“Programmazione ad Oggetti”

Forti Mattia
Marchetti Davide
Pagliarani Martino
Pracucci Filippo

Giugno 2023

Indice

Capitolo 1

Analisi

1.1 Requisiti

L'applicazione vuole emulare il gioco da tavolo "Risiko", con qualche modifica al regolamento per rendere le partite più veloci. La partita avverrà tra diversi giocatori (all'interno dello stesso calcolatore) e si dovranno seguire le regole del gioco. Nel dettaglio si seguiranno le varie fasi della partita:

- **Preparazione della partita**: distribuzione casuale degli obiettivi e dei territori e distribuzione delle armate fornite dalla dotazione iniziale di ogni giocatore .
- **Fase di rinforzo**: il giocatore di turno riceve un numero di armate aggiuntive in base al numero di territori in suo possesso e ai continenti conquistati. Le armate possono essere posizionate a discrezione del giocatore sui propri territori.
- **Fase di combattimento**: il giocatore di turno può scegliere di attaccare un territorio confinante ai propri, avviando un combattimento. In caso di vittoria l'attaccante pesca una carta e, se il territorio attaccato ha perso tutte le truppe, può scegliere quante truppe spostare dal territorio da cui ha attaccato a quello appena conquistato.
- **Fase di spostamento strategico**: il giocatore di turno può spostare truppe tra i propri territori, purché siano adiacenti, prima di terminare il proprio turno. Una volta entrato in fase di spostamento il giocatore non potrà più attaccare.

Inoltre il giocatore di turno può giocare le sue carte attivando delle combinazioni prefissate, che gli permettono di ottenere truppe aggiuntive che potrà posizionare durante la sua fase di rinforzo al turno successivo.

Requisiti funzionali

- Il gioco dovrà permettere di gestire le varie fasi della partita, consentendo di interagire correttamente con i territori.
- I turni dei giocatori dovranno rispettare le fasi della partita, consentendo di compiere determinate azioni solo in caso di requisiti verificati, per esempio un giocatore non può spostare truppe da un territorio con 1 truppa ad un altro, perché lascerebbe il suo territorio con 0 armate e quindi indifeso.
- Ad ogni turno il giocatore riceverà truppe aggiuntive in base ai territori conquistati.
- Il combattimento tra due giocatori dovrà essere effettuato mediante il lancio di dadi da parte di entrambi i giocatori. Il massimo di truppe attaccanti o difendenti è 3 e il risultato del combattimento sarà deciso casualmente dal lancio dei dadi.
- Il giocatore di turno, dopo aver conquistato un territorio, pescherà una carta dal mazzo delle armate. Tramite specifiche combinazioni di carte, il giocatore potrà ottenere delle truppe aggiuntive, che potrà quindi piazzare sui suoi territori al turno successivo durante la fase di rinforzo.
- Ogni giocatore dovrà arrivare al compimento del proprio obiettivo per determinare la fine della partita.

Requisiti non Funzionali

- Il gioco dovrà essere efficiente e reattivo ai comandi dei giocatori.
- I giocatori dovranno essere in grado di visualizzare il campo di gioco e i loro territori in modo intuitivo e chiaro.
- L'interfaccia di gioco dovrà essere semplice e intuitiva.

1.2 Analisi e modello del dominio

RisikOOP si pone l'obiettivo di simulare una partita al gioco da tavolo "Risiko" tra 3 giocatori. Il gioco prevede le interazioni tra le diverse entità giocatore, le quali possiedono un'armata suddivisa sui diversi territori e un obiettivo da raggiungere per vincere la partita. I giocatori interagiscono tra

loro all'interno del tabellone di gioco, il quale è composto da diverse entità territorio che sono a loro volta raggruppate in continenti. I territori adiacenti sono collegati tra di loro permettendo il combattimento e lo spostamento di truppe. Il gioco mette a disposizione un mazzo di carte formato da tre tipologie (fanteria, cavalleria e artiglieria), da cui il giocatore pesca una carta ogniqualvolta avvenga una conquista di un territorio nemico. Lo stato del gioco è determinato dopo ogni fase di gioco tramite un controllo degli obiettivi di tutti i giocatori. Gli elementi costitutivi il problema sono sintetizzati in Figura 1.1. Le principali difficoltà sono la gestione dei territori tra giocatori diversi, le azioni dei giocatori quali combattimento e spostamento e il controllo degli obiettivi di fine partita.

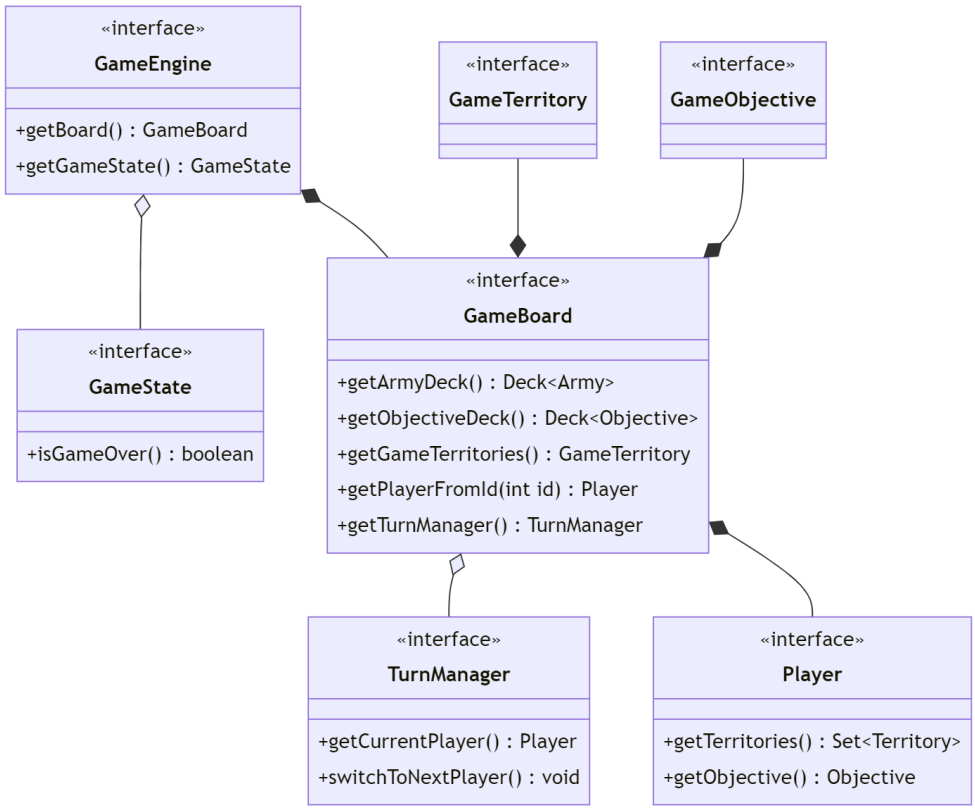


Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti tra essi.

