Facoltà di Ingegneria e Scienze Informatiche

Jurassiko

Relazione di progetto

Pierfederici Edoardo Mujaj Gjovalin
Marini Giuseppe Bao Botti

18 febbraio 2024

Indice

Capitolo 1

Analisi

1.1 Requisiti

La nostra applicazione ha lo scopo di simulare una partita del noto gioco da tavolo strategico "Risiko".
L'ambientazione della mappa di gioco è ispirata all'era dei dinosauri, in cui i territori sono parti di continenti e isole della Terra durante il Giurassico.
Le regole principali sono analoghe a quelle del Risiko tradizionale. La maggiore differenza consiste nella presenza degli oceani come parte del tabellone di gioco.

- Preparazione della partita: ogni giocatore posiziona liberamente i propri Dino iniziali sui territori assegnati randomicamente.
- Rinforzi: all'inizio del turno si deve posizionare il Dino acquatico e si hanno a disposizione un numero di Dino terrestri proporzionale al numero di territori posseduti e ad eventuali continenti conquistati interamente.
- Battaglia: la parte centrale del turno che permette al giocatore di conquistare territori nemici.
- Spostamento strategico: il giocatore prima di terminare il proprio turno può scegliere se spostare alcune sue unità da un suo territorio ad un altro suo confinante.

Requisiti funzionali

- Il gioco deve occuparsi di gestire correttamente la scansione dei turni di ogni giocatore.
- La mappa deve essere configurata secondo i confini tra i territori.
- Il gioco deve gestire propriamente la scansione delle fasi di gioco e dei turni dei giocatori.

- Il gioco deve terminare correttamente nel momento in cui un giocatore completa il proprio obiettivo.
- Ciascun giocatore deve avere la possibilità di gestire le proprie fasi di gioco tramite specifici bottoni della GUI.
- Ciascun giocatore deve avere la possibilità di selezionare i territori corretti in base alla fase di gioco.
- Il giocatore potrà attaccare un numero di volte a sua discrezione purché l'assetto delle proprie truppe lo permetta.
- Il giocatore deve poter effettuare al massimo un singolo spostamento strategico per turno.

Requisiti non funzionali

- L'interfaccia utente del gioco deve essere piuttosto semplice e intuitiva, per permettere al giocatore di eseguire le sue mosse in modo immediato.
- La sequenza delle fasi di gioco e dei turni deve essere fluida per tutta la durata della partita
- L'applicazione deve poter essere eseguita correttamente nei sistemi operativi Windows, Linux e MacOS.

1.2 Analisi e modello del dominio

"Jurassiko" è un'applicazione che consente di simulare una partita tra 3 giocatori di un gioco di strategia in cui lo scopo di ciascuno è quello di raggiungere l'obiettivo contenuto nella carta pescata all'inizio del gioco.
I tre giocatori interagiscono all'interno della mappa distribuendo il proprio esercito di "Dino" tra oceani e territori e attaccando in maniera opportuna per conquistare aree del mondo nemiche.

All'inizio della partita ciascun giocatore pesca il proprio obiettivo e ottiene 7 territori casuali. Per ogni territorio posseduto, il gioco piazza automaticamente una truppa del colore del rispettivo proprietario e si ottengono 13 ulteriori dino da posizionare per rinforzare i territori desiderati.
Una volta completata la preparazione del tabellone di gioco, a turno ogni giocatore posiziona un Dino acquatico e un numero di Dino terrestri basato sull'ammontare di territori posseduti e su eventuali continenti interamente occupati.
In seguito alla fase di rinforzi, il giocatore ha la possibilità di attaccare territori nemici a proprio piacimento. La meccanica di attacco è analoga al gioco "Risiko", con una lieve semplificazione: il territorio attaccante utilizza tante truppe quante sono quelle presenti meno una (l'attaccante non può perdere il proprio territorio), con un massimo di 3; in modo equivalente il territorio difensore userà

il massimo di truppe disponibili, ovviamente anche al costo di dover perdere il territorio. Durante la fase di attacco, l'applicazione simula il lancio dei dadi in un numero corrispondente alla quantità di truppe utilizzate rispettivamente da attaccante e difensore. Gli esiti vengono poi confrontati in modo decrescente e in caso di pareggio a subire la perdita sarà il Dino attaccante. Se il territorio difensore perde tutte le sue unità in battaglia, l'esercito attaccante sposterà un massimo di 3 unità nel territorio appena conquistato.

Gli oceani possono essere sfruttati per eseguire attacchi (o spostamenti) da un continente all'altro. All'inizio di ogni turno, ciascun giocatore ha la possibilità di posizionare strategicamente il proprio Dino acquiatico. Quest'ultimo può essere usato come "ponte" per i Dino terrestri tra un continente e l'altro durante l'intero turno del giocatore. Una volta terminato il turno, il Dino acquatico viene distrutto e lo stesso giocatore potrà piazzarne uno nuovo nel suo turno successivo.

A fine turno, il giocatore ha la possibilità di effettuare al più uno spostamento di uno o più truppe da un territorio ad un altro, secondo il criterio di confini descritto precedentemente ed eventualmente esteso dal proprio Dino acquatico.

Per quanto riguarda l'implementazione del gioco, le maggiori difficoltà saranno relative alla corretta scansione dei turni dei giocatori, alla gestione delle interazioni del tabellone e ad un'adeguata interazione tra meccaniche di gioco e interfaccia grafica.

