Relazione per "Risiko"

Michele Farneti: 0001080677
Anna Malagoli: 0001070926
Keliane Nidele Nana: 0001100749
Manuele D'Ambrosio: 0001080182

25 luglio 2024

Indice

Capitolo 1

Analisi

1.1 Requisiti

L'Applicazione si pone l'obiettivo di creare una versione digitale del celebre gioco da tavolo Risiko. Rispetto al classico gioco sono state effettuate delle modifiche marginali al regolamento in modo tale da rendere il gioco più immediato e più godibile. Sono state inoltre aggiunte nuove funzionalità, come la possibilità di selezionare una mappa diversa da quella classica e la possibilità di giocare insieme a giocatori virtuali controllati dall'applicazione.

Requisiti funzionali

- **Inizializzazione**
  L'utente che vuole iniziare a giocare dovrà poter personalizzare la propria partita attraverso la scelta della mappa e del numero di giocatori standard o AI.

- **Schermata di gioco**
  La schermata di gioco deve essere in grado di mostrare in tempo reale tutte le modifiche effettuate sui territori dovute alle scelte dei giocatori. Dovrà inoltre essere esplicita la fase di gioco in cui ci si trova in modo tale che il giocatore sia consapevole delle scelte a sua disposizione in quel momento della partita.

- **Selezione mappa** L'applicazione deve consentire la selezione della mappa di gioco al momento della creazione della partita.

- **Interazioni con la mappa**
  L'utente dovrà aver l'opportunità di compiere azioni di gioco che con-

templano la selezione di un territorio mediante il solo click della sua rappresentazione sulla mappa.

- **Esperienza di gioco guidata**
  L'esperienza di gioco dell'utente nelle proprie fasi di gioco dovrà essere indirizzata per evitare che esso si trovi a compiere azioni non coerenti col corretto svolgimento di una partita

- **Fasi di gioco semi-automatiche**
  Il giocatore dovrà aver l'opportunità di godere di un esperienza di gioco con un ridotto numero di casistiche in cui si ritrovi a compiere azioni obbligate.

- **Bot**
  I giocatori virtuali, i quali saranno presenti soltanto a discrezione dell'utente, dovranno poter svolgere le stesse decisioni di un vero giocatore, seppur in modo (per forza di cose) semplificato.

- **Strumenti di monitoraggio**
  Il giocatore dovrà aver l'opportunità di usufruire di strumenti per monitorare lo stato della partita e le azioni compiute dagli altri giocatori nei turni precedenti.

- **Salvataggio**
  Il giocatore avrà la possibilità si salvare lo stato corrente della partita, in questo modo la partita potrà essere interrotta e ripresa in un successivo momento.

Requisiti non funzionali

- **Risoluzione**
  Dovrà essere possibile selezionare una determinata risoluzione per la finestra di gioco tra quelle disponibili.

- **Compatibilità**
  L'applicativo deve garantire la possibilità di essere eseguito sia su piattaforme Windows che Linux.

1.2 Analisi e modello del dominio

Ogni partita conserva lo stato attuale della mappa, la quale è composta da un insieme di territori. Ogni territorio ha un giocatore proprietario e un

numero di armate posizionate su di esso. Ogni giocatore ha un obiettivo che deve completare per vincere e possiede inoltre un certo numero di carte che sono state pescate dal deck durante il corso della partita e che possono essere utilizzate per ottenere dei bonus. Ogni qual volta un giocatore compie un'azione questa azione viene memorizzata come evento in un registro degli eventi.

L'inizializzazione della partita permette la selezione della mappa con relativa implementazione di un ambiente di gioco personalizzato per quanto riguarda obiettivi, territori, deck e numero massimo di giocatore. L'unità fondamentale alla base del gioco è il territorio, il cui ruolo è quello di essere assegnato ai giocatori e di essere occupato da armate. In una mappa di gioco i territori sono raggruppati in continenti (tale caratteristica verrà approfondita in seguito) ed i giocatori se li contenderanno al fine di raggiungere il proprio obiettivo. Le tipologie di giocatori di cui si compone una partita possono essere:

- **Standard**
  Le cui azioni sono decise dall'utente.
- **Bot**
  Le cui azioni sono gestite in automatico dal gioco.

Essi disporranno di una propria personale collezione di territori e di un proprio arsenale, formato da un numero di armate posizionabili su di essi a propria discrezione. I giocatori agiranno con il fine di raggiungere il proprio obiettivo, che può essere di tre tipologie: L'inizializzazione della partita si occuperà, per ciascun giocatore, di generare uno tra tali obiettivi, di calcolare e di assegnare un numero di armate coerente al numero di sfidanti e di suddividere il totale dei territori tra i giocatori, provvedendo anche all'occuparli con almeno un'armata. Verranno inoltre mischiate le carte del deck per preparalo a successive pescate e sarà stabilito un ordine casuale per l'alternanza dei turni dei giocatori. I primi turni prevedono l'alternanza dei giocatori al fine di occupare i territori fino all'azzeramento delle armate nell'arsenale di ciascuno. Terminata tale fase inizierà l'alternanza dei turni del giocatore, che potrà compiere azioni quali:

- **Posizionamento di truppe su territori in suo possesso**
- **Attacco a territori avversari**
- **Spostamento di armate tra territori posseduti adiacenti**

- **Gestione delle carte al fine di ottenere bonus**

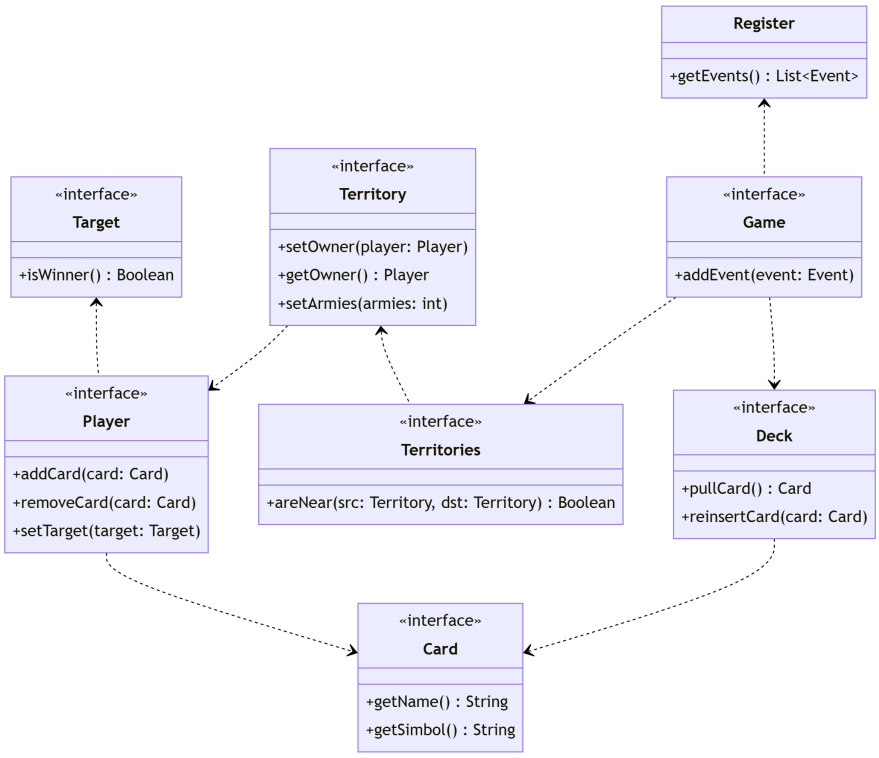

Figura 1.1: Schema UML dell'analisi del problema con entità principali

Capitolo 2

Design

2.1 Architettura

L'applicazione di **Risiko** segue il modello **MVC** visto a lezione con la differenza che il **controller** comunica con due interfacce di **view** separate, una per la schermata iniziale e l'altra per la schermata di gioco (Figura 2.1). Inoltre il **controller** utilizza tutte le classi principali del model (Figura 2.2) e ne chiama i metodi controllando ereindirizzando cosí gli input ricevuti dall'utente per mezzo dell'interfaccia interfaccia grafica. Il diagramma **UML** del modello **MVC** è stato suddiviso in due parti, una parte (Figura 2.1) mostra le relazioni del controller con la view, mentre l'altra parte (Figura 2.2) rappresenta le relazioni del **controller** con le classi del model.

Il **controller** comunica con la **view** mediante due interfacce separate e indipendenti. L'interfaccia **InitialView** fa riferimento al menù iniziale del gioco e consente di cambiare le impostazioni di risoluzione e iniziare una nuova partita. Questa schermata è di fatto facoltativa e non fondamentale, la sua rimozione non impedirebbe al gioco di funzionare correttamente. L'interfaccia **GameView** è invece relativa alla schermata principale di gioco, consente infatti al **controller** di modificare la **view** e al tempo stesso passarle eventuali parametri necessari per il suo aggiornamento. Le interfacce **Observer** ossia **InitialViewObserver** e **GameViewObserver** servono alle due **view** per comunicare al **controller** le decisioni prese dal giocatore permettendo così la prosecuzione del gioco. Cambiando l'implementazione della **view** senza però modificare le interazioni con le interfacce non comporta la necessità di modifiche sul **controller** per mantenere l'applicazione funzionante.

