UNIVERSITÀ MILANO BICOCCA

FACOLTÀ DI INFORMATICA

E3101Q119 Corso di Ingegneria del software

2022-2023

Relazione di Progetto DRisk

Marinoni,Mauri, Palmerini, Zorzin

Relazione di Progetto DRisk

Marinoni,Mauri, Palmerini, Zorzin

January 26, 2023

Contents

Abstract

Il progetto consiste nella progettazione analisi e svilluppo del sistema software per l'online version del gioco Risk. In questa relazione mostreremo i passi fontamentali e i documenti principali della costruzione del sistema affrontando le scelte di analisi e progettazione. A seguito di una breve introduzione sul gioco tradizionale e sui cambiamenti fatti svolgeremo un excursus sui documenti utili alla progettazione, seguiti poi dall'illustrare e commentare le scelte implementative.

Per maggior informazioni leggere la wiki cliccandoci sopra.

1 Introduzione

Risk è un gioco da tavolo a turni in cui dai 2 ai 6 giocatori si sfidano per occuppare territori.

Ad ogni giocatore è assegnato un obiettivo casualmente che può essere di diverso tipo:

- Sconfiggere un altro giocatore;
- Conquistare un certo quantitavo di territori;
- Conquistare determinate porzioni di mappa (continenti);

Il turno di ogni giocatore è diviso in varie fasi in cui solo un determinato tipo di azioni sono possibili; la partita va avanti finchè uno dei giocatori non raggiunge il suo obiettivo o tutti gli altri si arrendono.

Il gioco classico è però molto rigido: per esempio nella costruzione della mappa o nella difficoltà della partita, il progetto consiste nel cercare di risolvere queste questioni di rigidità, sfruttando la virtualizzazione del gioco, dando così la possibilità all'utente di caricare la propria mappa e permettendo di settare diversi livelli di difficoltà che corrispondono alla quantità di territori presenti sulla mappa, che porterà di conseguenza a un cambio di obiettivi.

Per fare ciò, abbiamo dovuto dapprima analizzare il gioco classico per capire quali requisiti dovessimo implementare, per poi trovare il modo di rendere il gioco più dinamico e meno rigido. Iniziamo parlando quindi dell'analisi e progettazione.

2 Analisi e Progettazione

2.1 Casi d'uso

Come primo passo, abbiamo analizzato diversi casi d'uso che il sistema avrebbe dovuto ricoprire per poter iniziare a comprendere quali fossero i requisiti funzionali e non funzionali da implementare nel corso degli sprint. Di seguito alcuni esempi:

- caso d'uso inizializzazione;
- caso d'uso turno,
- caso d'uso combattimento,
- caso d'uso spostamento.

I casi d'uso analizzati ci hanno permesso di progettare un diagramma dei casi d'uso. Notiamo che iniziano a delinearsi i componenti necessari per la costruzione del sistema(Appendice fig.1).

2.2 Diagramma delle classi a livello di dominio

Con questo diagramma vediamo che iniziano a delinearsi le componenti che abbiamo ritenuto fondamentali. Infatti sono presenti: il Player, che rappresenta l'utente che interagirà con il sistema; la Map che è costituito da un'aggregazione di Continent che a loro volta sono costituiti da aggregazione di Territory. Da questi componenti possono essere generati poi anche le CartaTerritorio, aggregati poi in dei DeckTerritori, e in analogo gli Obiettivi e i DeckObiettivi. Di determinare chi sarà il player che deve giocare sarà l'oggetto Turno, il tutto sarà poi gestito da un sistema per ora chiamato "Partita", che cambierà i vari stati del gioco per permetterne il corretto funzionamento (Appendice, fig.2).

Come vedremo poi, questo sistema Partita si è sviluppato nel componente ”StatiPartita”, che tiene assieme tutto e permette al giocatore di fare la sua partita. È possibile instanziare più partite che saranno differenziate dall'idPartita, che permetterà anche ai giocatori ”ospiti” di collegarsi alla partita corretta.

2.3 Diagramma di Dominio Classi software

Rispetto al modello di dominio sviluppato prima vediamo come abbiamo implementato le classi elencate(Appendice, fig.4).

Come si può notare il sistema è composto da tante classi ed è abbastanza complesso. IState è l'interfaccia che nel modello di dominio illustrato precedentemente era indicato dall'oggetto StatiPartita, attraverso delle funzioni che permettono il passaggio da uno stato partita all'altro il sistema ha visuale di una o più componenti. Una classe particolare è Dice.java, che è una classe utility indipendente da ogni altra classe perchè deve poter essere invocata da tutte le altre al momento del bisogno. Grazie al fatto che tutte le classi fanno capo a IState possiamo notare che ci sono poche associazioni tra classi con diverse resposabilità fatto per abbassare la possibilità che si creino dipendenze cicliche. Come si può vedere tutte le classi hanno i propri metodi e funzioni e non sono presenti data-class, ne classi troppo piene grazie all'assegnamento di responsabilità. Svolto attravverso i principi di Information Expert e Pure Fabrication, come nel caso dell'oggetto Turno(Appendice, fig. 10 e fig.11), ci hanno guidato all'assegnamento delle resposanbilità cercando di aumentare la coesione e diminuire la duplicazione di codice. Nell'ottica di buon assegnamento di resposabilità abbiamo deciso anche di concedere ai territori e di consequenza alla mappa la conoscenza di quale giocatore sia in possesso di un determinato territorio, visto che durante la progettazione la classe Player stava diventando sempre più grande abbiamo quindi deciso di delegare alcune responsabilità lasciando a player giusto le informazioni che concernano l'utente.