Capitolo 2

Design

2.1 Architettura

Per la realizzazione dell'applicazione "Jurassiko", il gruppo ha adottato il pattern architetturale MVC (Model-View-Controller).

Il model consiste nella definizione dei principali elementi del gioco, in particolare territori e oceani, contenenti proprietà essenziali per le interazioni dei giocatori durante la partita, come confini e continenti di appartenenza. Gli obiettivi sono divisi in tre sottoclassi fondamentali, che ne definiscono il tipo (conquista di continenti, conquista di territori e distruzione di eserciti nemici). Questi tre tipi di oggetti non possono essere istanziati singolarmente, ma vengono creati tramite apposite classi Factory che li creano il blocco. Sarà poi compito del controller dividerli e assegnarli opportunamente ai singoli giocatori.

L'interfaccia grafica è realizzata con il framework di Java *Swing* e consiste in un menu minimale da cui può essere avviato il gioco. L'applicazione vera e propria consiste in un frame principale composto da due panel: uno dedicato al tabellone di gioco, e l'altro ai bottoni per gestire le fasi di ciascun giocatore, oltre alle informazioni principali per il turno corrente.

Il controller è suddiviso in due package principali all'interno della directory del progetto: *core* e *controller* vero e proprio. Il primo contiene l'implementazione di game engine e game loop, con la gestione dei turni e delle fasi di gioco, oltre al controllo della condizione di fine partita in funzione degli obiettivi; il secondo contiene la classe centrale dell'applicazione, *MainController*, che ha il ruolo di gestire la comunicazione tra view e model, oltre quello di processare le azioni da compiere in base alla fase di gioco, e *MenuController*, che permette di avviare l'applicazione mediante il caricamento della schermata di menu e l'avvio del game loop una volta avviata la partita.

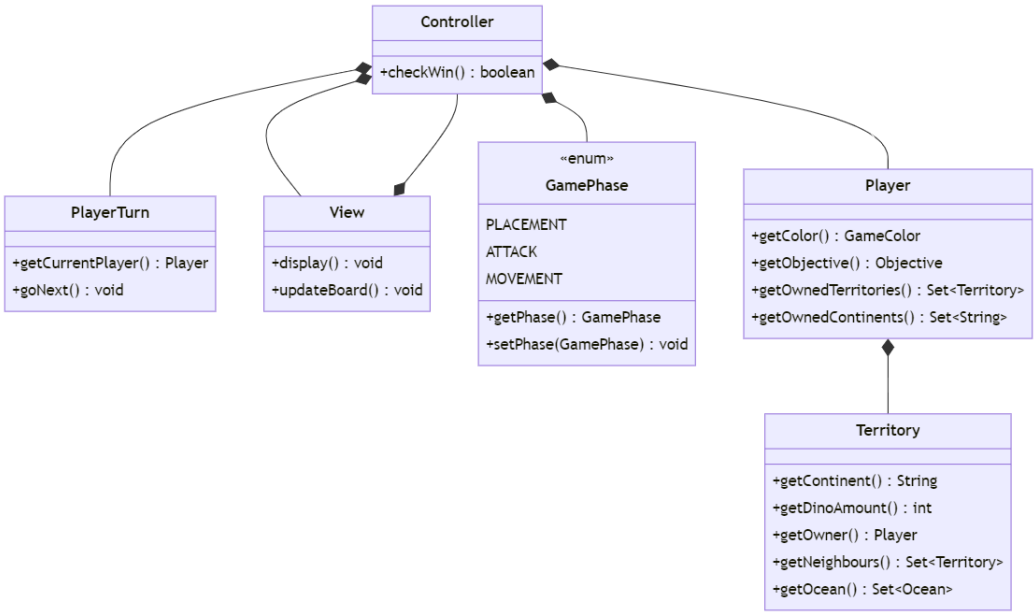

Figura 2.1: Architettura generale dell'applicazione

2.2 Design dettagliato

Pierfederici Edoardo

Lettura da file degli oggetti base del gioco

Problema Gli elementi di base del gioco (obiettivi, territori e oceani) devono essere definiti in specifici file che l'applicazione deve leggere ed inizializzare.

Soluzione Creazione di file JSON per la configurazione degli obiettivi, divisi per tipo, e di territori e oceani, compresi i loro rispettivi confini.
Utilizzo della libreria Jackson per il parsing dei file: uso dei generici e della classe *BoardDataReader*, nella quale il metodo readValue dell'ObjectMapper di Jackson sfrutta l'implementazione delle classi Territory, Ocean e Objective. Nel caso della posizione degli sprite nella GUI si è invece utilizzata l'API "ad albero" (*readTree*) dell'ObjectMapper di Jackson.

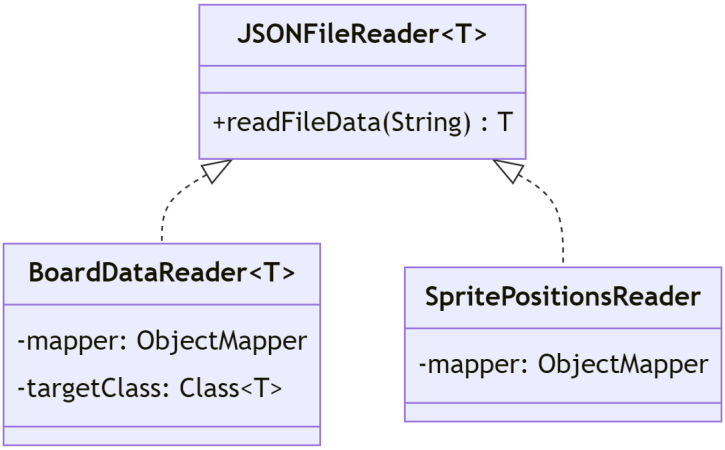

Figura 2.2: Diagramma UML di JSONFileReader

Creazione di territori e oceani

Problema È necessario mantenere traccia per ogni giocatore dei territori posseduti e degli oceani.