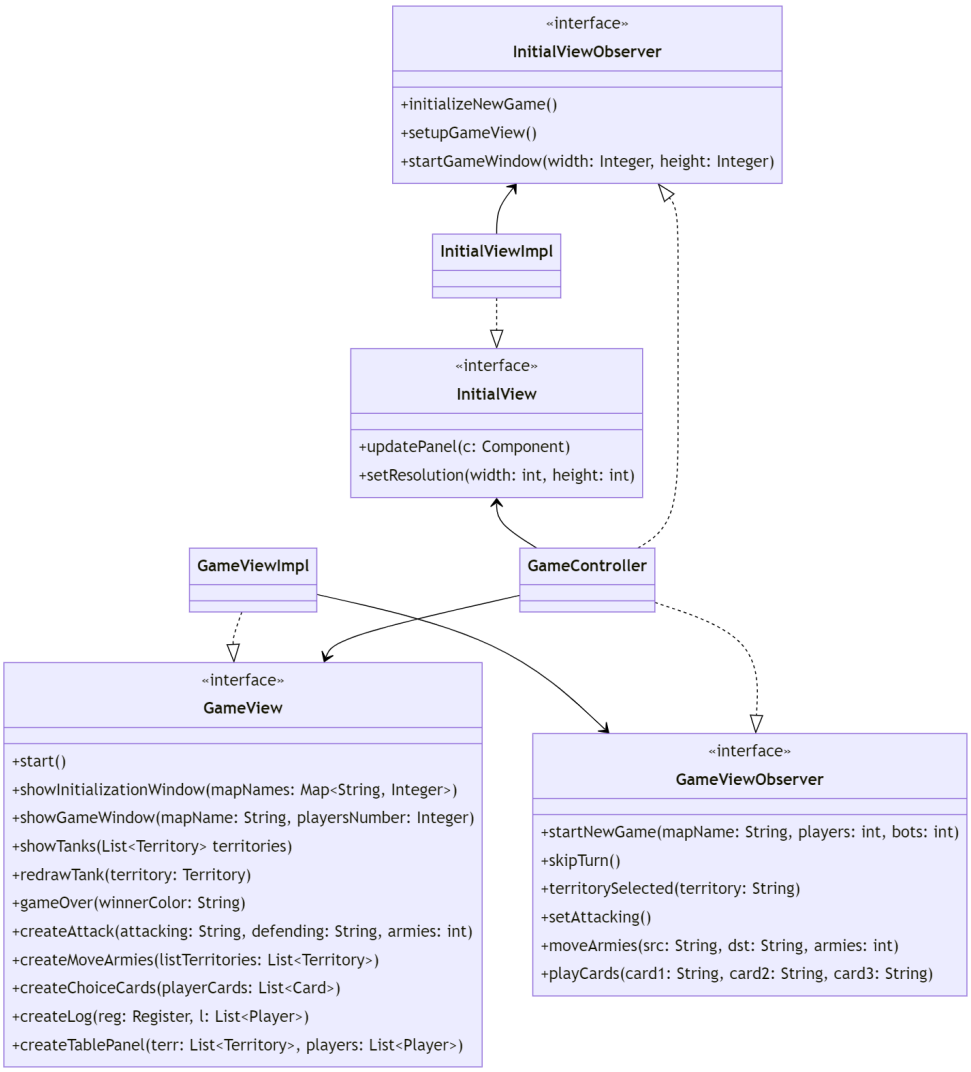

Figura 2.1: Schema UML che rappresenta il rapporto tra view e controller

Il **controller** utilizza le interfacce principali del **model**, senza che questo abbia un riferimento al **controller** (Ovviamente!). Nello specifico:

- **Territories** si occupa della gestione e della modifica dei territori e delle armate posizionate su di essi.
- **Deck** gestisce tutta la componente relativa alla gestione e all'utilizzo delle carte.
- **PlayerFactory** crea e inizializza i giocatori, sia reali che AI.
- **Register** tiene traccia degli eventi che si verificano durante la partita.

- **GameLoopManager** si occupa della gestione dei turni e delle varie fasi di gioco.

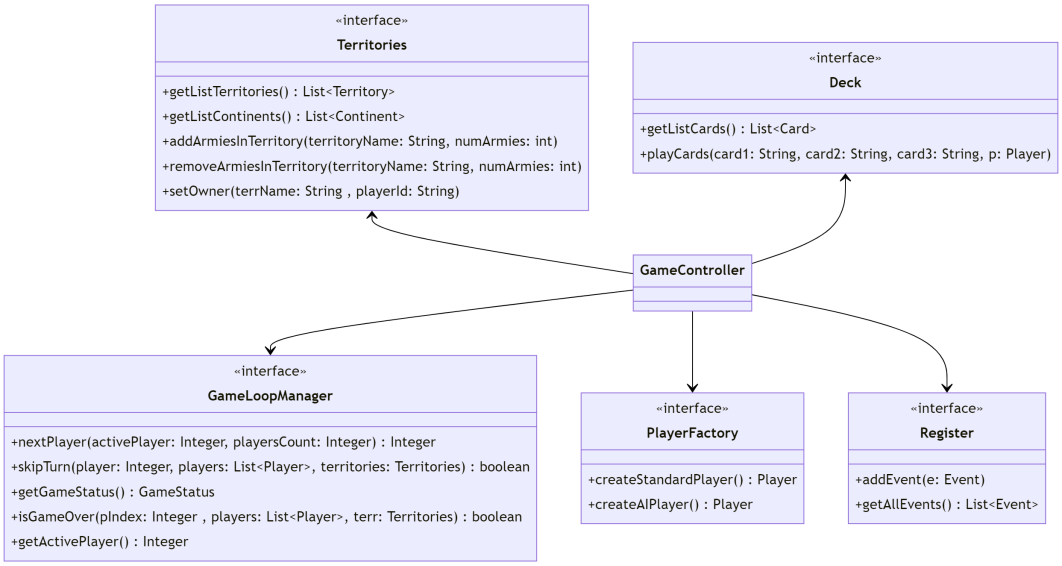

Figura 2.2: Schema UML che rappresenta il rapporto tra model e controller

2.2 Design dettagliato

2.2.1 Farneti Michele

Nel progetto ho realizzato, per quanto riguarda la parte di model, le dinamiche di inizializzazione della partita ed il meccanismo che coordina l'alternanza dei turni e delle fasi di gioco. Ho poi implementato la parte di controller atta a bloccare o ad attivare le azioni dell'utente in base allo stato del gioco e quella necessaria alla gestione dei click sui territori. A livello di view ho realizzato il layout del pannello di gioco principale comprensivo di mappa interattiva e barra delle azioni dell'utente.

Gestione del loop di gioco

Problema: Necessità di calcolare i risultati delle azioni del giocatore riducendo il numero e la complessità delle chiamate al gestore del loop da parte

del controller. Gestore del loop che a sua volta dovrà evitare ripetizioni di codice nella gestione di casi con effetti simili, come il posizionamento di armate effettuato in fasi diverse.

Soluzione: Risolvo il problema andando a realizzare dei gestori di turni, tali classi memorizzeranno stato di gioco, rappresentato da un enumeratore. ed indice del giocatore attivo riferito alla lista dei giocatori. Per mezzo dell'interfaccia **ActionHandler**, tramite metodi di tipo get, tali valori saranno restituiti una volta calcolati come risultanti dall'esecuzione di un azione di gioco. Tale interfaccia viene poi implementata da una classe astratta **ActionHandlerImpl** rappresentate un actionHandler generico. ActionHandlerImp sarà anche la base per realizzare il gestore del loop con cui si interfaccerà il controller attraverso l'interfaccia **GameLoopManager**, la quale metterà a disposizione metodi rappresentanti le principali azioni di gioco (Figura 2.3). Per gestire il posizionamento delle armate vado, nel gestore dei turni, a delegare il compito ad altri actionHandler specializzati per le varie fasi di gioco, come estensioni della classe astratta **PlaceArmiesActionHandler**. Per realizzarli, ricorro al pattern ***template method*** al fine di accorpare le azioni comuni, quali controlli sulla legittimitá dell'azione, decremento delle armate posizionabili del player e aggiornamento dello stato del territorio. La specializzazioni andranno ad implementare il metodo astratto ***updateGameStatus*** di **PlaceArmiesActionHandler**, calcolando i risultati delle azioni su stato di gioco e indice del giocatore attivo. La soluzione proposta fornisce la possibilità,in caso di future aggiunte alle funzionalità di gioco, di avere già a disposizione uno scheletro da estendere per implementarne il funzionamento. L'implementazione di gestori altamente specializzati rende inoltre più agevole intervenire modificando gli effetti di una singola azione di gioco senza andare ad intaccare le altre(Figura 2.4).

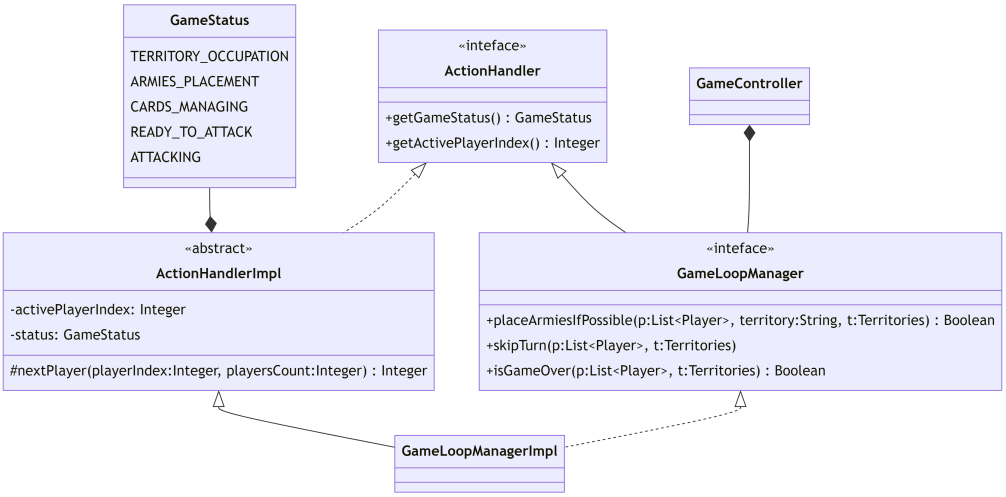

Figura 2.3: Schema UML del gestore del loop principale

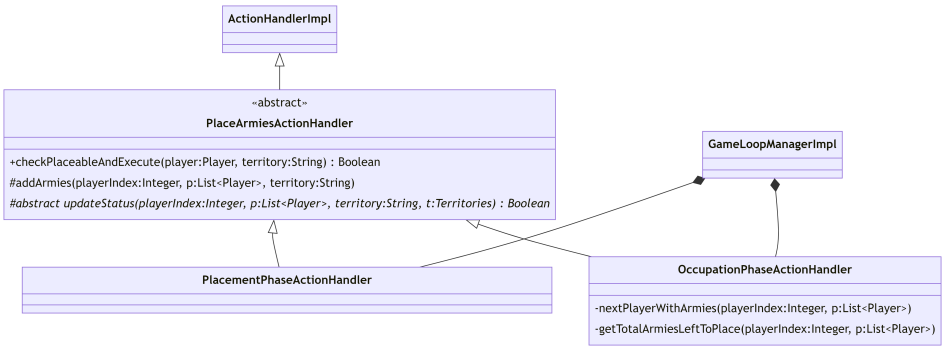

Figura 2.4: Schema UML degli action handler del piazzamento di armate

Gestione dell'inizializzazione dell'ambiente di gioco

Problema : Garantire scalabità del gioco riguardo l'implementazione di nuove mappe ed i loro relativi ambienti di gioco senza la necessità di intervenire sul codice.

Soluzione: Vado a definire una struttura organizzativa per i file che permetta, seguendo delle specifiche direttive di denominazione, di aggiungere nuove mappe selezionabili dal sistema di gioco mediante la sola aggiunta di una cartella per ciascuna, contente i file di testo per le specifiche dell'ambiente di gioco. Nel concreto, vado a realizzare dei metodi statici che associo alla classe **GameMapInitializerImpl** i quali permettano, esaminando i contenuti della cartella **maps**, di conoscere nomi e relativi numeri di giocatori massimo consentiti per ciascuna mappa. Tali dati saranno restituiti sotto forma di Map¡String,Integer¿, dove la stringa rappresenterà il nome della mappa. Sarà poi questa stessa classe che permetterà al controller, per mezzo dell'interfaccia **GameMapInitializer**, di ottenere i path per accedere alle risorsa della singola mappa una volta selezionata. La stessa istanza della classe fornirà anche un metodo per la generazione di target random da assegnare ai giocatori(Figura 2.5).