2.4 Architettura

A livello architetturale è organizzato a package. I principali sono 3 Domain, GUI e Servizi. In particolare il Domain è organizzato in vari altri sotto package ognuno contenenti le classi che gestiscono una diversa responsabilità. L'uso di GitHub ha favorito e permesso un'architettura di questo tipo perchè ogni responsabilità era rappresentata da una issue e di conseguenza da un branch che ci ha permesso di lavorare in simultanea su diversi aspetti del sistema (fig.13).

2.5 Diagrammi Di Sistema

Per farsi un'idea migliore su come il sistema debba agire abbiamo poi sviluppato i diagrammi di sistema e di sequenza sistema(Appendice fig.3; fig.12). Possiamo così vedere come si sviluppa l'inizializzazione di una partita e come si svolge un turno. In particolare una partita viene creata da un utente player che può aggiungere giocatori in una lobby fino ad esaumerimento posti, dopo di che il sistema crea la partita e assegna i territori ai player che posizionano le loro armate a turno nel modo che preferiscono dopo di che con l'elenco dei giocatori il sistema da il turno al primo giocatore che dovrà iniziare il suo primo turno di gioco.

Il turno di un giocatore invece inizia con il sistema che manda un segnale all'oggetto turno che farà iniziare il prossimo giocatore che in ordine prima piazzerà i suoi rinforzi, poi compirà delle azioni se esse porteranno alla conquista di un territorio il giocatore potrà pescare una carta territorio, per poterla riscattare (se possibile) nella sua prossima fase di rinforzo.

2.6 Diagrammi di classi a livello di progettazione

In questa sezione sono presenti i diagrammi delle varie componenti del dominio e di come ragionano tra di loro.

Il primo diagramma è riguardo il GameState (Appendice, fig.5). Il package contenente gli stati di una partita: da quando viene iniziata e tutte le fasi di un turno. Funziona usando uno State Pattern: Un interfaccia la quale include i vari metodi che le varie classi implementano, per esempio il metodo ”endTurn()” che cambia lo State in un NewTurnState. Gli stati presenti sono: il NewGameState che fa da pozzo per aspettare che la partita cominci; l'InitialFaseState che serve per la fase iniziale del gioco in cui gli utenti scelgono i loro territori iniziali di cui impadronirsi; il NewTurnState conclude il turno corrente e inizia quello successivo; il RenforceState da un numero di armate al giocatore in base ai territori che possiede e, inoltre, può guadagnare un magggior numero di esse in base a una combo di carte, come definito dalle regole ufficiali del Risiko; l'ActionState in cui il giocatore può sciegliere se attaccare un territorio nemico e poi spostare le sue armate o solo attuare lo spostamento ma concludere poi automaticamente il suo turno.

Le classi richiamano spesso l'interfaccia IContext che va a prendere informazioni dalla mappa e dai deck di carte territorio e carte obiettivi: questa interfaccia è connessa infatti ad altri package tra cui Map, Turn e i deck sopraccitati.

Il Map parte da una classe che ha una collezione di continenti che a sua volta contiene una collezione di territori. Questi ultimi includono una lista di territori stessi che sono ”vicini” tra loro cosi da definire la corretta disposizione di essi sulla mappa.

Il Turn è una singola classe che gestisce una lista di Player che sono in gioco e può concludere una partita quando uno dei giocatori stessi completasse uno degli obiettivi.

Il Dice è una singola classe connessa a Turn che computa il valore di uscita di un semplice dado il quale viene usato per richiedere il lancio dei dadi stessi al fine di stimare il vincitore o perdente di un attacco.

Il Player è la classe dove si contengono non solo i dati di un giocatore, tra cui le carte che possiede, ma anche le azioni che avvengono in gioco, come attaccare un altro giocatore o rinforzare i propri territori. Questa classe istanzia un interfaccia iPlayer per agevolare e standardizzare i metodi con le altre classi.

Objectives è la classe che genera gli obiettivi che verranno poi distribuiti come carte al giocatore. La classe principale è un interfaccia che implementa i vari tipi di obiettivi:

- OpponentDefeated nel caso devi sconfiggere un determinato giocatore;
- TotTerritories nel caso devi conquistare un determinato numero di territori;
- ConquestContinent nel caso devi conquistare uno specifico continente.

Le Cards infine, contengono un numero di carte in base al numero di territori e di obiettivi presenti rispettivamente in DeckTerritories e DeckObjectives. Per i DeckTerritories crea una carta per territorio associandone per ognuna di esse un tipo di armata tra ”Fante”, ”Cavallo”, ”Cannone” e ”Jolly” (Create in una classe separata). Per i DeckObjectives prende tutti i tipi di obiettivi in Objectives possibili, tra cui: la sconfitta di un giocatore, il conquistare un numero variabile di territori (il 65% per due giocatori e 62% se di più) e la conquista dei continenti.

2.7 State Machine & Activity Diagrams

Abbiamo scelto di riportare come State Machine Diagram l'oggetto GameState(Appendice, fig.6)che sembrava il più adeguato allo scopo di illustrare meglio come avvenisse il passaggio tra le varie fasi

del turno e della partita. Come activity Diagram abbiamo scelto il diagramma attività del Action State, uno degli stati in cui ci si troverà all'interno della partita (Appendice, fig.7).

3 Implementazione

3.1 Backend

Ora vedremo come la fase di progettezione si è evoluta per risolvere i problemi di implementazione che abbiamo incontrato. A partire dalla gestione delle partite usando lo State Pattern per il gameState, o anche il pattern Strategy per lo sviluppo degli obiettivi.