Soluzione Con l'aiuto del collega Mujaj, sono state create delle classi utilizzate da *BoardDataReader* per deserializzare i file JSON corrispondenti. In questo modo, una volta creati gli oggetti *Territory* e suddivisi a inizio partita tra i 3 giocatori, i metodi del controller li gestiscono per mantenerne traccia quando

vengono conquistati da altri giocatori, insieme al numero di truppe per ciascun area della mappa.

Creazione di obiettivi

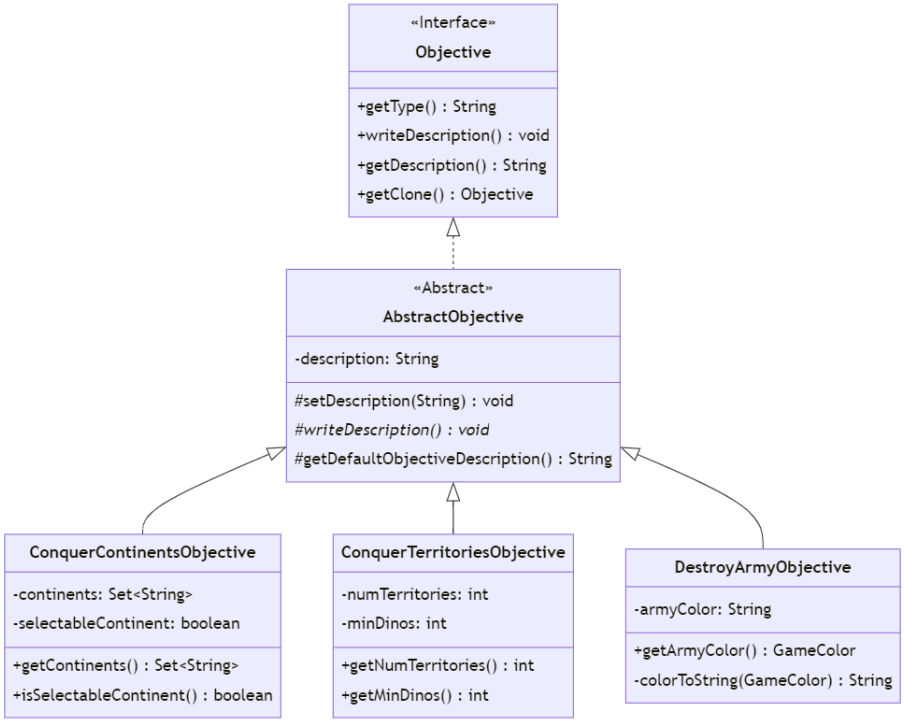

Figura 2.3: Diagramma UML di Objective

Problema Gli obiettivi devono essere suddivisi per tipo e devono memorizzare le informazioni necessarie.

Soluzione Creazione di una classe astratta *AbstractObjective* che contiene i metodi comuni a tutti gli obiettivi, inclusa la creazione della descrizione e la possibilità di settare quest'ultima per essere inserita nella carta obiettivo della GUI e di ottenere una copia dell'istanza obiettivo. Creazione delle sottoclassi *ConquerContinentsObjective*, *ConquerTerritoriesObjective*, *DestroyArmyObjective* per trasmettere le informazioni specifiche per ogni tipo al resto dell'applicazione.

Posizionamento corretto degli sprite nel tabellone di gioco

Problema All'interno del tabellone di gioco, tutti i giocatori devono visualizzare il proprietario e il numero di unità presenti in ciascun territorio.

Soluzione Creazione di un file JSON contenente le coordinate degli sprite di ogni territorio e oceano in proporzione alla dimensione dello schermo. Questo file viene letto grazie alla classe *SpritePositionsReader*, che sfrutta la libreria Jackson.

Interazione con i territori

Problema Durante la partita, i giocatori devono poter selezionare i territori in accordo con la fase di gioco corrente.

Soluzione Creazione di un JFrame che utilizza un GridBagLayout nel quale sono presenti tanti bottoni quanti sono territori e oceani. La parte di controller gestisce in base alla fase di gioco e al giocatore corrente quali bottoni devono essere attivati e quali disattivati.

Gestione della fase di spostamento strategico

Problema Il giocatore deve poter spostare un numero di unità da un suo territorio ad un altro alla fine del proprio turno.

Soluzione Utilizzo della schermata pop-up *TerritorySelector* per selezionare il territorio (i bottoni sono attivati/disattivati grazie al game engine) e memorizzazione del primo per controllare che lo spostamento sia valido. Tramite un JSpinner il giocatore può modificare l'ammontare di Dino da spostare.

Controllo della condizione di vittoria

Problema L'applicazione deve mostrare il vincitore e terminare nel momento in cui uno dei giocatori completa il proprio obiettivo.

Soluzione Creazione di una classe specifica *WinCondition*, che in base al tipo di obiettivo del giocatore ne determina la condizione per essere completato. Il *MainController* verifica periodicamente questa condizione e in caso sia vera, mostra la schermata di fine gioco.