Capitolo 2

Design

2.1 Architettura

L'architettura di RisikOOP segue il pattern architetturale MVC (Model-View-Controller) e utilizza le librerie di Java Swing per gestire l'interfaccia grafica. L'utente ha la possibilità di interagire con i pulsanti dell'interfaccia grafica e, quando un pulsante viene premuto, il controller invia l'input al modello che lo processa e aggiorna i valori interni, per poi comunicare nuovamente al controller di aggiornare l'interfaccia grafica. La pressione di un pulsante determina un'azione corrispondente alla fase di gioco e al giocatore di turno. Con questa architettura possono essere aggiunti un numero arbitrario di territori, obiettivi e giocatori senza dover modificare le meccaniche di gioco. Lo stesso discorso non può essere fatto per le carte che forniscono truppe aggiuntive, in quanto si dovrebbero aggiungere nuove tipologie di carte con le relative combinazioni. In caso di aggiunta di giocatori è necessario aumentare il numero di carte disponibili nei vari mazzi. Di seguito viene spiegata la divisione dei ruoli seguita nel progetto per il pattern MVC:

- **Model**: il package *it.unibo.model* si occupa del modello, cioè della gestione di tutti i valori interni e l'esecuzione delle azioni su giocatori, territori, obiettivi e carte.
- **Controller**: il package *it.unibo.controller* viene utilizzato per mettere in comunicazione il modello con le interfacce grafiche (tabellone, pannelli e finestre a comparsa) e viceversa.
- **View**: il package *it.unibo.view* contiene le interfacce grafiche utilizzate. Servendosi di Java Swing, è stato scelto l'utilizzo di una finestra unica per separare il menù principale dalla schermata di gioco, cambiando all'occorrenza i pannelli contenuti al suo interno. Vengono uti-

lizzate diverse finestre a comparsa per gli eventi di combattimento e movimento.

In Figura 2.1 si può osservare lo schema dell'architettura di RisikOOP. Viene mantenuta la modularità del pattern MVC, in quanto la View può essere modificata o ampliata senza dover apportare modifiche al modello e viceversa. Questo è ottenibile implementando le interfacce di View già presenti che avranno le stesse funzionalità delle classi create con Java Swing, evitando conflitti con il modello. Anche l'aggiunta di territori, obiettivi e giocatori non porterebbe a modifiche nella View, permettendo quindi di ampliare il gioco a proprio piacimento senza dover cambiare l'architettura iniziale.

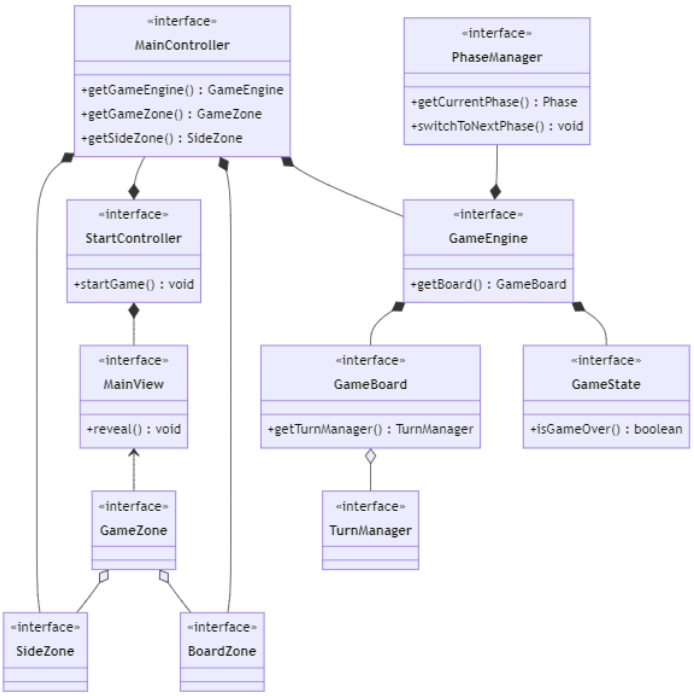
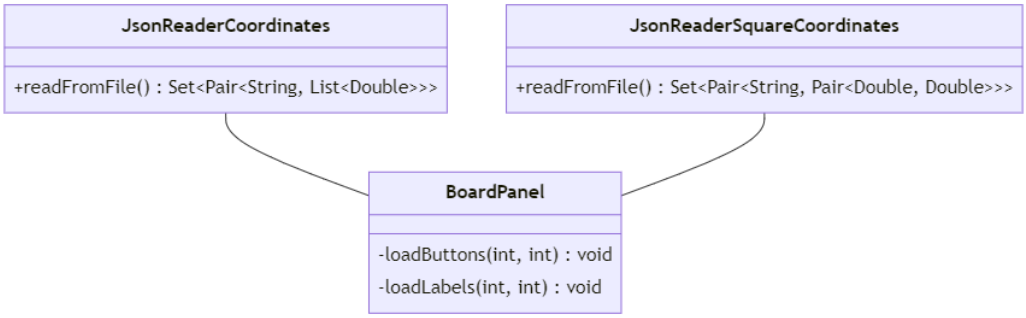


Figura 2.1: Schema UML dell'architettura di RisikOOP con tutte le parti relative al pattern MVC e le loro relazioni.