Lo Strategy ha un funzionamento molto simile allo State ma, a differenza di questo, che serve a mostrare i diversi componenti di oggetti simili, lo Strategy Pattern permette di incapsulare una serie di algoritmi correlati e al client di mutarli a sua scelta in fase di esecuzione. Quindi il primo è usato per incapsulare oggetti, Strategy Pattern incapsula una serie di algoritmi correlati e consente al client di utilizzare comportamenti intercambiabili attraverso la composizione e la delega in fase di esecuzione.

Un altro pattern utilizzato è stato il controller ovvero, oggetti creati allo scopo di gestire gli input provenienti dal presentation layer e, conseguentemente, di trasferirli al corretto oggetto che ne detiene la responsabilità nel domain layer.

Nel backend sono presenti inoltre i già citati Information Expert e Pure Fabrication.

Codesmells & Bugs Report Dovendo consegnare un progetto valutato "A" su SonarCloud, gli unici code smells presenti nel progetto riguardano la sintassi della nomeclatura dei pacchetti che, anche provando a sistemarli, il compilatore non trovava più quegli oggetti necessari alla compilazione stessa dei test, anche se gli import risultavano corretti.

A riguardo dei bugs, invece, è degno di nota solo un major bug di sicurezza riguardante la classe Dice.java nella quale, per "lanciare" il dado, avevamo inserito un metodo Math.random(), che però era visto appunto come un bug grave per la sicurezza, quindi l'abbiamo infine sostituito con un oggetto SecureRandom.

Anti-Pattern Cercando di evitare di inserire degli Anti-Pattern, ci è stato impossibile non comprendere nel nostro codice una dipendenza ciclica (a cappio) sulla classe Territory(Appendice, fig.8) al fine di poter avere una lista di territori vicini. Un altro anti-Pattern trovato è una Local-Butterfly (Appendice, fig.9) costituita dalla classe Map.java, in quanto ha molti pacchetti dai quali dipende e altrettanti che dipendono da lei.

La Map, quindi, è una classe molto stabile perché è difficile in primis modificarla ma la si può estendere abbastanza facilmente. La classe Dice, al contrario, è una classe altamente instabile perchè non avendo dipendenze è molto facile modificarla.

3.2 Frontend

Un design pattern usato per il funzionamento del front-end è il Data Transfer Object, meglio noto come DTO, il quale consiste nel creare oggetti i quali contengono i dati che è necessario trasportare; questo, infatti, permette un minor uso delle chiamate ai metodi.

I DTO normalmente vengono creati come POJO, infatti, sono strutture di dati piatte che non contengono alcuna logica aziendale, contenenti solo:

- Archiviazione
- Funzioni di accesso
- Metodi relativi alla serializzazione o all'analisi.

Il DTO permette e anzi necessita soprattutto dell'uso di un Mapper, il quale costituisce l'interfaccia dell'oggetto utilizato.

Bugs Report Avendo scelto per il nostro Frontend come tecnologia Vue.js, abbiamo dovuto affrontare come problematica del trasporto dati tra le varie componenti del nostro progetto.

Dopo molteplici analisi, siamo giunti alla conclusione, un po' per inesperienza con la tecnologia scelta un po' per l'effettiva difficoltà nel testare tramite Jest ciò che avevamo prodotto, di semplificare il Frontend, non più con un copioso numero di componenti, ma limitandoci ad una componente per il menu e, di conseguenza, una per il gioco.

Anti-Pattern L'unico Anti-Pattern constatato nel Frontend, a causa dei bug sopraccitati, è una struttura monolitica, la quale abbiamo inizialmente provato a scomporre ma, come riportato nella sezione precedente, e a causa del poco tempo rimanente, abbiamo deciso di mantenere cosi.

4 Conclusioni

Questo progetto è stato un esperienza interessante, e scrivere un sistema così complesso è stata un'ardua sfida. Abbiamo constatato che, tramite gli insegnamenti che ci sono stati dati sia nel corso di Analisi e Progettazione Software sia in quello di Ingegneria del Software, abbiamo potuto strutturare e ben identificare già da subito quali fossero le criticità e, di conseguenza degli iniziali metodi risolutivi, per il progetto stesso.

La più grande criticità sorta, che ci ha obbligato ad una semplificazione del progetto, è stata l'inesperienza nel comunicare tra i vari componenti del gruppo, infatti abbiamo dovuto nell'ultima settimana riuscire a connettere Backend e Frontend in modo celere e poco ottimale.

In conclusione, è stata un esperienza sicuramente molto utile per comprendere come si lavora di gruppo in un progetto di medio/grandi dimensioni che in confronto ad altri corsi non permettono di sperimentare cosi a tuttotondo.