Mujaj Gjovalin

Aggiornamento dello stato dei territori

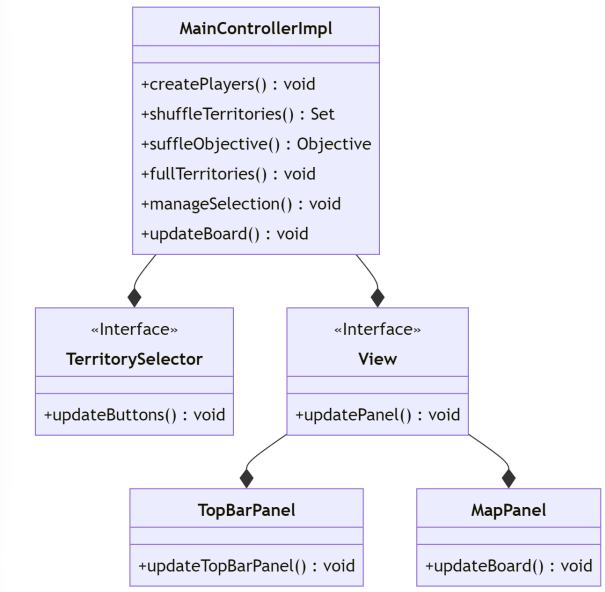

Figura 2.4: Diagramma UML dell'interazione Controller-View

Problema Assegnazione per ogni territorio un colore e un numero di Dino e aggiornamento di essi.

Soluzione Ho inizializzato dentro il *MainController* una Map con all'interno il territorio utilizzato come chiave, un Pair con all'interno il colore e l'ammontare di Dino utilizzati come valore. Andando a creare i Player con i relativi metodi in modo randomizzato, possiamo metterli dentro la mappa. Così facendo, riusciamo ad aggiornare sia l'ammontare di Dino che il colore di essi, per il territorio corrispondente e attraverso l'updateBoard aggiorniamo i button del territory selector e il main frame.

Immagini scalabili

Problema Scalabilità delle immagini all'interno della GUI.

Soluzione All'interno della classe *ViewImpl*, per rendere le immagini all'interno della GUI scalabili, ho creato un metodo apposito, chiamato getScreenSize, che ritorna un Dimension, da cui andiamo a prendere l'altezza e la larghezza

e attraverso il metodo scaleImage andiamo a scalare l'immagine nel modo più ottimale.

Selezione dei confini

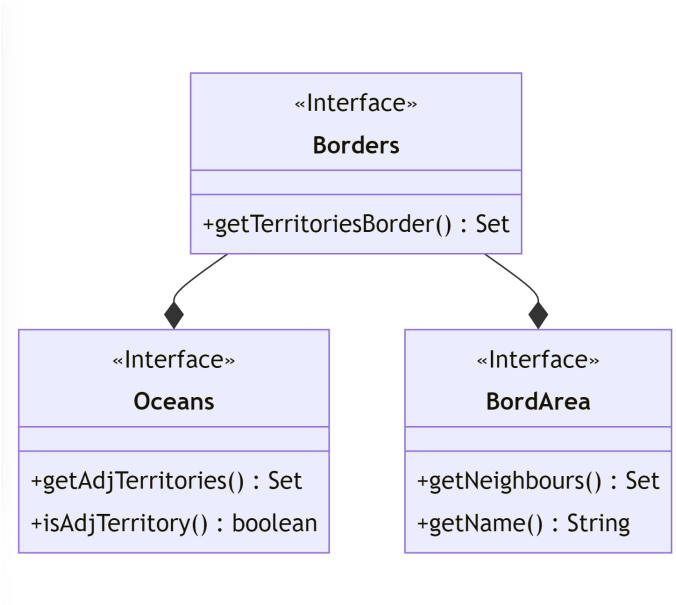

Figura 2.5: Diagramma UML di Objective

Problema Andare a selezionare tutti i confini per ogni territorio passato con o senza oceano.

Soluzione Ho creato un metodo apposito che utilizza i metodi implementati nei territori getNeighbourNames e getAdjTerritoryNames per selezionare i confini del territorio e dell'oceano passati in input e vado a ritornare un Set di stringhe con tutti i confini del territorio interessato.

Pannello dei bottoni

Problema Inserimento di una Top Bar con all'interno i bottoni distanziati correttamente. Crea una griglia dove tu specifichi 2 variabili gridx e gridy dove specifichi la posizione nella griglia del componente.

Soluzione Ho creato una JLabel che utilizzava un GridBagLayout per andare a specificare tramite le 2 variabili passate in input (gridx e gridy) la posizione nella griglia del componente e tramite un Insents per andare a distanziarli nel

metodo corretto. E infine tramite un addComponent aggiungo il bottone e la griglia che va a settare il bottone nella posizione adatta.

Marini Giuseppe

Rappresentazione grafica dei dinosauri

Problema Caricamento degli sprite dei dino per essere mostrati

Soluzione Creazione di una classe che andasse a caricare gli sprite in una mappa per tipologia di dino (di terra e marini), li caricasse in dei jlayeredpane per poi caricarli in dei Jframe. Si e' pero' notato che questa soluzione andava a intaccare sensibilmente le prestazioni del programma poiche' gli sprite venivano caricati in memoria ogni volta che doveva essere impostato un JlayerdPanel. Per questo si e' optato per la creazione di una classe a parte che andasse a riempire le mappe dei dino e abbiamo aggiunto dei metodi che ne ritornassero delle copie gia' riempite.

Gestione meccaniche di attacco e difesa

Problema Calcolare il numero di truppe perso nella battaglia che segua le regole del gioco.

