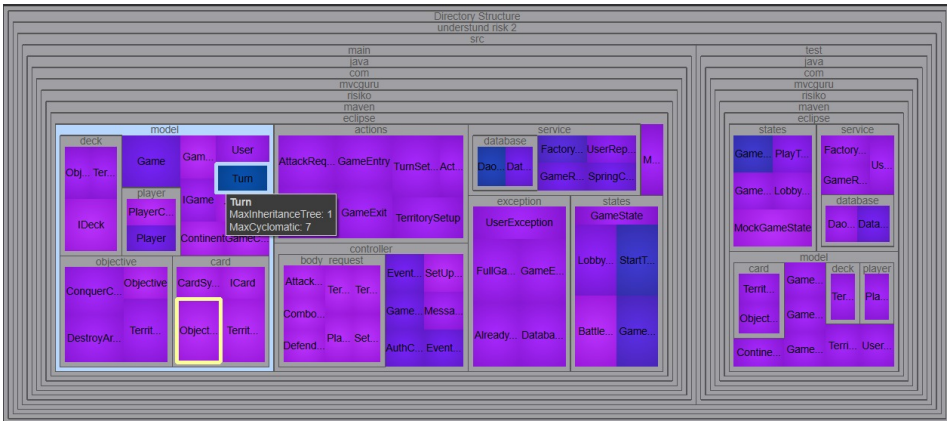
Questi pattern architetturali sono stati fondamentali per strutturare il sistema in modo robusto, flessibile e manutenibile, consentendo un'evoluzione facilitata del software e una chiara separazione delle responsabilità all'interno dell'applicazione.

Analisi del Codice

Grazie al tool Understand abbiamo analizzato le metriche fornite dalle analisi sul nostro progetto, facendo particolare attenzione ad avere determinate metriche sotto al valore di soglia critico (ad esempio max cyclomatic ¡ 10). Inoltre, abbiamo progettato l'interfaccia del nostro database in modo che fosse un facade controller e grazie ai grafici delle chiamate forniti da understand siamo risuciti a mantenere una struttura coerente con quanto progettato.



(a) Grafo chiamate database



(b) Metric Treemap