A Allegati

Figure 1: Use Case Diagram, 2.1

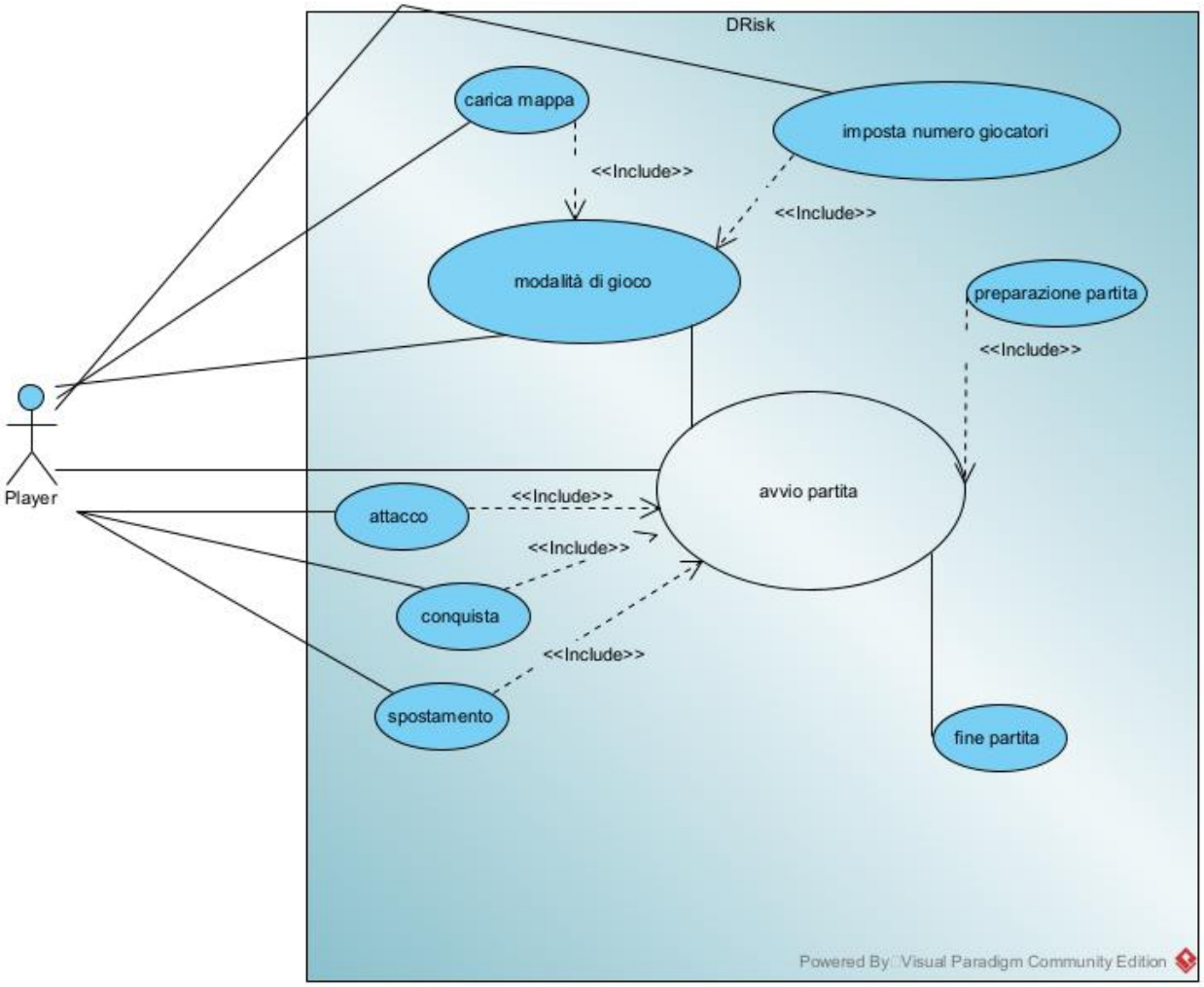

Figure 2: domain model, 2.2

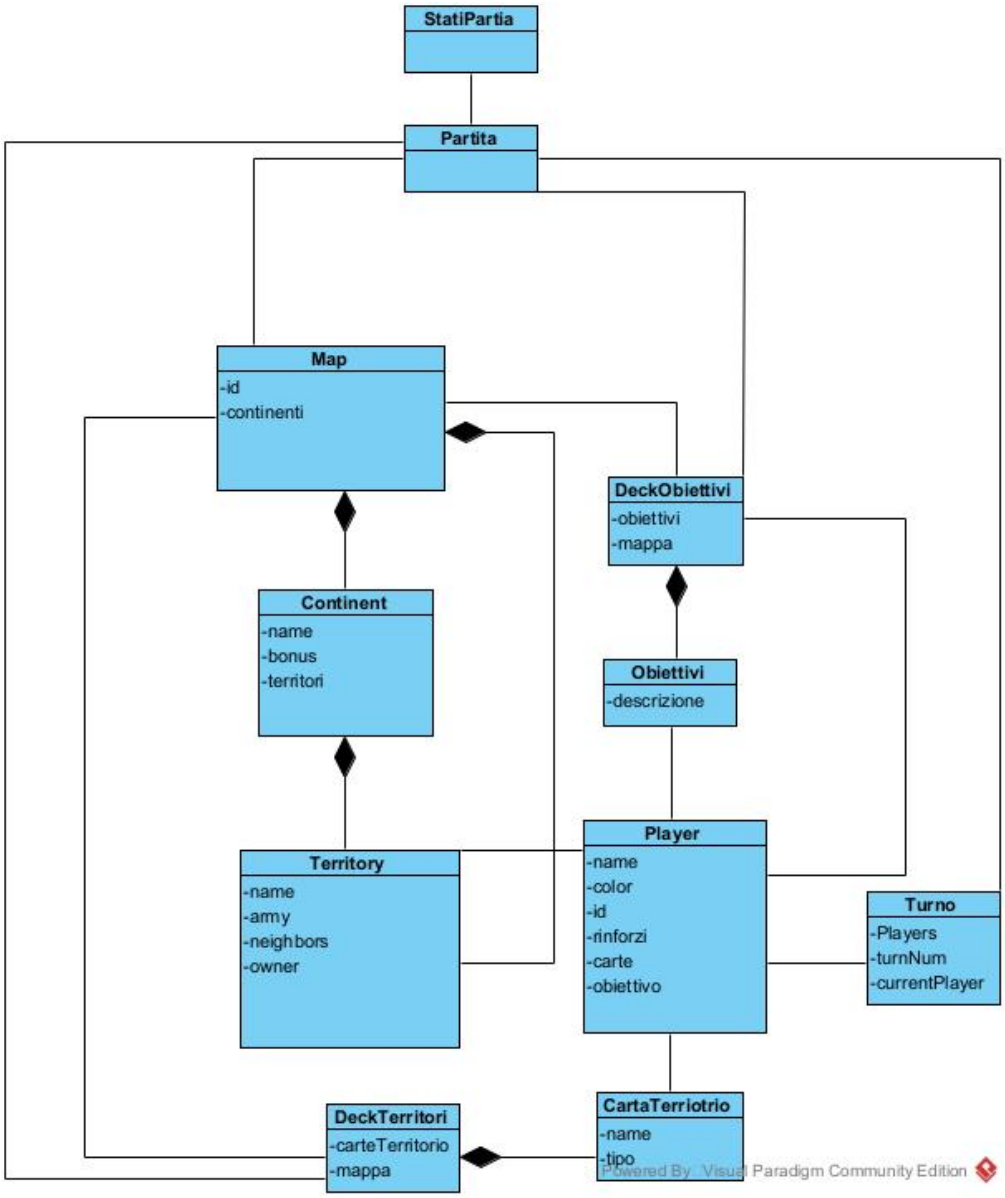

Figure 3: SSD inizializza Partita, 2.5

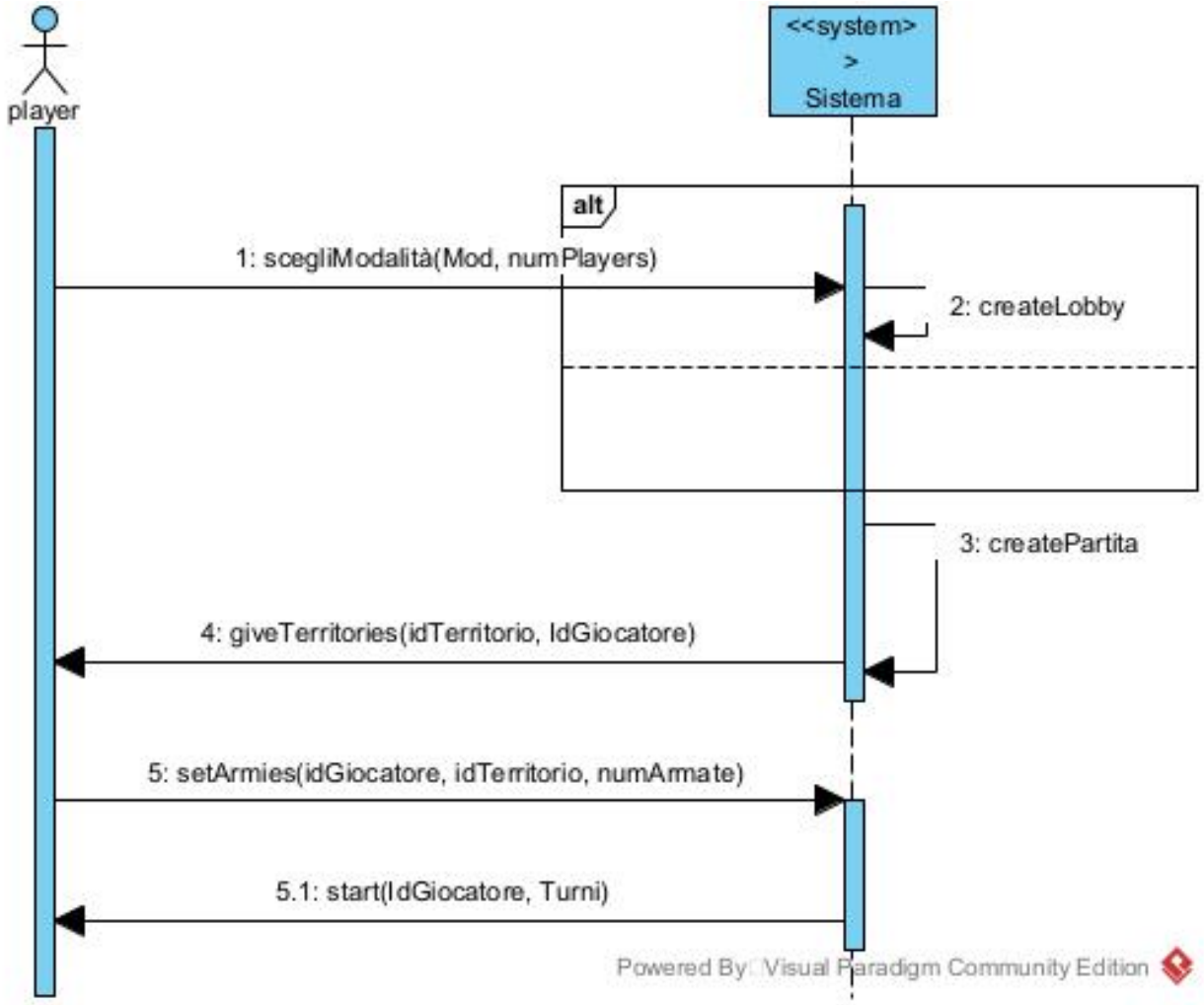

Figure 4: modello di dominio classi software, 2.3

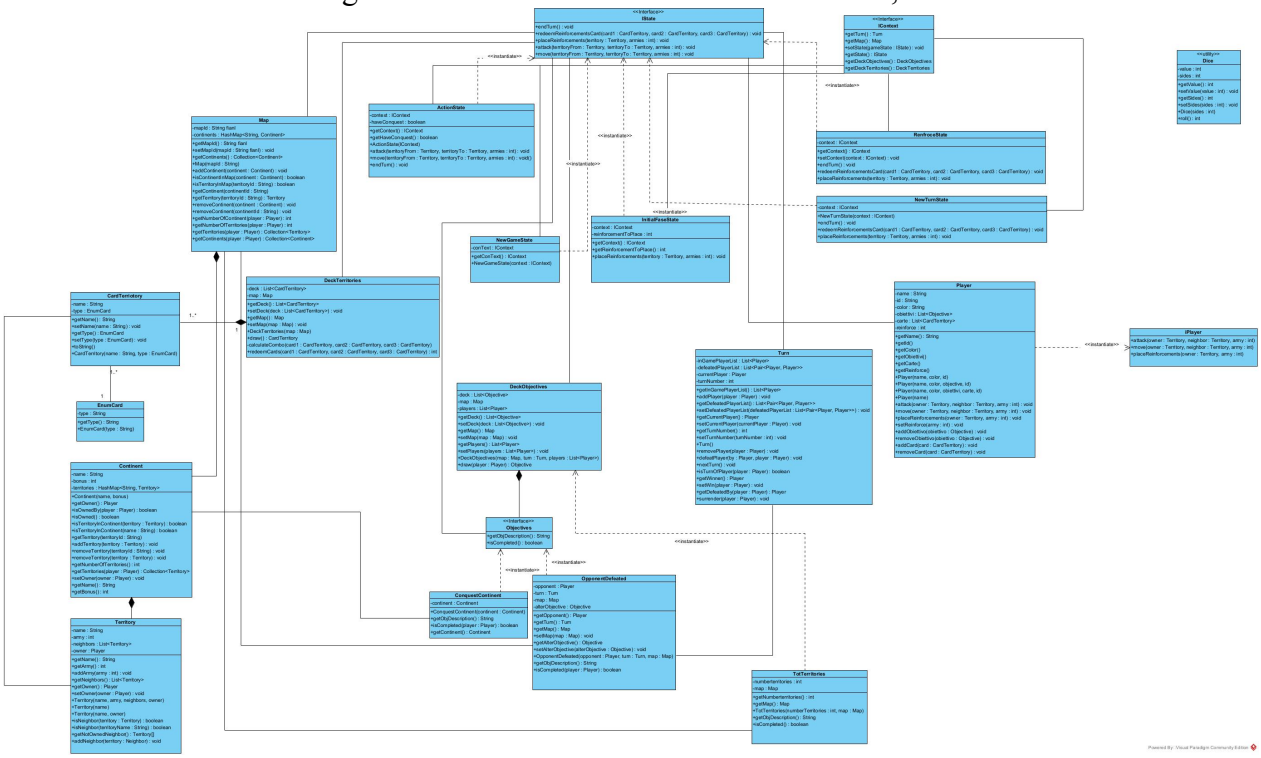

Figure 5: gameState, 2.3

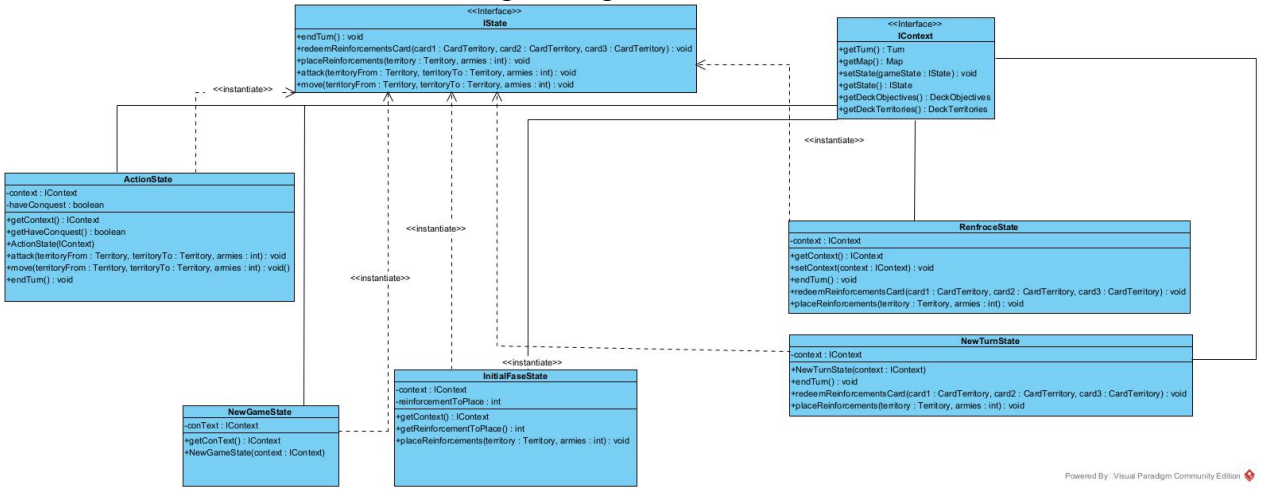

Figure 6: state machine diagram, 2.7