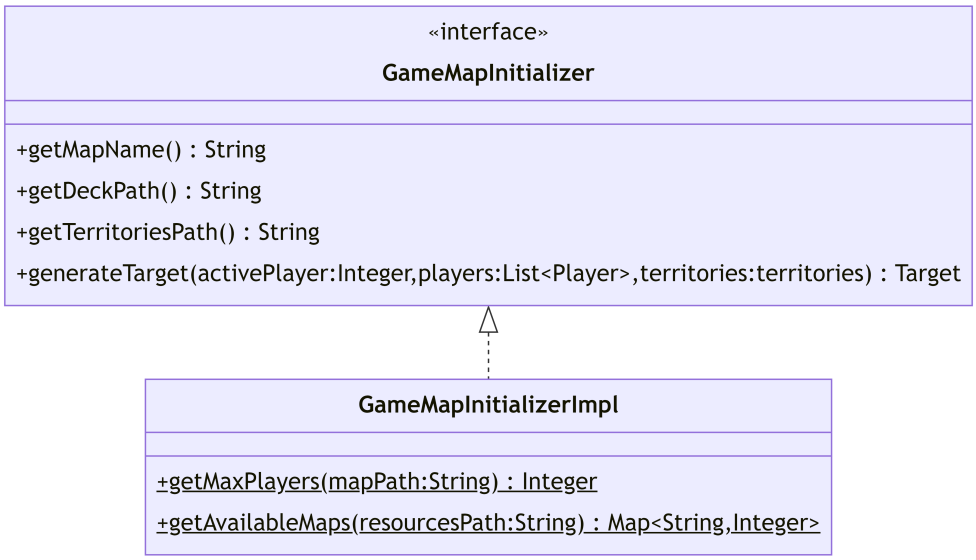

Figura 2.5: Schema UML del gameMapInitializer

Realizzazione della mappa di gioco interattiva

Problema : Necessità di realizzare entità sovrapposte alla mappa che permettano di interagire col territorio sottostante, oltre al farne visualizzare lo stato nel gioco, composto dal colore del giocatore possessore e dal numero di armate che lo occupano. Tali entità dovranno anche essere in grado di rimanere sovrapposte al territorio che rappresentano indipendentemente dalle dimensioni con cui la mappa viene renderizzata.

Soluzione: Vado a memorizzare nell'implementazione della **gameView** una collezione di entità associate ai territori, realizzati mediante un interfaccia **TerritoryPlaceHolder**, che poi andrò ad implementare in **TankIcon**. Ciascuna istanza di tale classe sarà la rappresentazione di un singolo territorio e disporrà di metodi per mezzo dei quali sarà possibile impostare l'aspetto grafico del bottone in base allo stato del territorio nel gioco. A tale scopo tankIcon mantiene come campi, oltre a riferimenti a colore del proprietario del territorio e nome del territorio, la JLabel utilizzata per mostrare il conteggio delle armate e l'effettivo bottone con cui l'utente potrà interagire. Quest'ultimo viene realizzato mediante la classe **ColoredImageButton** che va ad estendere il JButton di java swing, aggiungendo le funzionalità che permettono ad esso di apparire con l'aspetto di un carro armato di un determinato colore. Nello specifico tale classe manterrà un riferimento al colore del player che, al momento della stampa del componente sulla finestra andrà a selezionare l'immagine del colore corretto.
La generazione di **TerritoryPlaceHolder** è gestita sfruttando il pattern ***Simple Factory***. Nello specifico, una **TerritoryPlaceHolderFactory**, verrà inizializzata indicando il file da cui ottenere le coordinate per i vari territori e genererà nuove istanze di TerritoryPlaceHolder soltanto qualora le coordinate del territorio siano rintracciabili sul file. Il corretto posizionamento dei placeHolder sulla mappa indipendentemente dalla sua dimensione, viene eseguito sfruttando il pattern ***Strategy***. Nello specifico sarà **GameViewImpl** a specificare in base alle dimensioni della mappa una funzione per il ricalcolo delle coordinate dei placeHolder da passare al costruttore della factory (Figura 2.6).

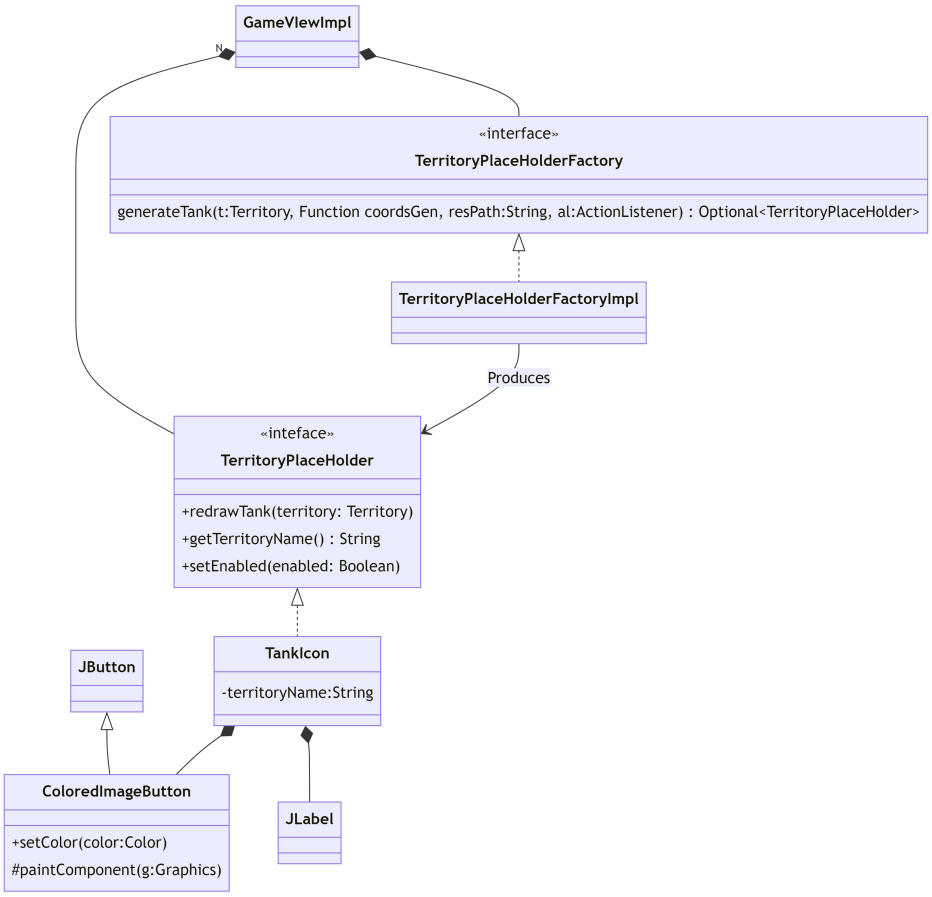

Figura 2.6: Schema UML dell'implementazione di TerritoryPlaceHolder

Ottimizzazione degli update grafici

I **ColoredImageButton** utilizzati per rappresentare i territori devono, ad ogni aggiornamento della grafica, andare a caricare da file un immagine di colore volta per volta diverso in base a chi é il possessore del territorio. Viene dunque realizzata l'interfaccia **ColoredImageReader** a cui i bottoni del progetto andranno volta per volta a riferisi specificando sia quale sia l'immagine richiesta che quale sia il suo colore, senza la necessitá di specificarne l'intero percorso. L'implementazione di tale interfaccia, ossia **ColoredImageReaderImpl**, andrá poi ad affidarsi ad un lettore di immagini standard, rappresentato dall'interfaccia **StandardImageReader**.

Problema: Il controller segnala all'interfaccia grafica con un unico metodo ***redrawView*** la necessità aggiornare la mappa di gioco. Risulta tuttavia

inutilmente dispendioso a livello di risorse andare ogni volta a recuperare dal file system le immagini rappresentative dei territori (i carri armati), in quanto solo un numero limitato di esse andrá ad essere utilizzato durante una singola partita.

Soluzione : Realizzo un sistema di caching per il lettore di immagini. Nello specifico, seguendo il pattern ***Proxy*** vado a wrappare un **SimpleImageReader**, implementazione di **StandardImageReader** in un **ImageReaderWithCache**, garantendo un'interfaccia trasparente per la lettura delle immagini ma riducendo di molto lo spreco di risorse (Figura 2.7).

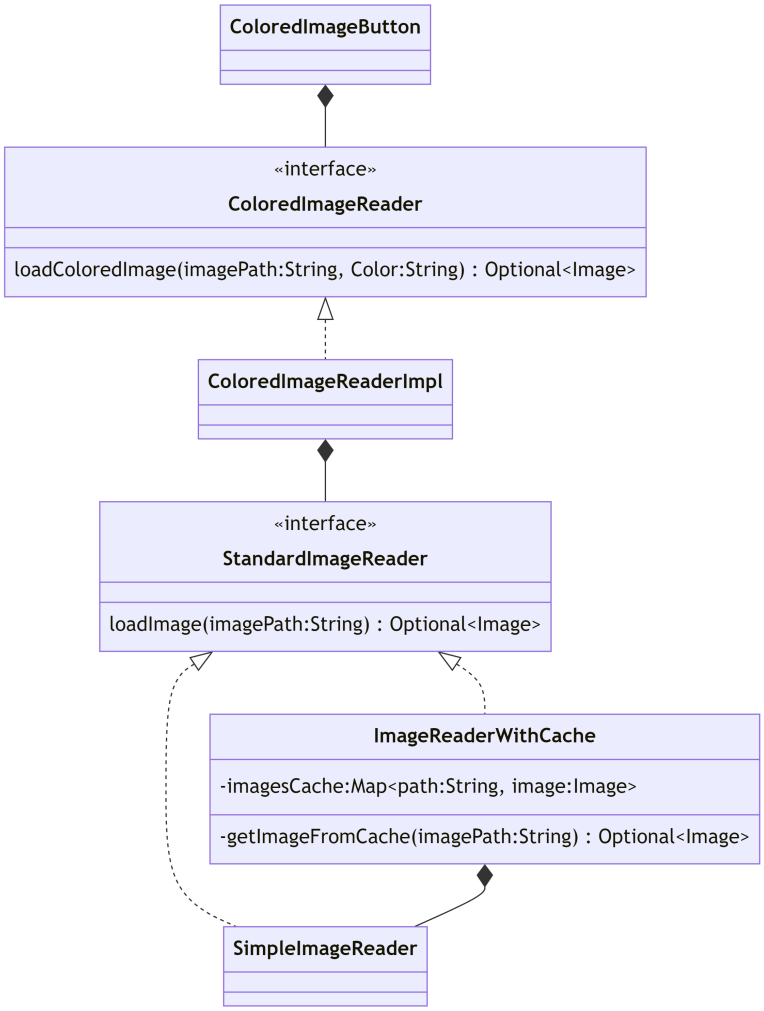

Figura 2.7: Schema UML delle classi per la gestione della lettura di immagini

2.2.2 Manuele D'Ambrosio

Nel progetto ho gestito la creazione dei giocatori (compresa l'implementazione dei giocatori AI) e tutta la parte relativa alla fase di attacco (model e view), ho inoltre creato la schermata di nuova partita (non citata nella proposta di progetto) e il sistema di salvataggio del gioco. Dopo aver implementato le componenti grafiche che ho utilizzato nei panel, le ho rese disponibili a tutto il progetto affinché potessero essere utilizzate anche dagli altri membri del gruppo.