2.2 Design dettagliato

2.2.1 Forti Mattia

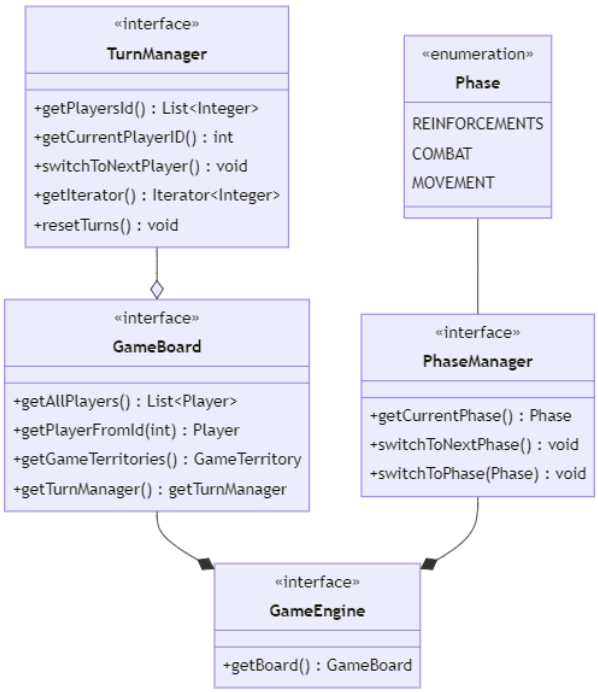
Creazione del tabellone



Problema Creazione di numerosi pulsanti e etichette con posizione e dimensioni diverse all'interno di un'unica classe.

Soluzione Vengono usate le classi `JsonReaderCoordinates` e `JsonReaderSquareCoordinates` per leggere tutti i dati necessari dai file `Coordinates.json` e `SquareCoordinates.json` nella cartella *resources*. Questi valori vengono passati alla classe `BoardPanel` che implementa `BoardZone`, che li utilizza per creare i pulsanti dei territori e le etichette che conterranno il numero di truppe nei relativi territori.

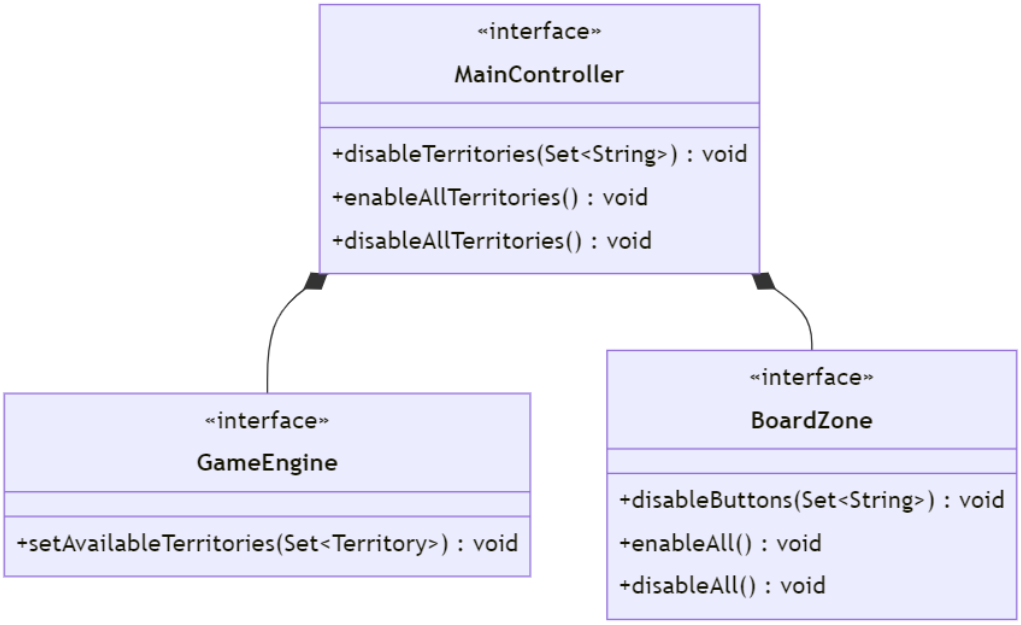
Separazione dei turni dalle loro fasi



Problema Ogni turno è suddiviso in diverse fasi nelle quali il giocatore può eseguire un'azione diversa, quindi cliccando un territorio durante le diverse fasi del turno si avranno diverse azioni, ciò complica la suddivisione dei compiti di gestione delle azioni eseguibili.

Soluzione La classe `TurnManager` gestisce i turni dei giocatori, mentre la classe `PhaseManager` gestisce le diverse fasi di un turno. Le due classi non hanno alcun legame tra loro e lavorano indipendentemente: `TurnManager` viene usata per passare da un giocatore all'altro e viene istanziata all'interno di `GameBoard`, che a sua volta viene istanziata in `GameEngine`; `PhaseManager` viene usata per passare da una fase all'altra e viene istanziata direttamente in `GameEngine`.

Controllo di corretta selezione di un territorio

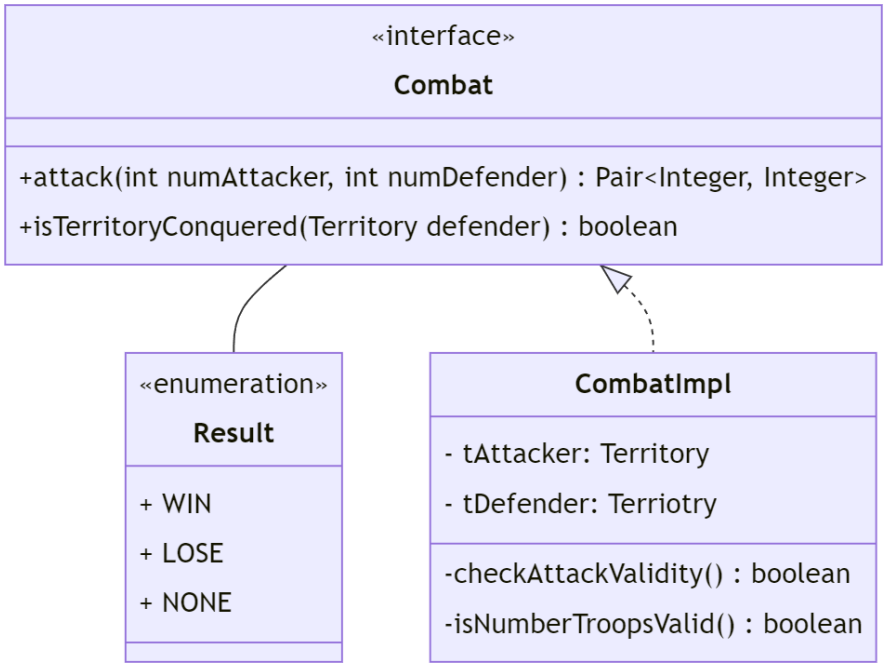


Problema Ogni giocatore deve essere in grado di selezionare solo i propri territori, oppure i territori nemici adiacenti in caso stia attaccando.