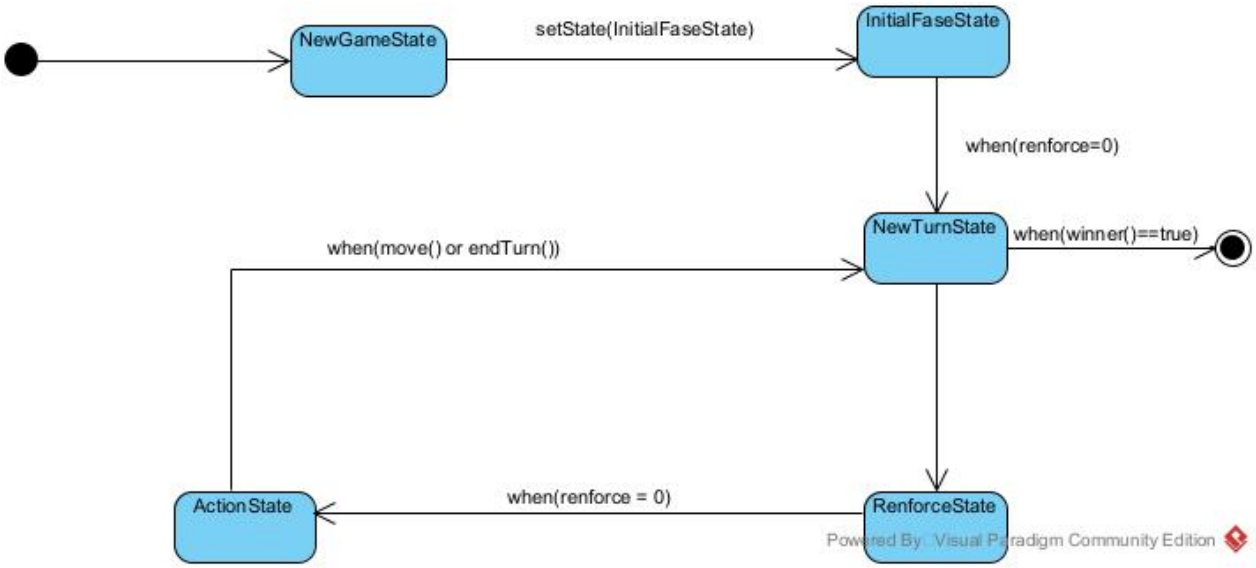

Figure 7: activity diagram, 2.7

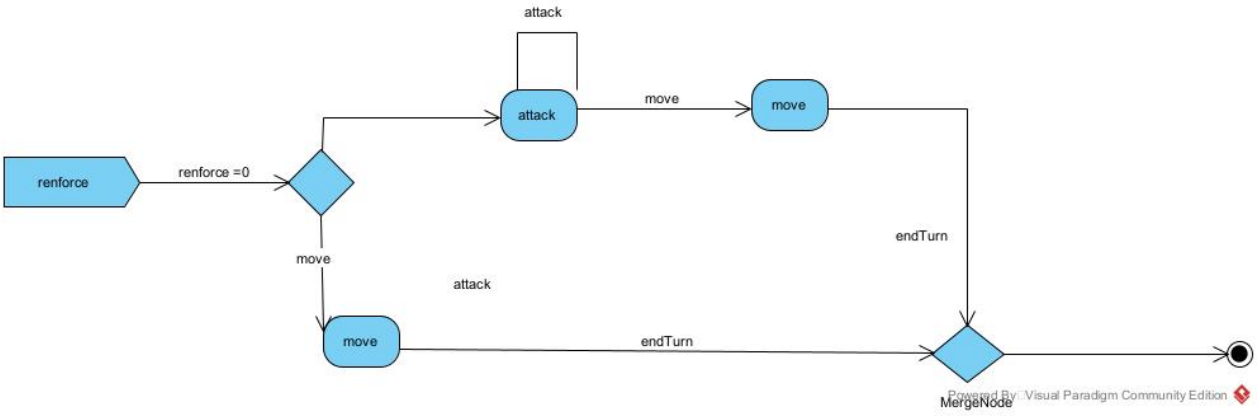

Figure 8: screen analisi grafica di understand, per la classe Territory, 3.2

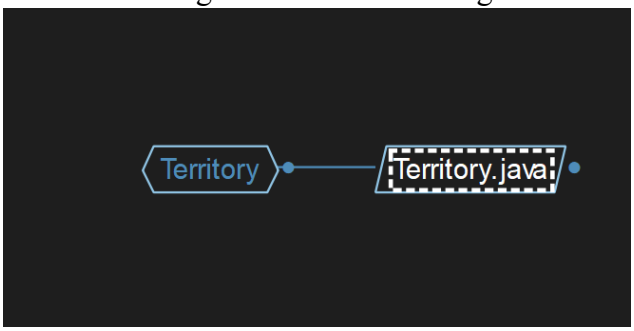

Figure 9: screen analisi grafica di understand, per la classe Map, 3.2

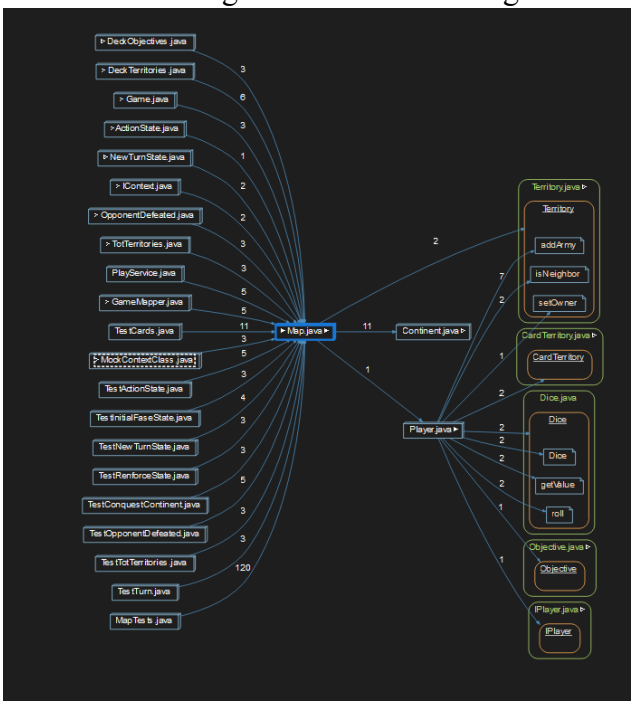

Figure 10: Turno prima, 2.3

Turn
- playerList : List <Player> - currentPlayer : Player - nextplayer: Player - turnNumber : int - d : Dice