Creazione dei Giocatori

Problema: L'insieme dei giocatori all'interno di una partita, può essere composto da una qualsiasi combinazione di giocatori reali e giocatori AI, con i secondi che devono poter effettuare le stesse azioni che possono effettuare i primi. Tutti i giocatori devono essere distinguibili gli uni dagli altri, inoltre deve essere possibile determinare se un giocatore sia reale oppure un giocatore AI.

Soluzione: Ho creato una classe **PlayerFactoryImpl** (Figura 2.8) che ha il compito di creare le istanze dei giocatori quando una nuova partita viene inizializzata, la factory semplifica la creazione dei due tipi di giocatori e fa anche in modo di assegnare a ciascuno di essi un differente colore impedendo che possano esistere due o più giocatori con lo stesso colore, in questo modo il colore svolge anche la funzione di identificatore. I giocatori reali ed AI sono stati implementati nella stessa classe **StdPlayer** e possono essere distinti con l'ausilio del metodo **isAI()**. Il compito di fare in modo che i giocatori AI possano prendere decisioni è stato assegnato alla classe **AIBehaviourImpl**, la quale utilizzando le informazioni del giocatore consente di formulare semplici scelte, le stesse scelte che vengono prese da parte del giocatore reale attraverso la **GUI**.

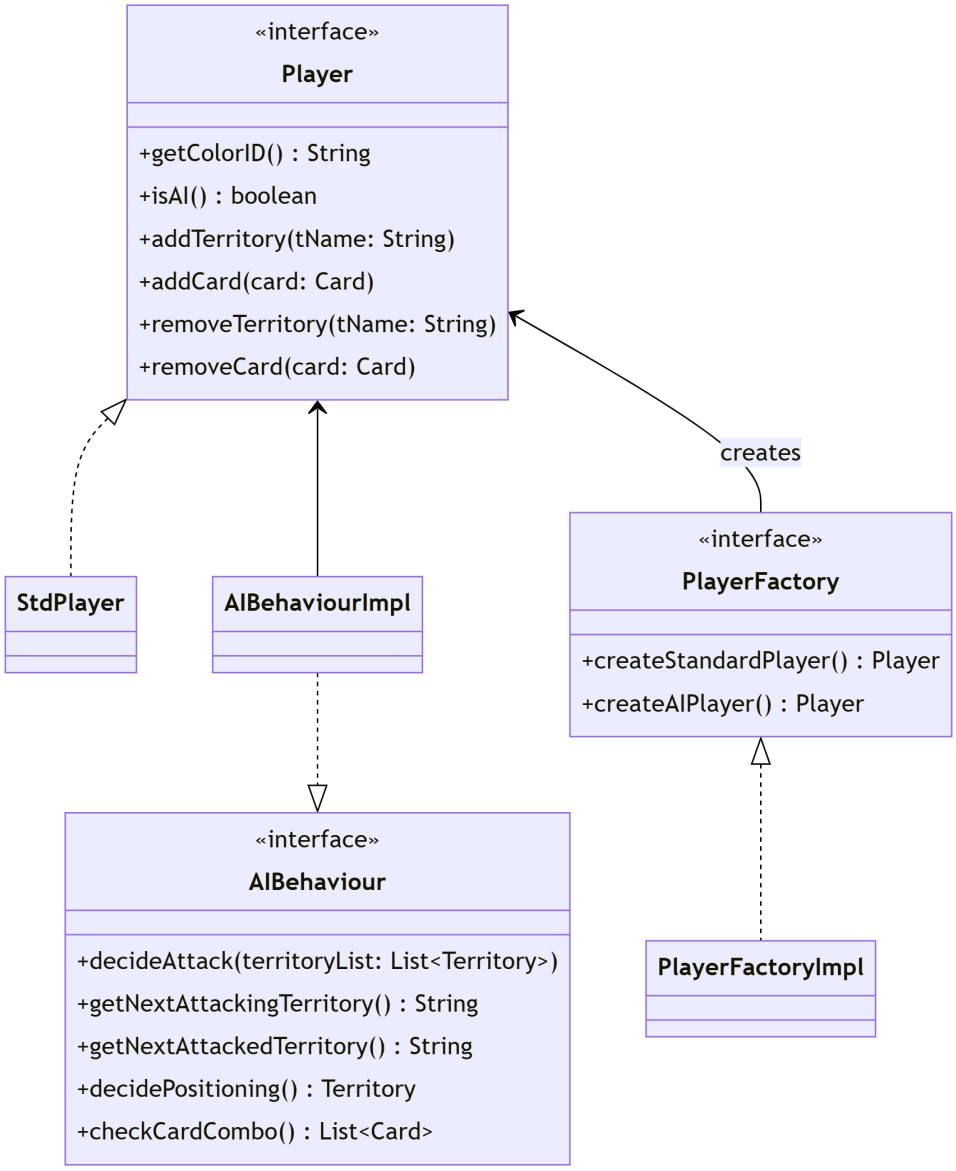

Figura 2.8: Diagramma UML della PlayerFactory e dei Player.

Gestione della fase di attacco

Problema: La fase di attacco è una componente centrale del gioco e si basa su un insieme di scelte strategiche che coinvolgono due giocatori (Quello attaccante e quello attaccato), e la parte più complessa da realizzare di questa fase è l'utilizzo dei dadi. Nello specifico l'attaccante può lanciare fino a tre dadi così come il difensore, i risultati vengono poi ordinati e confrontati e

determinano il numero di armate distrutte nei rispettivi territori dei due giocatori. Nel caso in cui l'attaccante abbia distrutto tutte le armate del difensore è presente anche una fase di conquista.

Soluzione: L'intera fase di attacco è gestita dalla classe **AttackPhaseImpl** (Figura 2.9) la quale si occupa di effettuare i lanci dei dadi, mediante l'utilizzo della classe **TripleDiceImpl**, e calcolarne i relativi esiti. La classe permette inoltre di segnalare al controller quante armate sono da distruggere e se durante l'attacco il territorio difensore è stato conquistato.

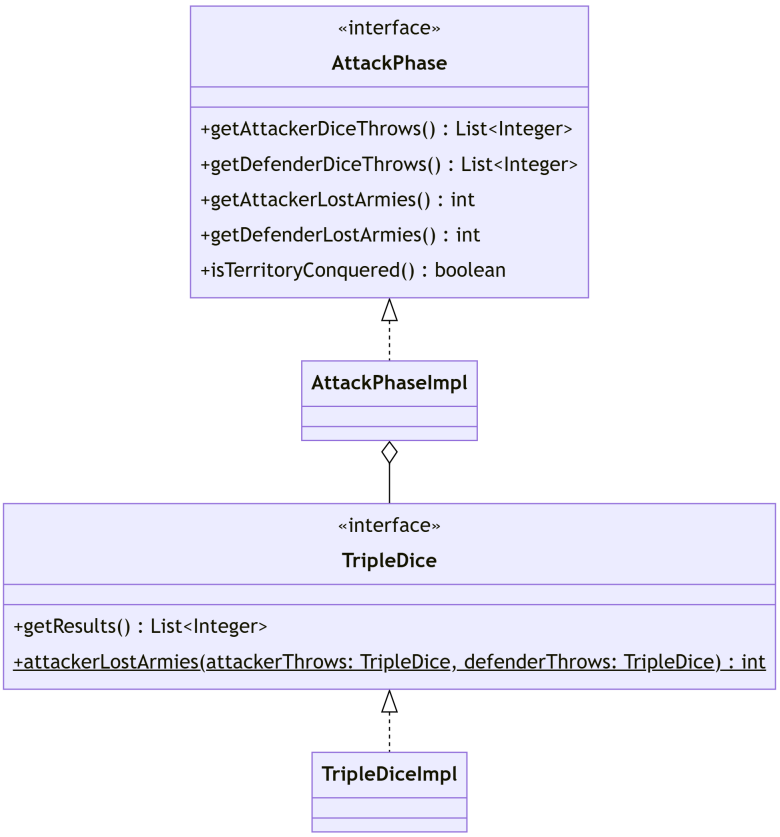

Figura 2.9: Diagramma UML del sistema di attacco.

Creazione del pannello di attacco

Problema: Durante la fase di attacco è necessario l'intervento del giocatore che, dopo aver selezionato il territorio da cui attaccare e il territorio da attaccare, deve scegliere il numero di armate con cui attaccare affinché

vengano calcolati gli opportuni esiti. Nel caso in cui il giocatore attaccante abbia conquistato il territorio attaccato allora dovrà scegliere il numero di armate da spostare in tale territorio.

Soluzione: Al momento della dichiarazione dell'attacco, ossia dopo aver selezionato il proprio territorio da cui vuole attaccare e il territorio avversario, si apre il pannello di attacco (Figura 2.10) che consente mediante l'uso di due pulsanti selettori di impostare il numero di armate attaccanti; durante la selezione viene controllato che il numero di armate attaccanti sia "legale" impedendo al giocatore di incrementare o decrementare oltre un certo limite in base alla situazione del territorio. Dopo aver selezionato le armate attaccanti queste verranno comunicate al **controller** utilizzando un **observer** (Implementato dal controller). Nel caso in cui il territorio sia stato conquistato, il **controller** segnala di aprire anche il panel di conquista (Il quale appartiene sempre all'**AttackPanel**) dove il giocatore dovrà selezionare (sempre mediante due pulsanti incremento/decremento) il numero di armate da spostare nel nuovo territorio. Dopodiché il panel viene chiuso dal **controller** e sarà possibile effettuare un nuovo attacco.

Creazione nuova partita

Problema: Dopo aver avviato l'applicazione l'utente potrà scegliere se creare una nuova partita oppure continuarne una precedentemente salvata. Nel caso in cui decida di iniziare una nuova partita il giocatore dovrà scegliere la mappa in cui giocare, il numero di giocatori reali e il numero giocatori AI; il totale dei giocatori dovrà essere sempre pari o inferiore al numero massimo di giocatori concessi dalla mappa selezionata (Il numero minimo di giocatori è stato arbitrariamente fissato a due).