Soluzione Il metodo più efficiente per controllare se un giocatore ha cliccato un territorio valido è disattivare i pulsanti non validi. Ad ogni cambio di fase, il `GameEngine` identifica i territori con cui il giocatore può interagire e comunica al `MainController` di aggiornare la View, disabilitando i pulsanti non validi per la selezione. In questo modo viene eliminata la possibilità di cliccare un territorio non disponibile, evitando quindi di fare un controllo inutile.

2.2.2 Marchetti Davide

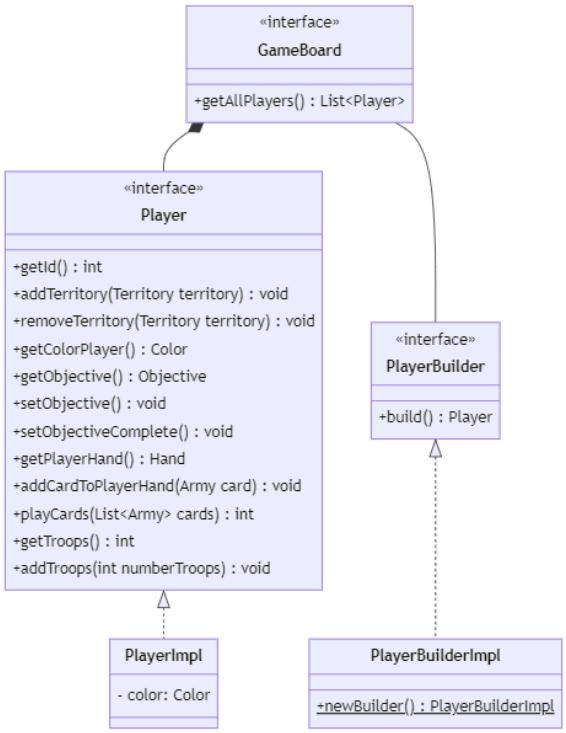
Gestione del combattimento tra due territori



Problema Creazione del combattimento tra due territori presenti nella mappa di gioco utilizzando un lancio di dadi per ogni truppa selezionata dal possessore del territorio. Una volta definiti i risultati di tutti i lanci dei dadi in modo automatico, questi devono essere confrontati tra di loro per determinare il risultato del combattimento.

Soluzione L'interfaccia `Combat` che permette di istanziare un combattimento tra due territori, viene implementata dalla classe `CombatImpl` in cui il costruttore prende come parametri i territori interessati nel combattimento. Questa classe gestisce il combattimento seguendo dei vincoli definiti all'interno dell'implementazione dell'interfaccia. Il combattimento avviene confrontando il risultato dei dadi lanciati a coppie in ordine decrescente così da poter definire il numero di truppe che sono state sconfitte.

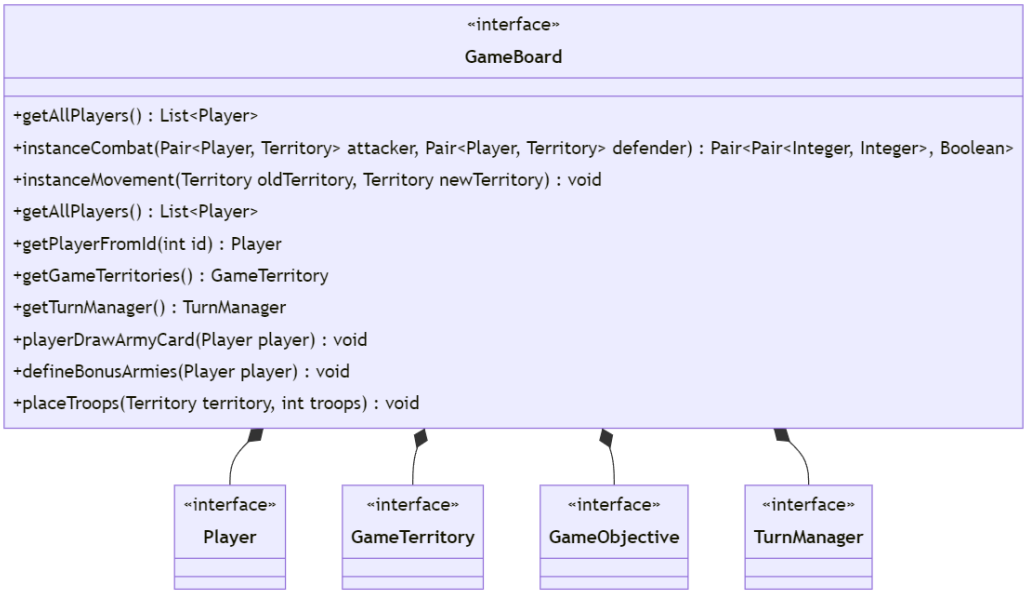
Creazione dei giocatori



Problema Creare un numero di giocatori definito in fase di progettazione, modellati dall'interfaccia `Player`.

Soluzione La classe `GameBoardImpl` ha il compito di creare i giocatori della partita ed inizializzarli. Viene utilizzato il pattern creazionale *Builder* con lo scopo di semplificare la creazione dei giocatori assegnando adeguati valori ai numerosi attributi dei giocatori. Viene costruito un oggetto `Player` mediante la classe `PlayerBuilderImpl`, accedendo staticamente al metodo *newBuilder* e garantendo la coerenza nell'inizializzazione dell'oggetto da costruire.