Soluzione Creazione di una classe con all'interno un metodo che vada a creare 2 liste riempite tramite il metodo che lancia i dadi, controlli quale sia il numero minore tra le truppe schierate e confronti i primi elementi delle liste, ripetendo questo calcolo un numero di volte calcolato in precedenza e tornando una pair con i risultati.

Sprite all'interno dei territori

Problema Mostrare i dino sopra la mappa con il numero di dino in quel territori

Soluzione Creazione di una classe che estende un JPanel che carica la mappa del tipo giusto tramite un boolean configura una JLabel (tramite le dimensioni dello sprite) dove ci andrà quest'ultimo e la label del contatore sempre basata sulle stesse dimensioni. Queste label vengono aggiunte a un JLayeredPane e poi aggiunte al panel di questa classe. Poi nella classe della mappa tramite cicli che utilizzano gli stream si vanno a caricare i JPanel.

Bao Botti

Gestione dei turni e della fase di gioco

Problema In ogni Turno di un singolo giocatore si suddivide in diverse fasi, in ogni fase di gioco al giocatore è permesso solo di eseguire determinate azioni.

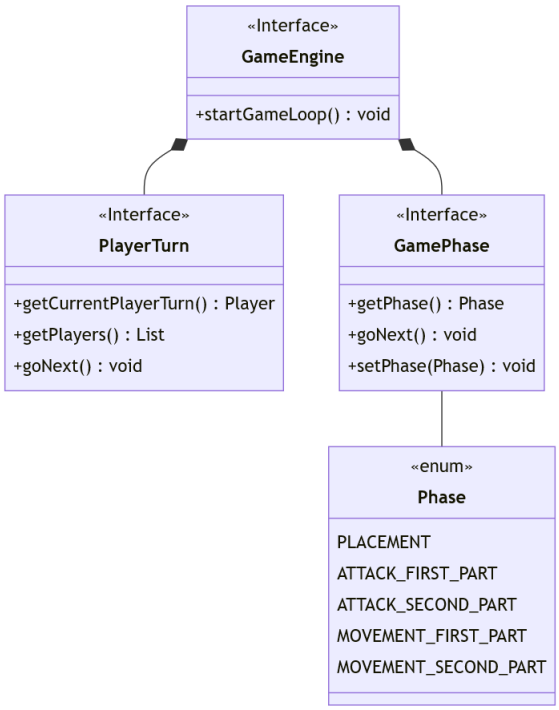

Figura 2.6: Diagramma UML di GameEngine, PlayerTurn e GamePhase

Soluzione Creazione di due classi individuali: PlayerTurn che gestisce solo il cambiamento del turno del giocatore e GamePhase che gestisce solo il cambiamento della fase del gioco. Le due classi sono indipendenti una rispetto all'altra e vengono istanziate all'interno del GameEngine, che viene a sua volta utilizzato per cambiare o passare a una fase del gioco a un'altra o al giocatore successivo.

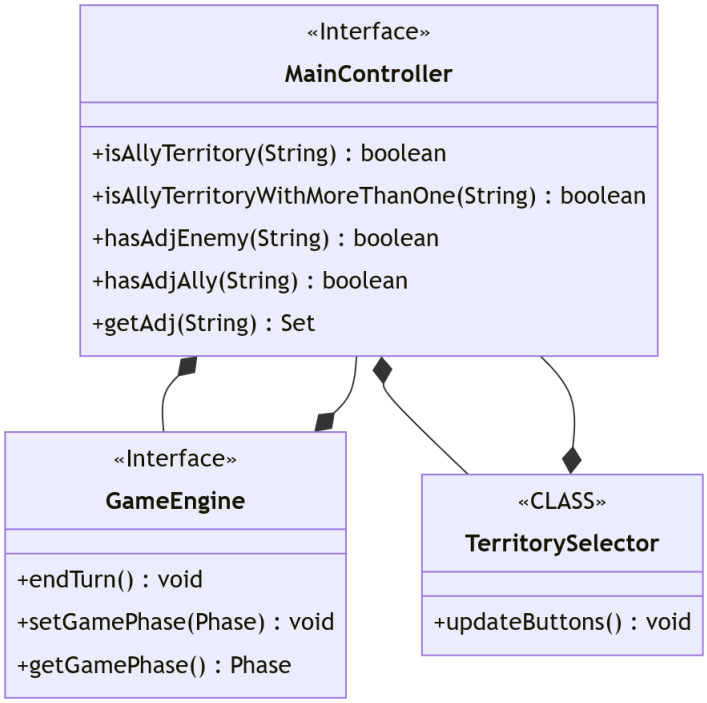

Figura 2.7: Diagramma UML di TerritorySelector

Gestione della corretta visualizzazione dei territori selezionabili

Problema Ogni giocatore deve essere solo in grado selezionare i territori corretti consentiti, dipendentemente alla fase del gioco.

Soluzione Disattivazione di tutti i bottoni che non devono essere selezionati in un determinato turno, ad ogni cambio di fase o del turno del giocatore, il MainController prende la fase del gioco dal GameEngine, e chiama il metodo per aggiornare i bottoni del TerritorySelector, quindi display dei bottoni cliccabili cambia rispettando il giocatore e della fase corrente.

Lettura da File del MenuPanel

Problema Lettura da file di immagine e file di testo.

Soluzione Per la lettura da file dell'immagine è stato creato un BufferedImage, in modo tale da poter modificare la grandezza dell'immagine, invece per la lettura da file di testo si è usato un BufferedReader con all'interno un InputStreamReader.