Soluzione: Alla pressione del tasto di nuova partita presente nella schermata iniziale si aprirà il panel denominato **NewGameInitViewImpl** (Figura 2.10) il quale darà la possibilità di selezionare la mappa e il numero di giocatori. Ad ogni mappa è associato un numero massimo di giocatori oltre il qual non sarà più possibile aggiungerne. Al momento della conferma delle varie scelte la partita inizierà soltanto se tali scelte rispettano i requisiti (Citati nel paragrafo "Problema"), nel qual caso il panel comunicherà al **controller** le variabili necessarie all'inizializzazione della nuova partita mediante un **observer**. Per come è stata implementata la creazione della partita e la conseguente creazione della mappa di gioco, nel caso si volessero aggiungere nuove mappe non sarà necessario effettuare particolari modifiche al codice

ma sarà sufficiente inserire gli opportuni file (contenenti le informazioni della mappa) nella cartella nelle risorse dell'applicazione.

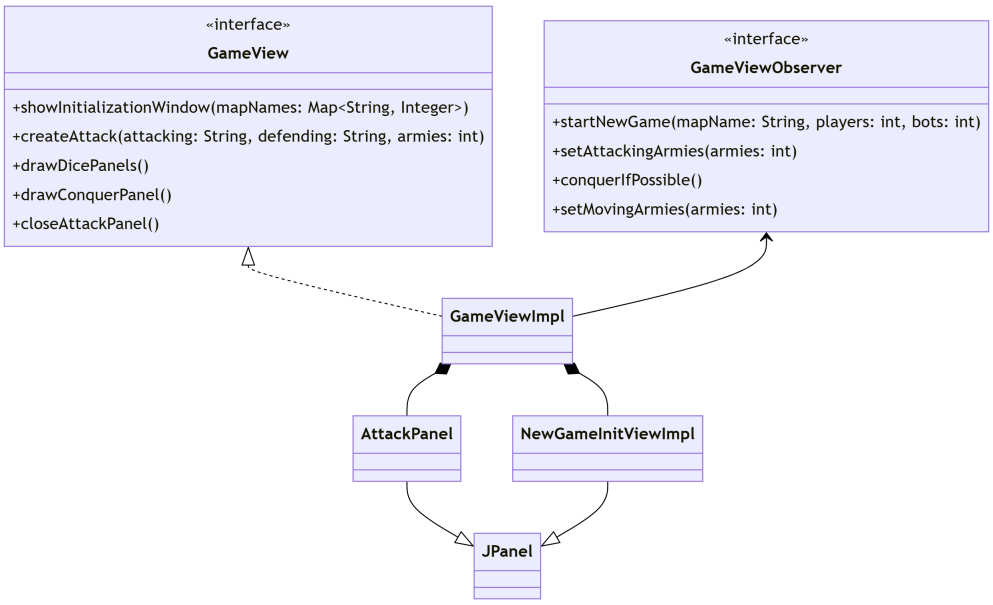

Figura 2.10: Diagramma UML rappresentante il panel di attacco e il panel di nuova partita.

Salvataggio

Problema: Durante la partita l'utente dovrà avere la possibilità di salvare lo stato del gioco, in questo modo all'apertura dell'applicazione sarà possibile continuare la partita precedentemente salvata.

Soluzione: Ho deciso di implementare il salvataggio della partita nella classe **GameSave** (Figura 2.11) la quale viene direttamente utilizzata dal **controller**. In particolare la classe si occupa di memorizzare tutte le informazioni fondamentali della partita salvandole su un file di testo, e successivamente queste informazioni saranno lette sempre dalla classe **GameSave** e passate al **controller**. Il salvataggio avviene mediante la pressione di un tasto nella

schermata principale di gioco, e sarà possibile premerlo soltanto se il giocatore corrente non sta svolgendo altre azioni (Ad esempio non è possibile salvare quando il panel della fase di attacco è aperto). Nel caso in cui l'utente prema il tasto per continuare una partita quando non è presente nessun file di salvataggio allora sarà reindirizzato alla schermata di creazione nuova partita.

NOTA: *Seppur teoricamente non corretto, per semplicità il file di salvataggio si trova nella cartella delle risorse dell'applicazione.*

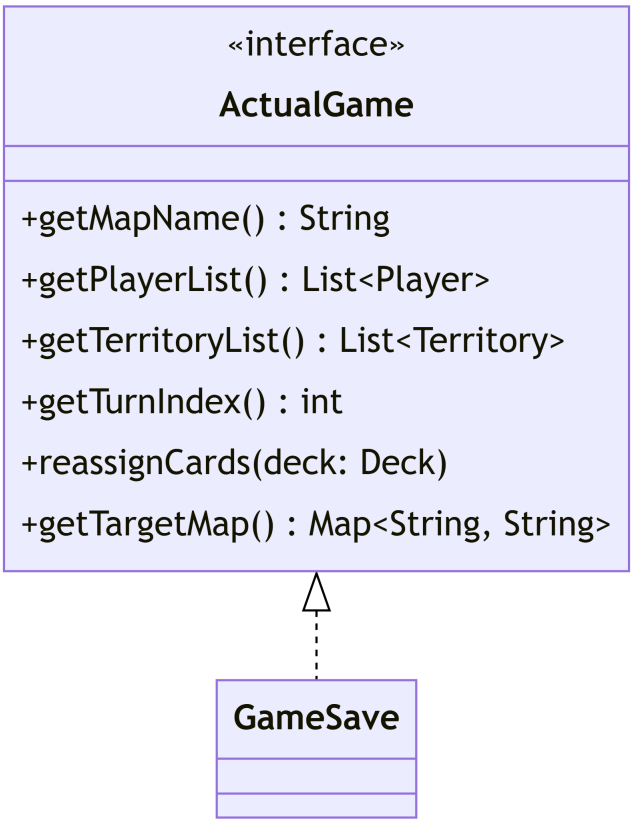

Figura 2.11: Diagramma UML della classe responsabile del salvataggio della partita.

2.2.3 Keliane Nana

Aggiornamento del loggerView durante la partita

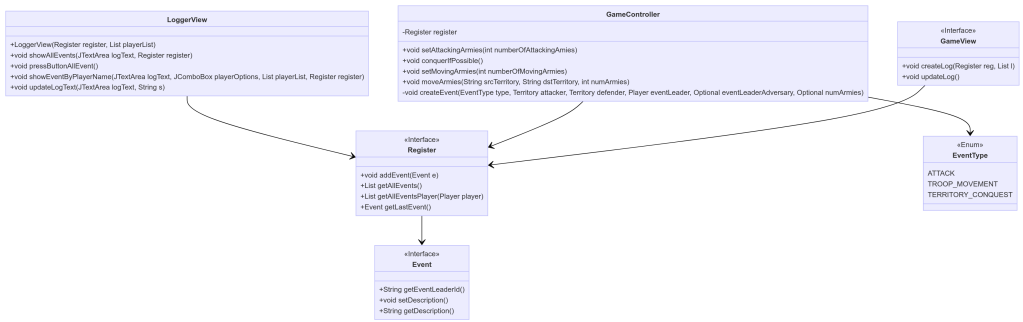

Figura 2.12: UML Aggiornamento LoggerView

Problema Realizzare un'implementazione del loggerView in modo che usufruisca di un aggiornamento a "tempo reale", per rassicurarsi che gli eventi siano tracciabili.

Soluzione Per risolvere questo problema è stato utilizzato il pattern Observer. Dato che il LoggerView dispone di un TextArea per contenere gli eventi della partita, una volta creato una sua istanza nel GameView grazie ad un register che li lega, gli aggiornamenti si fanno in modo sicuro con il metodo UpdateLog() chiamato dal controller ogni volta che si verifica un evento.

Riutilizzo del codice di baseTarget per creare diverse categorie di target

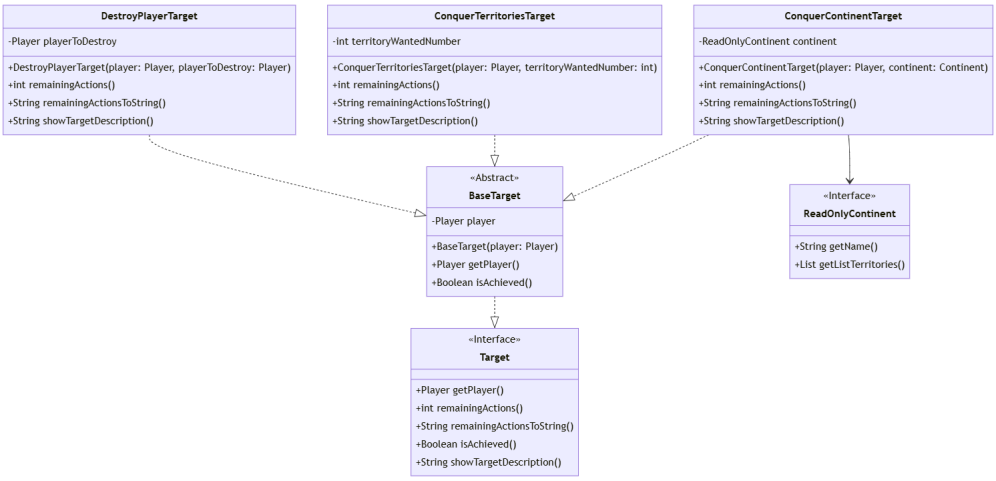

Figura 2.13: UML Gestione Target

Problema Progettando, dovevamo lavorare con più tipologie di Target avente alcune caratteristiche in comune.Era quindi importante scegliere il modo adeguato per modellarli.

Soluzione Per rendere la progettazione interessante, ho utilizzato Strategy pattern per consentire l'intercambiabilità dei target senza dover fare troppe modifiche e garantire una buona leggibilità del codice. Così fatto, ogni diversa tipologia si vede implementare solo le funzioni non avente un'implementazione comune con gli altri. Prima di scegliere quest'alternativa, ha valutato quella non usare BaseTarget come classe astratta ma avrebbe causato ridondanza inutile.