Collegamento delle varie entità

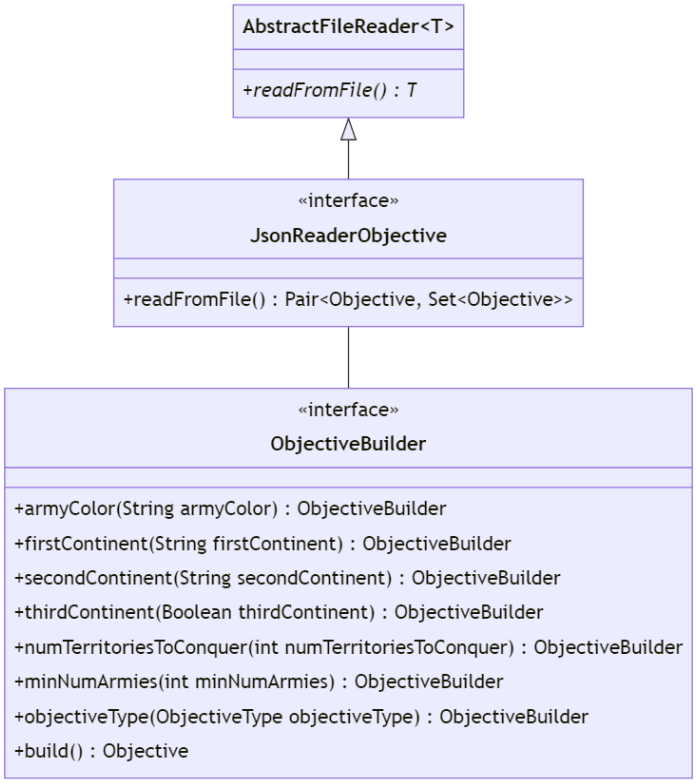


Problema Mettere in comunicazione le varie entità con il resto del modello.

Soluzione Per mettere in comunicazione le entità si fa uso dell'interfaccia `GameBoard`. Quest'ultima ha il compito di istanziare tutti gli oggetti necessari al corretto funzionamento dell'applicazione, fornendo dei meccanismi per accedervi. Inoltre mette a disposizione dei metodi per poter interagire con le istanze degli oggetti, modificandone le proprietà in modo adeguato senza effettuare operazioni pericolose rischiando di violare l'incapsulamento dei dati.

2.2.3 Pagliarani Martino

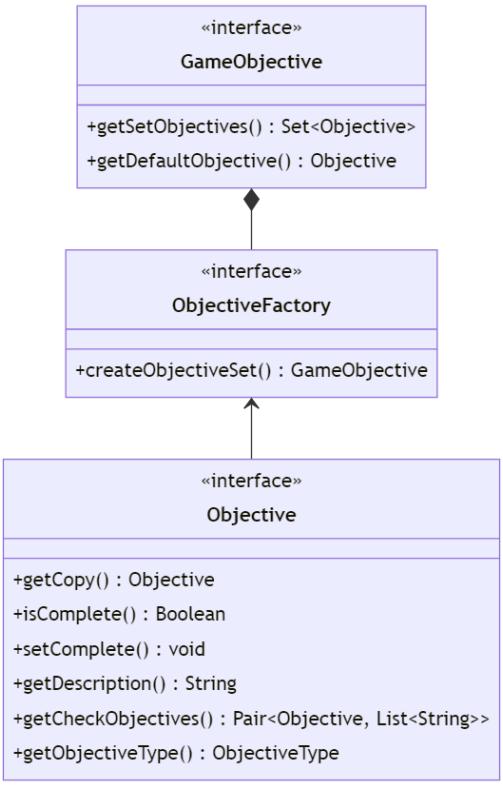
Creazione e lettura degli obiettivi



Problema Creare e leggere i diversi tipi di obiettivi da un file di configurazione.

Soluzione La classe `JsonReaderObjective`, che estende la classe astratta generica `AbstractFileReader`, si occupa della lettura degli obiettivi dal file di configurazione `Objectives.json`. Inoltre, viene utilizzato il pattern creazionale *Builder* per assegnare i valori corretti al rispettivo obiettivo in base al suo tipo nella classe `ObjectiveBuilder`.

Classificazione e controllo degli obiettivi

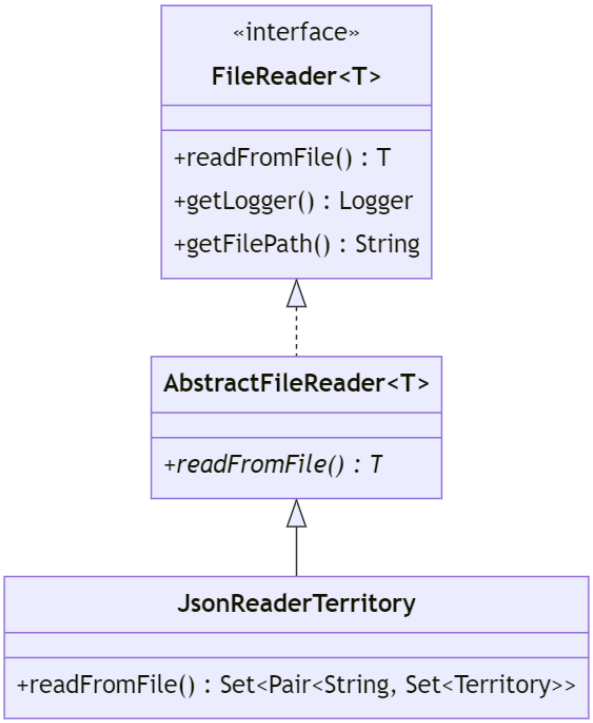


Problema Rappresentare i diversi tipi di obiettivi e controllarne il completamento.

Soluzione Viene utilizzato il pattern creazionale *Factory Method* nella classe `ObjectiveFactory` che si occupa di creare un oggetto di tipo `GameObjective` i quali metodi restituiscono l'obiettivo base e tutti gli obiettivi che la classe `GameState` controlla in base al tipo.

2.2.4 Pracucci Filippo

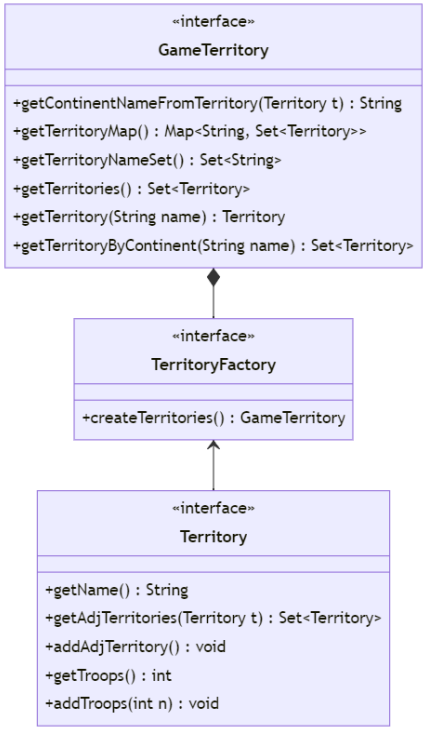
Lettura da file



Problema Lettura dei territori da un file di configurazione.