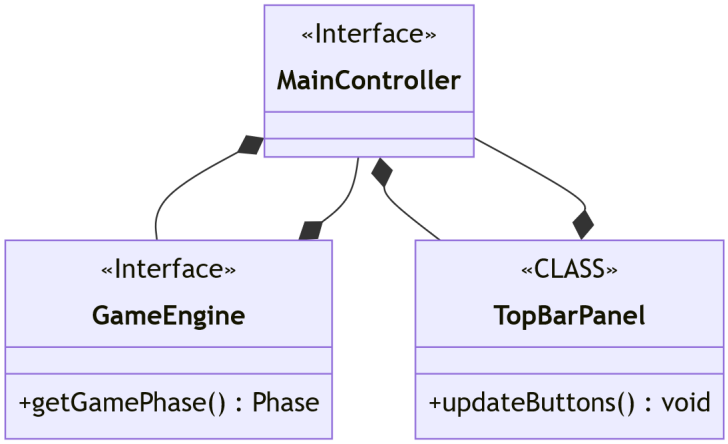

Figura 2.8: Diagramma UML di TopBarPanel

Appropriata attivazione/disattivazione dei bottoni

Problema Inizialmente tutti i bottoni erano cliccabili, ma alcuni devono essere disattivati in base alla fase di gioco.

Soluzione Il *TopBarPanel*, contenente i bottoni per le azioni di gioco, ottiene tramite il MainController, che ha un istanza del GameEngine, la fase del turno e in base ad esse determinati bottoni vengono disabilitati.

Gestione della fase di piazzamento del primo turno

Problema Prima di iniziare il gioco si ha una quantità predefinita di dino da piazzare, che valgono soltanto per la preparazione della partita.

Soluzione Nel GameEngine, classe responsabile per la gestione dei turni e fasi, esiste una variabile booleana che controlla se si è ancora nel primo turno o meno. In caso questa fosse vera, allora, una volta terminata piazzate le truppe, le altre fasi del giocatore non saranno disponibili.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il corretto funzionamento delle singole meccaniche dell'applicativo è testato in modo automatizzato mediante il noto framework di testing JUnit 5.
Di seguito una breve descrizione di ciascuna classe di test:

- TestBattle: testa la correttezza dei risultati del metodo BattleImpl tramite la verifica il conteggio delle morti per ogni fazione sia minimo o uguale a 0 e massimo o uguale al numero di truppe che hanno combattuto, questo nel primo test. Nel secondo vado a testare il metodo per calcolare quanti dino devono combattere o in caso di conquista del territorio quanti ne devono essere spostati.
- TestBorder: verifica che per ogni territorio passato in input vengano restituiti i confini corretti, considerando la presenza o meno dell'oceano.
- TestDice: questa classe si occupa di controllare se output del dado sia nei limiti permessi.
- TestGamePhase: questa classe si occupa di controllare se il passaggio di una fase del gioco all'altra sia corretto e ciclico.
- TestObjective: verifica la corretta lettura da file degli obiettivi di gioco, tra cui valori specifici e descrizione.
- TestOcean: verifica la corretta lettura da file degli oceani controllando il numero totale, i confini e i territori adiacenti.
- TestPlayer: questa classe si occupa di controllare il corretto funzionamento dell'aggiunta e della rimozione di territori al giocatore, la corretta creazione del duplicato del giocatore e il corretto calcolo dei dino bonus in base ai territori e continenti posseduti.

- TestPlayerTurn: questa classe si occupa di controllare se il passaggio di turno tra un giocatore e l'altro sia corretto e ciclico.
- TestSpritePositionReaders: verifica la corretta lettura da file delle coordinate degli sprite per l'interfaccia grafica del tabellone.
- TestTerritory: verifica la corretta lettura da file dei territori controllando il numero di territori per continente e i singoli confini.
- TestWinCondition: verifica le condizioni di vittoria per ogni tipo di obiettivo.

3.2 Metodologia di lavoro

Il gruppo ha iniziato il lavoro definendo le regole principali del gioco, in particolare stabilendo le varianti del gioco rispetto all'originale gioco da tavolo "Risiko". In seguito si è passati ad immaginare l'aspetto del tabellone di gioco e un primo approccio all'architettura del progetto dal punto di vista del codice, cercando quindi di costruire uno schema UML generale.
Prima di iniziare a scrivere codice, sono state fissate delle "convenzioni" interne riguardo al workflow di Git per il controllo versione: si è deciso di mantenere un branch principale (*main*), sempre testabile e compilabile e di aprire *feature branch* per ogni nuova aggiunta di ciascuno o eventuali futuri bug fix che avrebbero richiesto una cospicua modifica del codice. I feature branch, una volta completati e testati con successo, sono stati integrati al main utilizzando il sistema di *pull request* e *merge* integrato in GitHub.
In generale, nel corso dello sviluppo, ciascun membro ha lavorato individualmente sulle proprie parti assegnate nella proposta del progetto (sebbene siano state apportate alcune leggere modifiche per motivi di comodità). In alcuni casi, tuttavia, è stato necessario riunirsi, anche se a distanza, per organizzare e definire insieme alcune sezioni di progettazione di maggiore importanza e complessità.