+ Turn (playerList, d, turnNumber) + Turn (playerList) + setPlayerOrder(playerList, d) : void + getDice() : Dice + getPlayerList (): Listt <Player> + setPlayerList (playerList): void + getCurrentPlayer : Player + playersInGame(playerList): List <Player> + setCurrentPlayer (playerList): void + setNextPlayer(playerList) : void + goHeadTurn:() void + getTurnNumber (): int + winner(playersListInGame) : Player + winningCOndition(playersListInGame): void

Figure 11: Turno dopo, 2.3

Turn
-inGamePlayerList : List<Player> -defeatedPlayerList : List<Pair<Player, Player>> -currentPlayer : Player -turnNumber : int
+getInGamePlayerList() : List<Player> +addPlayer(player : Player) : void +getDefeatedPlayerList() : List<Pair<Player, Player>> +setDefeatedPlayerList(defeatedPlayerList : List<Pair<Player, Player>>) : void +getCurrentPlayer() : Player +setCurrentPlayer(currentPlayer : Player) : void +getTurnNumber() : int +setTurnNumber(turnNumber : int) : void +Turn() +removePlayer(player : Player) : void +defeatPlayer(by : Player, player : Player) : void +nextTurn() : void +isTurnOfPlayer(player : Player) : boolean +getWinner() : Player +setWin(player : Player) : void +getDefeatedBy(player : Player) : Player +surrender(player : Player) : void

Figure 12: Diagramma di Sequenza, 2.5

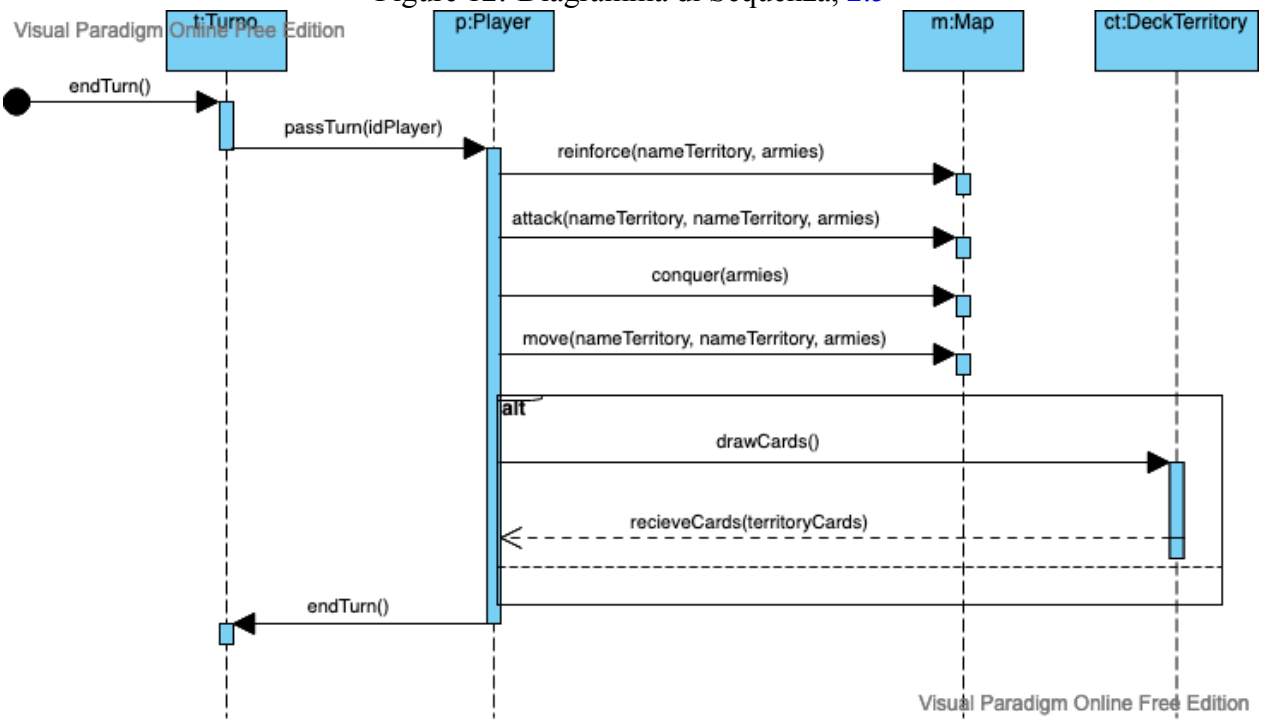

Figure 13: Architettura, 2.4