Soluzione L'interfaccia `FileReader` definisce la lettura da ogni tipo di file, questa viene implementata da una classe astratta generica `AbstractFileReader` in modo da consentire la sua estensione a diverse classi che si occupano della lettura. Nello specifico la classe `JsonReaderTerritory` la estende per leggere i territori da un file di configurazione di tipo JSON, per questo motivo viene utilizzata la libreria esterna *org.json.simple*. La libreria è stata scelta per motivi di comodità e semplicità di utilizzo.

Creazione dei territori



Problema Creazione dei territori, suddividendoli in continenti utilizzando un file di configurazione.

Soluzione Utilizzo del pattern creazionale *Factory Method* che utilizza `JsonReaderTerritory` per leggere i territori dal relativo file di configurazione. Il metodo factory è *createTerritories()* che si occupa di creare un oggetto di tipo `GameTerritory`, cioè una mappa che associa ad ogni continente il corrispondente insieme di territori che lo compongono. Questa modalità di creazione dei territori è stata scelta per facilitarne l'utilizzo all'interno dell'applicazione.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

I test sono stati realizzati utilizzando JUnit 5 e sono:

- **TestTerritory:** questa classe si occupa di controllare la correttezza della creazione dei territori. In particolare di un territorio vengono verificati nome, territori adiacenti e continente a cui appartiene.
- **TestObjective:** questa classe si occupa di controllare la creazione degli obiettivi di tutte le tipologie possibili: distruzione dell'armata di un avversario, conquista di continenti e conquista di un numero di territori presidiati da una minima quantità di truppe.
- **TestCombat:** questa classe si occupa di verificare la meccanica di combattimento, quindi controlla la validità del numero di attaccanti, il risultato dello scontro e la rimozione di truppe dal territorio perdente, con la possibilità di conquista di un territorio.
- **TestMovement:** questa classe si occupa di verificare la validità di uno spostamento di truppe tra due territori. Viene controllato che con lo spostamento il territorio a cui vengono tolte le truppe resti con almeno una truppa, in più i territori coinvolti nello spostamento devono essere adiacenti.

3.2 Metodologia di lavoro

L'architettura è stato il primo argomento affrontato, ragionando sulle diverse interfacce e sui nuclei del gioco, ovvero le classi principali che si occuperanno

della realizzazione del pattern MVC. Il modello è stato suddiviso in diverse entità a sé stanti, in modo da consentirci di effettuare un lavoro separato e riunirci solo quando necessario per discutere di eventuali problematiche o possibilità di riutilizzo di varie classi. Per la realizzazione della View e del Controller invece è stato necessario lavorare in gruppo per collegare le diverse parti sviluppate individualmente. Abbiamo seguito un metodo di lavoro basato sul DVCS, come spiegato a lezione, creando una repository su GitHub e definendo due branch: `main` e `dev`. Sulla branch `main` abbiamo caricato solo le modifiche definitive, con classi funzionanti e senza errori di *Checkstyle*, *PMD* o *SpotBugs*, mentre sulla branch `dev` abbiamo fatto tutte le modifiche necessarie per sistemare gli errori. Inoltre `dev` è stata usata come branch di prova per il funzionamento delle meccaniche di gioco.
La suddivisione dei compiti è presentata di seguito:

Forti Mattia

- Creazione del tabellone di gioco (package *it.unibo.view.gamescreen*).
- Creazione della barra laterale e del pannello dei pulsanti di azione (package *it.unibo.view.gamescreen*).
- Implementazione di un pulsante personalizzato (`CustomButton`).
- Logica del GameLoop (`GameEngine`), dei turni e delle fasi di gioco (package *it.unibo.gameengine* e package *it.unibo.model.turns*).
- Creazione del controller principale e sua implementazione (package *it.unibo.controller.gamecontroller*).
- Controllo di validità di spostamento delle armate tra territori (package *it.unibo.model.movement*).

Marchetti Davide

- Creazione dei giocatori e gestione dei loro attributi (package *it.unibo.model.player*)
- Creazione e gestione di un combattimento tra territori di due giocatori (package *it.unibo.model.combat*)
- Creazione e gestione del tabellone di gioco (package *it.unibo.model.board*)
- Creazione del file `RisikoMap.jpg` contenente la mappa di gioco.

- Implementazione del pannello, e del rispettivo controller, che permette ai giocatori di giocare le carte (package *it.unibo.view.gamescreen*), (package *it.unibo.controller.playerhand*).
- Implementazione della view, e del rispettivo controller, per il movimento di truppe tra territori di un giocatore (package *it.unibo.view.movement*), (package *it.unibo.controller.movement*).
- Creazione e gestione del controller per avviare l'applicazione (package *it.unibo.controller.gamecontroller*).

Pagliarani Martino

- Creazione e gestione dei mazzi di carte delle armate e degli obiettivi (package *it.unibo.model.deck*, package *it.unibo.model.army* e package *it.unibo.model.objective*).
- Creazione e gestione della mano del giocatore e dei dadi (package *it.unibo.model.hand* e package *it.unibo.model.dice*).
- Creazione del file `Objectives.json` con i diversi tipi di obiettivi.
- Implementazione della classe `JsonReaderObjective` per la lettura da file JSON (package *it.unibo.controller.reader*).
- Creazione e gestione della fase di preparazione della partita (package *it.unibo.model.gameprep*).
- Controllo della condizione di fine partita (package *it.unibo.gameengine.gamestate*).
- Creazione della finestra di combattimento (package *it.unibo.view.combat*).

Pracucci Filippo

- Creazione dei territori, classificati nei diversi continenti, tramite factory (package *it.unibo.model.territory*).
- Creazione file `Territories.json` con le definizioni di tutti i territori.
- Creazione dell'interfaccia `FileReader`, la sua implementazione nella classe astratta `AbstractFileReader` e relativa estensione `JsonReaderTerritory` per leggere i territori dal rispettivo file di configurazione (package *it.unibo.controller.reader*)
- Creazione `MainFrame` usato dai vari pannelli delle schermate di gioco (package *it.unibo.view.gamescreen.impl*)

- Creazione del menù iniziale (`MainPanel` nel package *it.unibo.view.gamescreen*)
- Pannello con le informazioni del giocatore momentaneamente in gioco (`InfoPanel` nel package *it.unibo.view.gamescreen.impl*)

Parti comuni

- `GameEngine`: unione di tutte le parti di modello sviluppate da ogni partecipante al progetto.
- Interfaccia grafica: unione del tabellone e della barra laterale all'interno della finestra di gioco.
- Unione dei pannelli di informazioni, carte e azioni all'interno della barra laterale
- `GameBoard`: unione di tutte le parti del modello continuativamente nel ciclo vitale dell'applicazione.
- Risoluzione di tutti gli errori di *Checkstyle*, *PMD* e *SpotBugs*.