Seguono le parti di lavoro svolte da ognuno dei membri del gruppo:

Pierfederici Edoardo

- Lettura da file JSON (Jackson) per territori, oceani, obiettivi e posizioni degli sprite nella GUI
- Creazione oggetti obiettivo
- Creazione carta obiettivo
- Creazione di finestre informative (ad esempio vincitore, esito battaglia)
- Gestione dello spostamento di fine turno
- Condizione di vittoria
- Gestione dello strumento di logging e creazione del file logback

Mujaj Gjovalin

- Creazione del file map.png con la mappa di gioco.
- Creazione del tabellone di gioco con i rispettivi nomi dei territori.
- Implementazione di territori e oceani
- Creazione della barra superiore e dei pulsanti d'azione.
- Implementazione del controller principale (MainController), inizializzando la fase iniziale del gioco (fase di piazzamento) e l'update della mappa.
- Controllo dei confini di ogni territorio considerando l'eventuale presenza dell'oceano
- Creazione del file con il regolamento

Marini Giuseppe

- Rappresentazione grafica degli sprite dei Dino
- Implementazione della battaglia
- Gestione delle meccaniche di attacco e difesa
- Classe di caricamento delle immagini per gli sprite
- Visualizzazione all'interno della GUI degli sprite

Bao Botti

- Implementazione del modello dei dadi
- Implementazione del modello dei giocatori
- Creazione del menu di gioco
- Implementazione del game engine
- Gestione dei turni e delle fasi di gioco
- Aggiornamento dei bottoni della GUI in base alla fase di gioco

3.3 Note di sviluppo

Pierfederici Edoardo

Utilizzo di Lambda Expressions e Stream

L'uso di paradigmi della programmazione funzionale è piuttosto diffuso in buona parte delle classi dell'applicazione e ha permesso una maggiore flessibilità e pulizia del codice. Di seguito alcuni esempi:

- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blob/0d2785912c316f3c63f364de8b8921b2d7d89a58/src/main/java/it/unibo/jurassiko/core/impl/WinConditionImpl.java#L41C4-L46C6
- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blob/41f16ba62baa6eef6156748f6a5b8fe37c3e739f/src/main/java/it/unibo/jurassiko/core/impl/WinConditionImpl.java#L79C9-L97C83

Utilizzo della libreria Jackson

- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blob/66395acbf3464bbf2496c4e5eae769fdf03a0301/src/main/java/it/unibo/jurassiko/reader/impl/BoardDataReader.java#L38C5-L50C6
- Permalink:https://github.com/Dodop12/OOP23-jurassiko/blob/66395acbf3464bbf2496c4e5eae769fdf03a0301/src/main/java/it/unibo/jurassiko/reader/impl/SpritePositionsReader.java#L35C5-L51C6

Utilizzo di Optional

- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blob/0b803cfe26fb7a6a10dabedd6452a242f3dac04c/src/main/java/it/unibo/jurassiko/core/impl/WinConditionImpl.java#L69C9-L71C10

Utilizzo del pattern Factory

- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blob/main/src/main/java/it/unibo/jurassiko/model/territory/impl/OceanFactoryImpl.java

Utilizzo della libreria SLF4J per logging

- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blob/0b803cfe26fb7a6a10dabedd6452a242f3dac04c/src/main/java/it/unibo/jurassiko/Jurassiko.java#L27C9-L27C45
- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blob/0b803cfe26fb7a6a10dabedd6452a242f3dac04c/src/main/java/it/unibo/jurassiko/core/impl/GameEngineImpl.java#L50C13-L50C72

Mujaj Gjovalin

Utilizzo di Lambda Expressions e Stream

- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blame/main/src/main/java/it/unibo/jurassiko/controller/impl/MainControllerImpl.java#L506-L511

- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blob/main/src/main/java/it/unibo/jurassiko/controller/impl/MainControllerImpl.java#L477-L500

Utilizzo dello switch con Lambda Expressions

- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blob/main/src/main/java/it/unibo/jurassiko/view/panels/TopBarPanel.java#L153-L160

Marini Giuseppe

Utilizzo di Lambda Expressions e Stream

- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blob/0b803cfe26fb7a6a10dabedd6452a242f3dac04c/src/main/java/it/unibo/jurassiko/view/panels/MapPanel.java#L68C9-L74C1
- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blob/0b803cfe26fb7a6a10dabedd6452a242f3dac04c/src/main/java/it/unibo/jurassiko/view/panels/MapPanel.java#L100C5-L114C6

Bao Botti

Utilizzo di Lambda Expressions e Stream

- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blob/main/src/main/java/it/unibo/jurassiko/view/panels/MenuPanel.java#L106-L109

Utilizzo di metodi generici

- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blob/main/src/main/java/it/unibo/jurassiko/view/windows/TerritorySelector.java#L287-L293
- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blob/main/src/main/java/it/unibo/jurassiko/view/windows/TerritorySelector.java#L422-L429

Utilizzo di Optional

- Permalink: https://github.com/Dodop12/OOP23-jurassiko/blob/main/src/main/java/it/unibo/jurassiko/view/windows/TerritorySelector.java#L205-L207

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Pierfederici Edoardo

La realizzazione di questo progetto è stata un'esperienza stimolante e particolarmente formativa. Personalmente, avendo iniziato a studiare informatica a livello didattico soltanto l'anno scorso con l'inizio di questo corso universitario, non avevo mai sviluppato un software così complesso. Realizzare questo gioco in maniera così rigorosa, a partire da proposta, dominio e analisi, fino ad arrivare all'implementazione vera e propria, la risoluzione di bug e la gestione del gruppo dall'inizio alla fine mi ha permesso di capire meglio la materia di sviluppo software anche da un punto di vista vicino a quello aziendale. Credo di aver notevolmente migliorato sia le mie skill da programmatore, in particolare riguardo al linguaggio Java, sia le mie skill di teamwork. Inoltre ho imparato ad utilizzare in maniera più avanzata strumenti tecnici quali Git e GitHub con i loro workflow, Gradle. In generale, mi ritengo soddisfatto del lavoro svolto e dell'organizzazione con il resto del gruppo. Tutti i membri del team si sono rivelati collaborativi e hanno mostrato un notevole impegno anche nei momenti di difficoltà. In futuro mi piacerebbe sicuramente partire da questo progetto per inserire nuove funzionalità oppure creare una nuova applicazione ancora più complessa.