Gestione della view iniziale

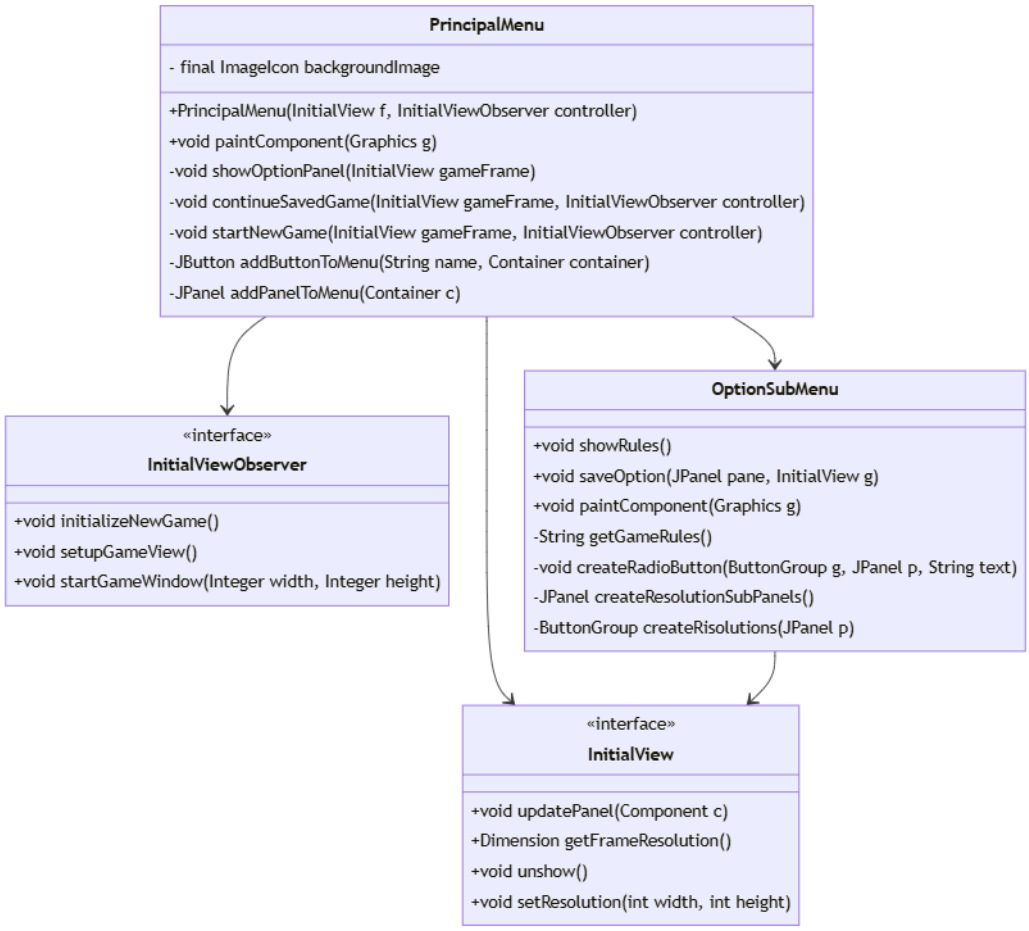

Figura 2.14: UML Gestione view iniziale

Problema La preoccupazione principale era di creare una schermata iniziale aggiornabile al secondo delle azioni svolte da un certo utente.

Soluzione La schermata iniziale InitialViewImpl viene inizializzata con un Panel (PrincipalMenu) dal quale si può decidere di iniziare una nuova partita, continuare quella salvata, visitare il menu delle opzioni o di uscire dal gioco.

2.2.4 Anna Malagoli

Gestione dei territori

Problema Modellazione dei territori della mappa del gioco.

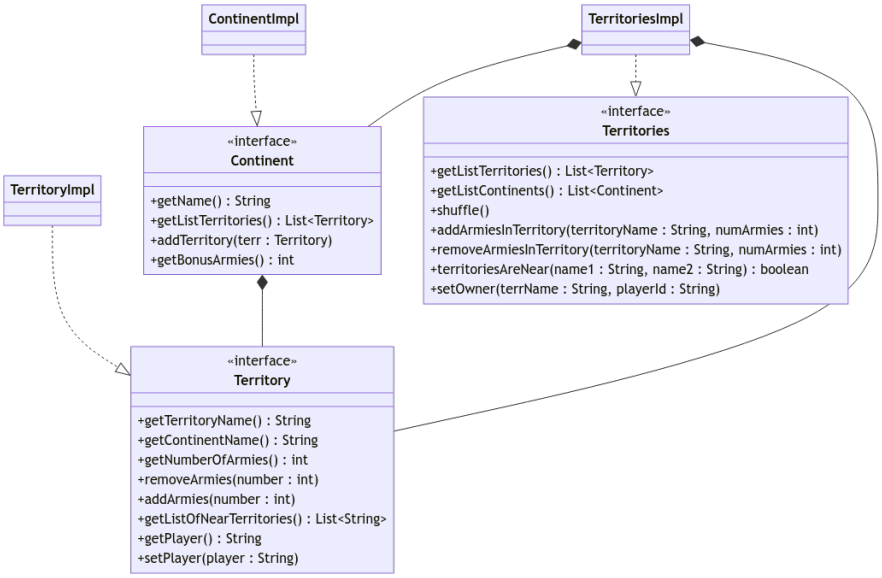

Figura 2.15: Schema UML dei territori e continenti

Soluzione I territori nel gioco del Risiko appartengono ad un continente, pertanto sono state divise queste due diverse componenti a livello implementativo separandole in due distinte classi. Sono state create pertanto due classi, una chiamata **Continent** e l'altra **Territory**, utilizzate per modellare continenti e territori che sono le interfacce delle implementazioni **ContinentImpl** e **TerritoryImpl**. Ogni territorio è caratterizzato da un nome univoco all'interno della mappa, dal nome del continente a cui il territorio appartiene, dal numero di armate presenti nel territorio, dalla lista di territori adiacenti e dall'id del giocatore che lo possiede. Non è presente nel territorio il riferimento al continente essendo il campo continente all'interno del territorio una stringa. A causa di quest'ultima scelta sarebbe necessario definire un metodo che dato il nome del continente ritorna il continente corrispondente per potervi fare modifiche. Però è stato scelto di mantenere solo il nome del continente in quanto a livello implementativo nel progetto è sufficiente conoscere il continente di un dato territorio senza dover successivamente agire su tale continente effettuandovi delle operazioni. Infatti, come si può vedere nel grafico UML sovrastante (Figura 2.15) è il continente a contenere una lista di territori. La classe **Continent** risulta essere funzionale

anche per memorizzare per ogni continente il numero di armate bonus che un giocatore riceve ad ogni turno se possiede tutti i territori del continente.

Per l'implementazione della lista di territori e di continenti del gioco si sceglie di non definire direttamente il loro inserimento nel codice, ma di inserire i territori con le loro specifiche proprietà (es. per i territori: il continente a cui territorio appartiene e la lista di territori adiacenti) in un file di testo. All'interno della classe **TerritoriesImpl** è presente un costruttore che prende come parametro di input una stringa che corrisponde al percorso del file da cui si vogliono estrarre le informazioni. All'interno di quest'ultima classe vengono inoltre definiti due metodi, *addArmies* e *removeArmies*, che consentono dal controller di modificare il numero di armate presenti in un dato territorio. Infatti, il controller non mantiene la lista di territori, ma solo un oggetto di **Territories** su cui viene chiamato il metodo *getListTerritories* per estrarre la copia della lista di territori. Per effettuare modifiche sullo stato degli oggetti territori è necessario chiamare i metodi messi a disposizione dalla classe **Territories**. In modo analogo ai metodi *addArmies* e *removeArmies* viene creato anche il metodo pubblico *setOwner* per cambiare in fase di gioco il possessore di un dato territorio. Una funzionalità che a livello di progetto si è deciso di inserire relativamente alla gestione dei territori è che all'inizio della partita, quando viene creata la lista di territori leggendola da file, venga utilizzato il metodo *shuffle* che permette di mischiare i territori all'interno della lista. In tale modo quando i territori vengono assegnati ai giocatori sono distribuiti in modo sempre diverso.

Gestione del mazzo di carte

Problema Creazione del mazzo utilizzato nel gioco e possibilità di giocare tre carte.

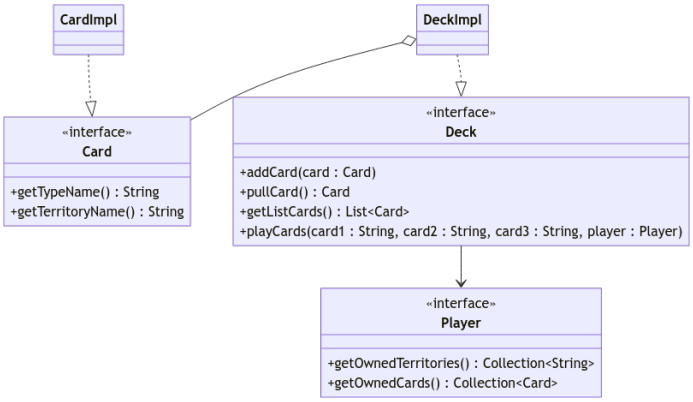

Figura 2.16: Schema UML del deck di carte

Soluzione Per la realizzazione del mazzo di carte è stata creata un'interfaccia **Card** che permette di modellare gli attributi legati ad una carta. Ogni carta è costituita da due proprietà, ovvero il nome di un territorio e il tipo di carta. La classe **CardImpl** risulta essere molto semplice in quanto viene creato l'oggetto carta dal costruttore che permette di settare i campi e sono presenti due metodi per estrarre tali valori. Il mazzo viene creato nella classe **DeckImpl** estraendo le informazioni per ogni carta dal file di testo passato al costruttore mediante il suo percorso. Alla fine della creazione del mazzo viene chiamato un metodo che permette di mischiare le sue carte. In questo modo la lista di carte ad ogni nuovo gioco risulta essere diversa rispetto all'ordine con cui le carte sono scritte nel file. All'interno della classe **Deck** viene definito il metodo *playCards* che permette ad un giocatore di giocare tre carte. In quest'ultimo metodo viene inserito, per ogni combinazione possibile di carte, il numero di armate che vanno aggiunte a quelle che il giocatore potrà giocare il turno successivo e viene calcolato il numero di armate extra che si guadagnano nel caso in cui le carte giocate siano relative a territori posseduti dal giocatore. Non viene verificato in questo metodo che le tre carte giocate costituiscano una combinazione di carte valida perché tale controllo è effettuato nella schermata di gioco realizzata per scegliere le tre carte da giocare.

Tabella dei territori

Problema Realizzazione di una tabella per visualizzare per ogni territorio il suo nome, il nome del continente, il numero di armate presenti e il colore con cui viene identificato il giocatore che possiede il territorio.