3.3 Note di sviluppo

Forti Mattia

Utilizzo di Streams e Lambda Expressions

Utilizzo di Streams e le Lambda Expressions per accorciare il codice e renderlo più intuitivo, soprattutto nelle classi che devono utilizzare le liste di territori e di giocatori. Di seguito vengono presentati alcuni esempi:

- Permalink: `https://github.com/davmarc-lab/OOP22-RisikOOP/blob/8d3e7d2eda0872e91c6c40ed304b3cec7435b460/src/main/java/it/unibo/gameengine/impl/GameEngineImpl.java#L196-L198`
- Permalink: `https://github.com/davmarc-lab/OOP22-RisikOOP/blob/8d3e7d2eda0872e91c6c40ed304b3cec7435b460/src/main/java/it/unibo/model/movement/impl/MovementImpl.java#L39-L41`
- Permalink: `https://github.com/davmarc-lab/OOP22-RisikOOP/blob/8d3e7d2eda0872e91c6c40ed304b3cec7435b460/src/main/java/it/unibo/controller/gamecontroller/impl/MainControllerImpl.java#L146-L150`

Utilizzo di Optional

Utilizzo di Optional per evitare la notazione null, così da poter inizializzare delle variabile senza valore. Di seguito un esempio:

- Permalink: https://github.com/davmarc-lab/OOP22-RisikOOP/blob/465746fdf6ee3c1d46209cce7b55c410f13fcd03/src/main/java/it/unibo/view/gamescreen/impl/BoardPanel.java#L70-L98

Marchetti Davide

Utilizzo di Streams e Lambda Expressions

Utilizzo frequente per migliorare la stesura del codice e la sua leggibilità. Di seguito vengono elencati alcuni esempi:

- Permalink: https://github.com/davmarc-lab/OOP22-RisikOOP/blob/8d3e7d2eda0872e91c6c40ed304b3cec7435b460/src/main/java/it/unibo/controller/playerhand/impl/PlayerHandControllerImpl.java#LL119C9-L121C26
- Permalink: https://github.com/davmarc-lab/OOP22-RisikOOP/blob/8d3e7d2eda0872e91c6c40ed304b3cec7435b460/src/main/java/it/unibo/model/board/impl/GameBoardImpl.java#LL161C9-L165C39
- Permalink: https://github.com/davmarc-lab/OOP22-RisikOOP/blob/8d3e7d2eda0872e91c6c40ed304b3cec7435b460/src/main/java/it/unibo/model/combat/impl/CombatImpl.java#LL171C9-L174C53
- Permalink: https://github.com/davmarc-lab/OOP22-RisikOOP/blob/8d3e7d2eda0872e91c6c40ed304b3cec7435b460/src/main/java/it/unibo/view/gamescreen/impl/CardPanel.java#LL138C9-L147C12

Pagliarani Martino

Utilizzo di Streams e lambda expressions

Utilizzo frequente di Streams e lambda expressions per migliorare l'efficienza e la chiarezza del codice. Alcuni esempi:

- Permalink: https://github.com/davmarc-lab/OOP22-RisikOOP/blob/1104a5e2bccea7b090f2a90e49414e4215243f33/src/main/java/it/unibo/model/gameprep/impl/GamePrepImpl.java#LL50C1-L68C6

- Permalink: https://github.com/davmarc-lab/OOP22-RisikOOP/blob/1104a5e2bccea7b090f2a90e49414e4215243f33/src/main/java/it/unibo/gameengine/impl/GameStateImpl.java#LL32C1-L36C6
- Permalink: https://github.com/davmarc-lab/OOP22-RisikOOP/blob/1104a5e2bccea7b090f2a90e49414e4215243f33/src/main/java/it/unibo/model/dice/impl/DiceImpl.java#LL43C1-L47C6

Utilizzo di Optional

Utilizzati per evitare notazioni null e definire il risultato del combattimento.

- Permalink: https://github.com/davmarc-lab/OOP22-RisikOOP/blob/1104a5e2bccea7b090f2a90e49414e4215243f33/src/main/java/it/unibo/controller/combat/impl/CombatControllerView.java#LL71C1-L75C6

Pracucci Filippo

Utilizzo di Streams e lambda expressions

Utilizzo frequente per accorciare e rendere più intuitivo il codice, in particolare nell'uso di collezioni. Di seguito sono forniti alcuni esempi:

- Permalink: https://github.com/davmarc-lab/OOP22-RisikOOP/blob/1104a5e2bccea7b090f2a90e49414e4215243f33/src/main/java/it/unibo/model/territory/impl/TerritoryImpl.java#L84-L92
- Permalink: https://github.com/davmarc-lab/OOP22-RisikOOP/blob/1104a5e2bccea7b090f2a90e49414e4215243f33/src/main/java/it/unibo/model/territory/impl/GameTerritoryImpl.java#L33-L39
- Permalink: https://github.com/davmarc-lab/OOP22-RisikOOP/blob/1104a5e2bccea7b090f2a90e49414e4215243f33/src/main/java/it/unibo/model/territory/impl/GameTerritoryImpl.java#L53-L57

Utilizzo di Optional

Utilizzati per il caricamento dell'immagine di sfondo del menù principale:

- Permalink: https://github.com/davmarc-lab/OOP22-RisikOOP/blob/bcac1b0281f3f5b035b7cee1a16e6575492991a2/src/main/java/it/unibo/view/gamescreen/MainPanel.java#L90-L93

Interfaccia e classe astratta generica

Utilizzo della genericità per consentire la lettura da diversi tipi di file.