Mujaj Gjovalin

La creazione di questo progetto è stato qualcosa di molto stimolante, inizialmente con tutte le difficoltà del caso, ma con: il lavoro di gruppo, l'organizzazione complessiva, condivisione di idee e opinioni abbiamo realizzato una versione del risiko particolare. Sicuramente ciò che abbiamo fatto ci ritornerà molto utile per il futuro, essendo che per la prima volta ci siamo cimentati in qualcosa, a noi sconosciuto. I vari controlli a noi forniti (CheckStyle, SpotBugs, PMD e il Build Tool di Gradle) mi sono stati molto utili, per la creazione di un codice più pulito

e performante. Con l'utilizzo di GitHub siamo riusciti a coordinarci e lavorare nel modo più ottimizzato possibile. In un futuro prossimo, vorrò certamente approfondire questo campo e migliorare il mio stile di programmazione.

Marini Giuseppe

Sono molto soddisfatto del progetto. Mi ha aiutato molto a rendermi conto di alcune lacune che avevo e a risolverle grazie all'aiuto degli altri. Questo mi ha fatto capire pure l'importanza del lavoro di gruppo e del sapersi organizzare in un progetto di questo genere, contando che era il mio primo che facevo. Inoltre mi aiutato a capire l'importanza dell'analisi e dell'organizzazione iniziale del e mi ha dato un'idea generale di come potrebbe essere il lavoro in ambito. Molto utili sono stati gli strumenti forniti dal docente attraverso Gradle per scrivere codice chiaro e pulito, molto importante quando altra gente deve leggere il tuo codice. La conclusione di questo progetto è stata cruciale per il mio percorso di studi, poiché mi ha fornito una base solida e migliorato il mio approccio alla programmazione.

Bao Botti

Mi ritengo soddisfatto del progetto. Questo è stato il mio primo lavoro di gruppo nell'ambito informatico, perciò ho incontrato molte difficoltà nel iniziare da zero, siccome dall'inizio non si ha un'immagine chiara del progetto finale, ma confrontandosi gli altri e chiarendo le idee il problema si alleggerisce. Ciò che ho ritenuto molto utile era la build di Gradle, con i CheckStyleMain, SpotBugsMain e PMDmain ho corretto molti miei errori che avrei solitamente ignorato.

4.2 Difficoltà incontrate

Se da un lato tutti noi eravamo entusiasti e desiderosi di affrontare questo progetto, inizialmente non è stato semplice gestire l'organizzazione e costruire un'idea di dominio e struttura a livello di interfacce e classi. Durante lo sviluppo del gioco ci siamo accorti che si trattava di un tipo di lavoro totalmente diverso da quelli affrontati durante il corso, per cui siamo stati costretti a chiarirci da soli molti dei concetti utili per lo sviluppo.
Anche se buona parte del lavoro è stata svolta individualmente, durante l'implementazione del Controller (soprattutto le interazioni all'interno dell'architettura MVC), che è stata la parte in cui abbiamo trovato maggiore difficoltà, siamo stati costretti ad aiutarci a vicenda e a sviluppare diversi blocchi di codice in modo collettivo.

Appendice A

Guida utente

Menu iniziale

Il menu che si apre avviando l'applicazione è composto da tre pulsanti:

- **Play**: avvio della partita
- **Quit**: chiusura dell'applicazione
- **Rules**: apertura della schermata delle regole

Partita

Una volta premuto il tasto "Play", appare la schermata di gioco vera e propria, divisa in due sezioni principali: il tabellone di gioco e il pannello superiore. Inoltre, in base alla fase di gioco, è presente una schermata pop-up che permette al giocatore di interagire con i territori e gli oceani del tabellone. Il tabellone di gioco mostra la mappa completa dei territori e degli oceani che sono inizialmente assegnati ai vari giocatori in modo randomico. Il pannello superiore mostra in alto a sinistra il colore del giocatore che sta effettuando il turno e il numero di truppe rimaste da posizionare. La parte restante del pannello superiore è formata dai seguenti pulsanti:

- **Obiettivo**: mostra la carta obiettivo del giocatore corrente.
- **Piazzamento**: entra nella prima fase del turno che consiste nel piazzare i propri Dino nei territori desiderati.
- **Attacco**: permette di attaccare territori nemici. Inizialmente il giocatore seleziona il proprio territorio da cui desidera sferrare l'attacco e subito dopo il territorio nemico da attaccare.

- **Fine Turno**: permette al giocatore di terminare la propria fase di attacco. Il giocatore può scegliere se effettuare lo spostamento strategico oppure se terminare il proprio turno.

Figura A.1: Il pannello superiore a inizio partita

Appendice B

Esercitazioni di laboratorio

B.1 Pierfederici Edoardo

- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=148025#p210363
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=149231#p211573
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212716

B.2 Mujaj Gjovalin

- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212728

B.3 Bao Botti

- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=148025#p210284
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212692
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=151542#p213935