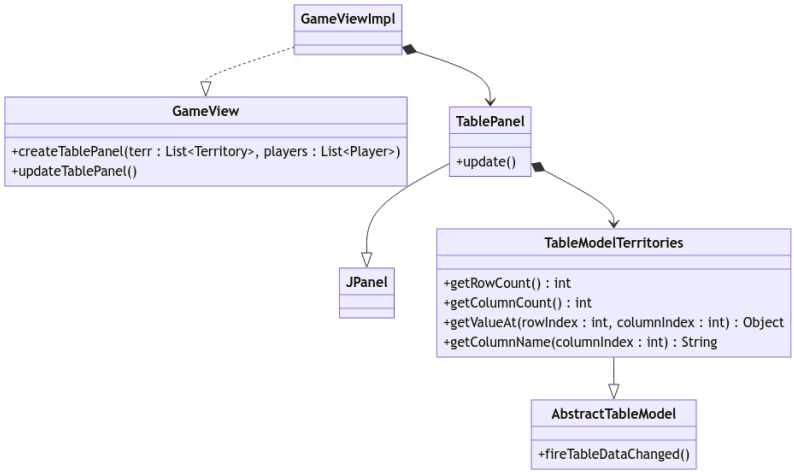

Figura 2.17: Schema UML della tabella dei territori

Soluzione La tabella viene creata nella classe **JTablePanel** che estende **JPanel**. Essa viene definita nel costruttore che prende come input un modello di tabella. Quest'ultimo è realizzato mediante un'apposita classe che estende la classe **AbstractTableModel**. Nel modello della tabella vengono gestiti i dati che si vogliono visualizzare all'interno di essa stessa. Dato che durante il gioco i valori che vengono visualizzati nella tabella mutano, si è deciso di inserire all'interno della classe **JTablePanel** il metodo *update* in cui viene chiamato il metodo *fireTableDataChanged* sull'oggetto del modello della tabella che permette di aggiornarla a fronte delle modifiche che sono state effettuate.

Grafica spostamento armate

Problema Creazione di una interfaccia grafica per effettuare l'operazione di spostamento di armate tra due territori adiacenti.

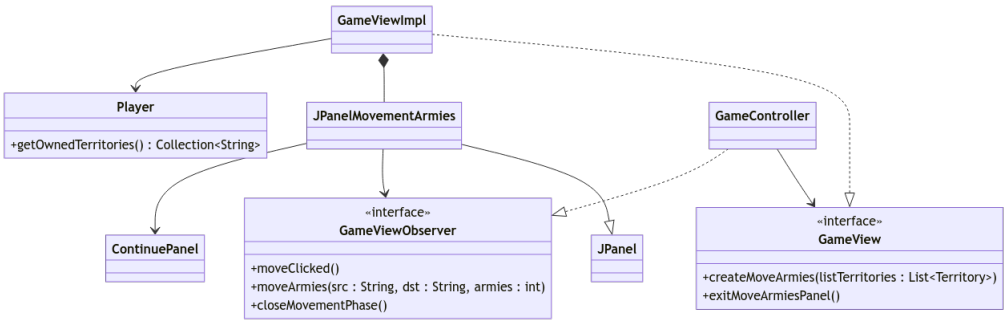

Figura 2.18: Schema UML del pannello per lo spostamento armate

Soluzione L'interfaccia grafica per lo spostamento delle armate viene creata attraverso un pannello contenente tre menù di selezione a tendina. I primi due sono utilizzati per l'inserimento dei territori e l'ultimo per la scelta del numero di armate da spostare. Nel pannello è inserito nella parte superiore un bottone che apre un box contente le informazioni utili all'utente per sapere come effettuare l'operazione di spostamento. Sono inoltre presenti due bottoni: uno per uscire dal pannello, ovvero per rimuoverlo dalla finestra principale di gioco, e uno per eseguire lo spostamento tra i territori selezionati. I bottoni sono implementati all'interno di pannelli creati attraverso l'uso della classe **ContinuePanel** in modo tale che siano conformi agli altri pannelli presenti nel gioco. Se l'utente inserisce territori che non sono adiacenti o se il numero di armate che vuole spostare risulta superiore o uguale a quello del territorio da cui vuole rimuoverle, allora l'operazione non viene eseguita, ma viene interrotta mostrando a video un pannello di errore. Se, invece, gli elementi selezionati dall'utente superano i controlli, allora viene chiamato un metodo presente nel **GameController** che aggiunge le armate al territorio destinazione e le rimuove da quello sorgente. Il pannello per lo spostamento delle armate tra due territori adiacenti viene creato tramite il metodo *createMoveArmies* presente nella classe **GameViewImpl**, che viene chiamato ogni volta che l'utente, durante il suo turno, clicca il bottone "Move" presente nella schermata di gioco.

Grafica per giocare una combo di carte

Problema Realizzazione di un'interfaccia grafica che consenta all'utente di giocare tre carte che danno origine ad una combo, ovvero ad una combinazione ammissibile delle carte secondo le regole del gioco Risiko.

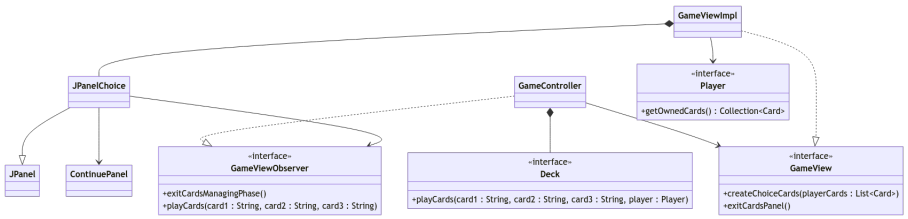

Figura 2.19: Schema UML del pannello per giocare le carte

Soluzione L'interfaccia grafica che viene visualizzata in fase di gioco, quando il giocatore possiede carte da giocare, viene creata similmente al pannello per lo spostamento delle armate trattato precedentemente (Figura 2.18). Infatti, contiene anch'esso un pulsante che permette di visualizzare le combinazioni di carte accettate dal gioco con i relativi bonus di armate, un bottone per uscire dal pannello e uno per giocare le tre carte selezionate. All'interno del pannello sono presenti tre celle di selezione contenenti le carte del giocatore. Una volta che queste ultime vengono selezionate, prima di essere giocate tramite la chiamata al metodo *playCards* del **GameController**, viene effettuato un controllo mediante l'uso di metodi privati della stessa classe del pannello che valutano che la combinazione di carte sia valida. In caso positivo vengono passate le stringhe con i nomi dei territori delle carte giocate al metodo *playCards* del controller che contiene l'oggetto **Deck** visto in precedenza (Figura 2.16). È tramite il deck, infatti, che viene effettuata l'operazione di gioco delle tre carte attraverso il metodo *playCards*.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il testing automatico delle classi realizzate é stato effettuato mediante l'uso di JUnit. Ciascun membro del gruppo si é reso responsabile di realizzare test sufficientemente esaustivi per le proprie classi.

3.1.1 Michele Farneti

GameLoop: È stata realizzata una classe di test che, emulando una serie di possibili azioni riconducibili ad un giocatore, é andata a verificare la correttezza degli aggiornamenti allo stato di gioco restiutiti dal gestore del loop principale e anche dai suoi singoli componenti indipendenti.
GameMap: Viene testata la corretta inizializzazione di una partita a partire dalle risorse messe a disposizione su file. È inoltre verificata la corretta implementazione della lettura di piú mappe.
TerritoryPlaceHolder: Tale classe di test verifica il corretto funzionamento delle classi di view atte alla generazione di segnaposti per i territori di una mappa.

3.1.2 Manuele D'Ambrosio

Per assicurarmi che funzionassero a dovere ho deciso di realizzare dei test per ognuna delle classi del **model** che ho realizzato. Nei test delle classi che ho scritto sono testati soltanto i metodi che hanno una implementazione non banale. Nello specifico per la classe **TripleDiceImpl** ho testato il corretto funzionamento del lancio dei dadi e del confronto dei risultati, successivamente ho testato anche la classe **AttackPhaseImpl** per verificare che gestisse

correttamente la fase di attacco e il relativo esito. Per la classe **Player** mi sono concentrato sul calcolo dei rinforzi e sulla gestione delle carte, mentre per la classe **AIBehaviour** ho verificato che l'AI compiesse decisioni plausibili. Infine ho testato la classe **GameSave** affinché i salvataggi caricati rispettassero esattamente la situazione della partita nel momento in cui è stata salvata.

Classi di test realizzate:

- TestTripleDiceImpl
- TestAttackPhase
- TestAIBehavuiour
- TestPlayer
- TestGameSave

3.1.3 Keliane Nana

- ***TestRegister****:* Test che ci permette di rassicurarsi del buon funzionamento del game register, che ci mantiene aggiornato sui diversi eventi che avvengono durante la pertita.
- ***TestTarget****:* Test che guarantisce che la gestione degli obbiettivi sia stata implementata in modo coretto, cioè che lo stato di un certo target sia corrispondente a quello del gioco.

3.1.4 Anna Malagoli

TestDeck: Testa che il Deck venga correttamente implementato tramite la lettura del file di testo e che il metodo playCards per giocare tre carte sia corretto.

TestTerritories: Testa la correttezza dell'implementazione di un oggetto Territory, dell'estrazione dei territori e dei continenti da file di testo e valuta il corretto funzionamento del metodo per verificare se due territori sono adiacenti.

3.2 Note di sviluppo

3.2.1 Michele Farneti

Feature avanzate del linguaggio e librerie utilizzate:

Utilizzo di `Stream`

: Piu volte utilizzate, un esempio é : link

Utilizzo di `Lambda Expressions`

: Piu volte utilizzate, un esempio é : link

Utilizzo di `Optional`

: Piu volte utilizzati, un esempio é : link

3.2.2 Manuele D'Ambrosio

Feature avanzate del linguaggio che ho utilizzato:

Stream:

Ho utilizzato gli stream in molti dei metodi che ho implementato.

Lambda Expressions:

Ho utilizzato le lambda per assegnare gli *Action Listeners* ad alcuni *JComponents* e nelle parti di codice in cui ho utilizzato anche gli stream.

Optional:

Ho utilizzato gli optional in tutti i casi in cui ci si postesse aspettare che una variabile assumesse valore null.

Esempi

- Esempio di un metodo che utilizza lambda e stream: link
- Esempio di una parte di codice che sfrutta gli optional: link

3.2.3 Keliane Nana

Utilizzo di Optional

Un esempio: link

Utilizzo di Stream e lambda expressions

Un esempio: link

Utilizzo di lambda expressions

Un esempio: link

3.2.4 Anna Malagoli

Optional: link
Lambda: link

Capitolo 4

Commenti finali

In quest'ultimo capitolo si tirano le somme del lavoro svolto e si delineano eventuali sviluppi futuri.

Nessuna delle informazioni incluse in questo capitolo verrà utilizzata per formulare la valutazione finale, a meno che non sia assente o manchino delle sezioni obbligatorie. Al fine di evitare pregiudizi involontari, l'intero capitolo verrà letto dai docenti solo dopo aver formulato la valutazione.