- Permalink: `https://github.com/davmarc-lab/OOP22-RisikOOP/blob/8d3e7d2eda0872e91c6c40ed304b3cec7435b460/src/main/java/it/unibo/controller/reader/impl/AbstractFileReader.java#L29`

Libreria esterna

Utilizzo della libreria esterna *org.json.simple* per consentire la lettura da file di tipo JSON.

- Permalink: `https://github.com/davmarc-lab/OOP22-RisikOOP/blob/8d3e7d2eda0872e91c6c40ed304b3cec7435b460/src/main/java/it/unibo/controller/reader/impl/JsonReaderTerritory.java#L53-L88`

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Forti Mattia

Mi ritengo soddisfatto del lavoro svolto. È stato il mio primo progetto di gruppo, perciò non sono mancate le difficoltà iniziali nella decisione dell'architettura e dell'organizzazione generale del progetto, ma lavorando con gli altri, condividendo idee e opinioni, siamo riusciti a rendere la realizzazione del progetto quanto più semplice possibile. Ciò che più mi ha aiutato a migliorare la qualità del mio codice sono stati i controlli di *Checkstyle*, *PMD* e *SpotBugs* del *Build Tool* di *Gradle*, che mi hanno abituato a scrivere codice molto più "pulito" e leggibile. Lavorare sulla View mi ha interessato molto, e preferirei migliorarla, aggiungendo dettagli e animazioni, eventualmente anche con JavaFX. L'unico punto debole che mi piacerebbe risolvere in futuro con conoscenze più avanzate è il fatto che le partite a RisikOOP si svolgono sullo stesso calcolatore, perciò ogni giocatore può vedere l'obiettivo degli altri; l'ideale sarebbe renderlo un gioco online, cosicché ognuno possa giocare sul proprio dispositivo senza "spiare" gli altri.

Marchetti Davide

Sono molto soddisfatto del lavoro svolto insieme agli altri membri del gruppo, questo mi ha fatto capire quanto sia importante avere un confronto con altre persone durante l'analisi di un problema. Durante la realizzazione del progetto mi sono reso conto di quanto sia complesso lavorare in gruppo, in quanto richiede un'accurata organizzazione e coordinazione tra i vari membri. Nonostante ciò ho potuto apprezzare i numerosi vantaggi della cooperazione con gli altri partecipanti, specialmente la condivisione di opinioni diverse per

raggiungere lo stesso obiettivo. Grazie ad uno strumento di controllo versione distribuito, nel nostro caso *GitHub*, ci è risultato molto semplice riuscire a lavorare in modo efficiente. Durante l'esecuzione del progetto mi sono reso conto delle difficoltà nel modellare una buona architettura per un'applicazione e vorrei in futuro riuscire a migliorare in questo campo, cercando di ristrutturare questo progetto oppure dedicarmi ad un nuovo lavoro.

Pagliarani Martino

Ritengo la realizzazione di questo progetto molto importante per capire il gioco di squadra all'interno di un gruppo di programmazione. Essendo il mio primo progetto, mi ha aiutato molto a mettere in pratica più su grande scala ciò che ho imparato in questi pochi anni di programmazione. Inoltre mi ha aiutato a capire l'importanza dell'analisi e dell'organizzazione iniziale del progetto e mi ha dato un'idea generale di come potrebbe essere il lavoro in questo ambito. I controlli forniti dal docente di *Checkstyle*, *PMD* e *SpotBugs* sono stati molto utili a migliorare il mio stile di programmazione in quanto rendevano chiari problemi che ignoravo precedentemente ma che sono riuscito a risolvere. In futuro mi piacerebbe rivedere il progetto ampliandolo con espansioni che aggiungano modalità diverse di gioco e meccaniche di gioco come potenziamenti e abilità. Infine la realizzazione di questo progetto è stata fondamentale al percorso di studi che sto seguendo in quanto mi ha dato un'impostazione corretta e uno stile di programmazione migliore.

Pracucci Filippo

La realizzazione di questo progetto mi ha fatto capire quali sono i vantaggi e le maggiori difficoltà nel lavoro di gruppo, in particolare lo scambio di idee e consigli è stato molto d'aiuto e credo mi abbia permesso di realizzare codice migliore. L'utilizzo di *GitHub*, seppur in modo basilare e limitato, mi ha permesso di capire più a fondo il suo funzionamento e le sue potenzialità. Il collegamento delle varie parti dell'applicazione è stata una delle maggiori difficoltà che ho riscontrato, oltre al coordinamento nel lavoro di gruppo. Lo svolgimento di un progetto di questa portata ha messo in luce la difficoltà nell'organizzazione di un'architettura complessa e di conseguenza la necessità di ragionare a fondo sulle possibili soluzioni ad ogni problema. La stesura della relazione ha mostrato delle complicazioni, in quanto si tratta della nostra prima esperienza in tale campo, ma ci ha permesso di imparare le basi di LaTeX e di organizzare innanzitutto lo scheletro dell'applicazione . In conclusione considero questa esperienza costruttiva e formante per i lavori futuri.

4.2 Difficoltà incontrate e commenti per i docenti

La principale difficoltà riscontrata è stata l'organizzazione dei moduli del pattern MVC, per alcune classi non eravamo sicuri del package in cui metterle e abbiamo trascorso un tempo a nostro avviso eccessivo a cercare di capire la realizzazione e il funzionamento di un Controller. Abbiamo provato a cercare delle risposte nel materiale fornito sul sito del corso e su Internet, ma abbiamo trovato soluzioni diverse allo stesso problema, creando indecisione e confusione. Riteniamo possa essere utile per gli studenti futuri ricevere una spiegazione più dettagliata sull'organizzazione dei moduli del pattern MVC o in generale fornire delle linee guida su come collegare Model e View tramite un Controller.

Appendice A

Guida utente

All'avvio dell'applicazione l'utente verrà interfacciato con il menù principale, dove potrà scegliere tra i seguenti pulsanti:

- `PLAY`: avvia una partita.
- `QUIT`: comparirà una finestra a comparsa di conferma per uscire dall'applicazione.
- `RULES`: comparirà una finestra a comparsa dove ci saranno scritte le regole del gioco.

Una volta avviata una partita ai giocatori sarà presentata la mappa di gioco con i vari territori su un mappamondo, mentre sulla destra troveranno una barra verticale con tre pannelli: uno per visualizzare il proprio obiettivo e il colore della propria armata, uno per giocare le carte e uno per eseguire le azioni di combattimento, spostamento e fine turno.