4.1 Autovalutazione e lavori futuri

Michele Farneti

Ritengo che il progetto mi abbia aiutato molto a comprendere al meglio le macchine del lavoro di gruppo in un progetto informatico, con i suoi lati negativi e positivi. La costante necessità di cooperazione con gli altri membri del gruppo ritengo mi abbia formato al punto da ritenermi in grado di riaffrontare esperienze simili senza la paura di gravare sulla tabella di marcia di altri membri di un team di sviluppo. Per quanto riguarda il lato meramente inerente alla programmazione, mi ritengo abbastanza soddisfatto dalle nuove conoscenze sul linguaggio che ho acquisito. Riga di codice dopo riga di codice ho compreso sempre di più l'importanza di una buona progettazione che deve stare alla base del progetto. In tal senso, purtroppo concludo il progetto quasi con l'amaro in bocca poichè ritengo che le conoscenze acquisite in ogni nuova fase del progetto mi avrebbero dato delle basi per realizzare una progettazione a monte più efficiente. Resto tuttavia in generale soddisfatto dal risultato finale ottenuto poichè all'apparenza risulta conforme con quelle che erano le mie idee di realizzazione iniziali. Pensando ad uno sviluppo futuro del progetto, non mi dispiacerebbe andare ad utilizzare come base di partenza per la realizzazione di un gioco con una user experience più accattivante

per il giocatore, tuttavia penso che alla fine la prima fase sarebbe comunque una riprogettazione completa dell'applicazione.

Manuele D'Ambrosio

Durante la realizzazione del progetto ho migliorato notevolmente le mie competenze nella programmazione a oggetti, e per questo ogni volta che torno a guardare parti di codice scritte in precedenza mi viene voglia di rifarle completamente da capo in quanto sono sicuro che ora saprei fare di meglio. Questa consapevolezza tuttavia mi impedisce di essere contento dei risultati ottenuti e mi lascia il desiderio di ricominciare tutto il progetto dal principio con le competenze attuali, in quanto saprei esattamente cosa migliorare, soprattutto dal punto di vista del design che ritengo non ottimo a causa del fatto che fosse la prima volta che ho utilizzato il modello MVC in un progetto di queste dimensioni. Quello di cui sono più soddisfatto è l'aver utilizzato spesso stream e lambda in quanto le ritengo feature estremamente utili una volta apprese, sono anche contento del risultato visivo delle mie parti di view e delle componenti che ho creato. In conclusione sono soddisfatto della riuscita del progetto come effettivo gioco ma non sono altrettanto soddisfatto di come ho scritto alcune parti di codice, in particolare le prime che ho scritto.

Keliane Nana

positivo Partecipare nel realizzare questa applicazione mi ha permesso di acquisire nuove competenze e sono soddisfata del risultato finale. Sono felice di aver svillupato con altre persone e di aver raggiunto l'obbietivo principale che avevamo fissato, contenta di aver concretizzato quanto imparato a lezione.

negativo Il fatto di aver ciascuno il proprio timeTable ci ha messo in difficoltà e quindi ralentato nel team coding; il che ci ha laciato poco tempo alla fine per poter risolvere gli errori generati dal programma. Penso che avremmo potuto organizzarsi meglio per non stressare a fine realizzazione. Magari se fosse stato il caso sarebbe stato possibile aggiungere alcune funzionalità al gioco come i sounds per esempio.

conclusione Di solito, preferisco lavorare da sola perché mi concentro molto sugli aspetti negativi del teamwork. Pero, grazie a quest'esperienza posso dire che adesso vedo le cose un po' diversamente. Non sto dicendo che sia sempre stato facile, ma in somma c'è stato più di positivo che di negativo ed ho imparato cose che magari lavorando da sola, non avrei mai imparato.

Anna Malagoli

Mi ritengo abbastanza soddisfatta del progetto che abbiamo realizzato. Questa è stata la prima volta che ho partecipato ad un progetto di gruppo così impegnativo. Sebbene inizialmente ci siano state delle difficoltà, questa esperienza è risultata essere formativa e stimolante. Infatti, mi ha permesso di ampliare le mie conoscenze di programmazione ad oggetti e le mie capacità collaborative in quanto ognuno dei componenti del gruppo è stato fondamentale per la realizzazione del progetto. Ho inoltre approfondito e migliorato l'utilizzo di alcuni strumenti come Git e di alcune classi per la lettura di file di testo. D'altra parte all'interno della mia parte di progetto non ho utilizzato pattern di programmazione in quanto non sono risultati essere necessari, cosa che però mi piacerebbe utilizzare in progetti futuri. In conclusione penso di aver partecipato attivamente alla realizzazione di questo progetto e mi sento fiera sia del risultato che delle competenze apprese lungo il percorso.

4.2 Difficoltà incontrate e commenti per i docenti

Michele Farneti

Innanzitutto ritengo che aver atteso svariati mesi dalla fine del corso per iniziare il progetto possa aver giocato a mio sfavore poichè mi sono ritrovato un pò disallenato dal seguire i paradigmi della programmazione Java, il che si è manifestato sotto forma di inconsapevole utilizzo di cattive pratiche nella scrittura di codice che poi ho dovuto ad andare a correggere, a volte costringendomi anche a rivedere completamente la progettazione. Non è stato semplicissimo lavorare in gruppo, la mia parte in particolare era estremamente dipendente da quelle di altri in quanto andava a mettere in relazione svariati oggetti del model di cui altri membri del gruppo erano responsabili.

Manuele D'Ambrosio

La principale difficoltà che ho incontrato durante la realizzazione del progetto è dovuta al fatto che fosse il mio primo progetto di questo tipo e che quindi inizialmente non avevo ben chiaro come dovessi procedere soprattutto a livello di design e interazione tra le varie classi principali. Avrei pertanto gradito avere a disposizione una guida di qualche tipo per capire in quale direzione procedere, in particolare per quanto riguarda l'implementazione dell'MVC quando si ha a che fare con molte classi diverse sia di model che di view. Un'altra grande difficoltà è stata il lavoro di gruppo: per quanto

ogni membro abbia realizzato in modo completo le sue parti posso dire che non tutti abbiano dedicato lo stesso impegno nella realizzazione delle parti comuni e nella correzione dei grossi problemi incontrati durante le ultime fasi di realizzazione dell'applicazione. Nel complesso ho comunque gradito come è stato realizzato il corso e di come mi abbia fatto acquisire competenze che difficilmente avrei ottenuto in altro modo.

4.2.1 Keliane Nana

Durante la realizzazione, ho incontrato qualche difficoltà nel stabilire la comunicazione MVC per la gestione del Logger e ho impiegato molto tempo nel risolvere alcuni bugs.

Appendice A

Guida utente

A.0.1 Avvio

Per avviare l'applicazione posizionarsi nella cartella *OOP23-RisiKo* ed eseguire il file '.jar' tramite il comando "*java -jar OOP23-Risiko-all.jar*", oppure eseguirlo manualmente.

A.0.2 Menu iniziale

La schermate inziale del gioco (Figura A.1) offre le seguenti opzioni:

- **New game:** Per inizare una nova partita, aprendo una nuova schermata dedicata alla personalizzazione.
- **Continue:** Per continuare la partita slavata in precedenza
- **Option:** Permette di accedere al setting delle impostazioni grafiche.

Figura A.1: Schermata inziale

Per selezionare la risoluzione, una volta aperto il menù opzioni, sarà sufficiente cliccare su **resolution**, spuntare la propria scelta e poi cliccare su **save** per applicare la modifica (Figura A.2).

Figura A.2: Selezione della risoluzione

A.0.3 Personalizzazione della partita

La schermata di personalizzazione (Figura A.3) permette, mediante la pressione del tasto **next**, di effettuare la selezione della mappa. Una volta selezionato anche il numero di giocatori, il tasto **start** permetterà di caricare la schermata di gioco.

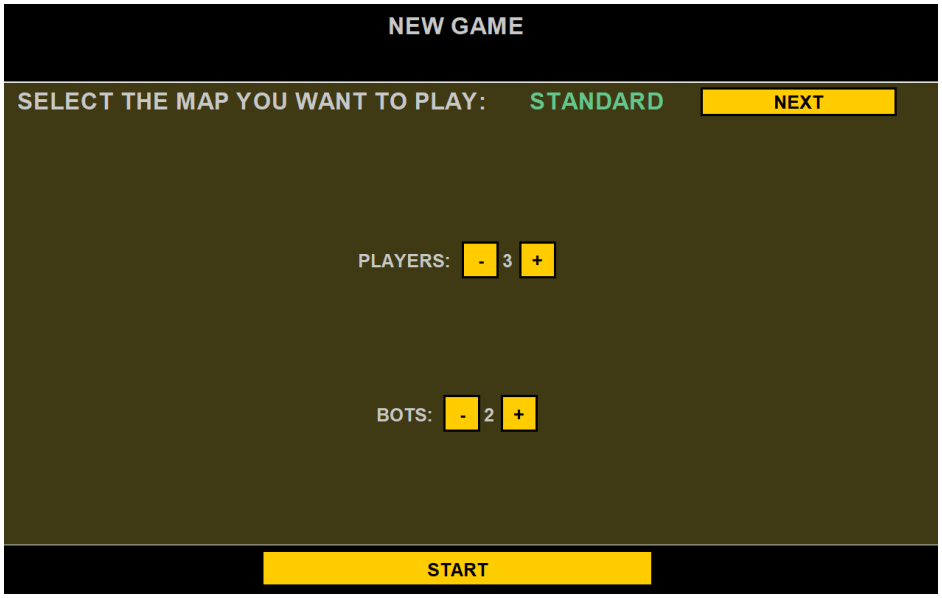

Figura A.3: Schermata di personalizzazione della partita

A.0.4 Schermata di gioco

Le schermata di gioco (Figura A.4) presenta i bottoni rappresentati le diverse azione del giocatore. In particolare per ogni fase di gioco i soli bottoni riferiti alle azioni consentite saranno attivati. Sulla sinistra sarà possibile tener traccia dell'alternanza dei turni grazie ad icone colorate rappresentanti i giocatori. L'icona varierà in base alla sua natura (AI/Standard). Sotto le icone dei giocatori sarà presente una casella dove visualizzare il nome dei territori selezionati. A destra della schermata sarà possibile visualizzare: Nella parte superiore il log delle azioni compiute dai giocatori (rappresenterà solo eventi di attacco, conquista di territori e spostamento di armate), nella parte inferiore la tabella delle nazioni. Al centro della barra d'attacco sarà possibile visualizzare il conteggio delle armate posizionabili dal giocatore corrente.

Figura A.4: Schermata di gioco

A.0.5 Azioni di gioco non banali

- **Posizionamento delle armate:** Eseguibile mediante click sui carri armati posti sopra i territori.
- Attacco: Il gioco entra in fase di attacco una cliccando il pulsante **attack**, dopodichè sarà sufficiente cliccare i due territori combattenti sulla mappa per far aprire la schermata per la personalizzazione dell'attacco (Figura A.5).
- Spostamenti strategici: Il tasto **Move** permette al giocatore corrente di visualizzare una schermata per la scelta di uno spostamento strategico da effettuare (Figura A.6) (ATTENZIONE:Una volta effettuato il turno viene skippato in automatico).
- Gestione delle carte: Quando un giocatore entra in possesso di carte dopo aver conquistato un territorio, avrà l'opportunità di selezionare la combo e giocarla attraverso una finestra (**??**) che si aprirà all'inizio del proprio turno.

Figura A.5: Attack panel

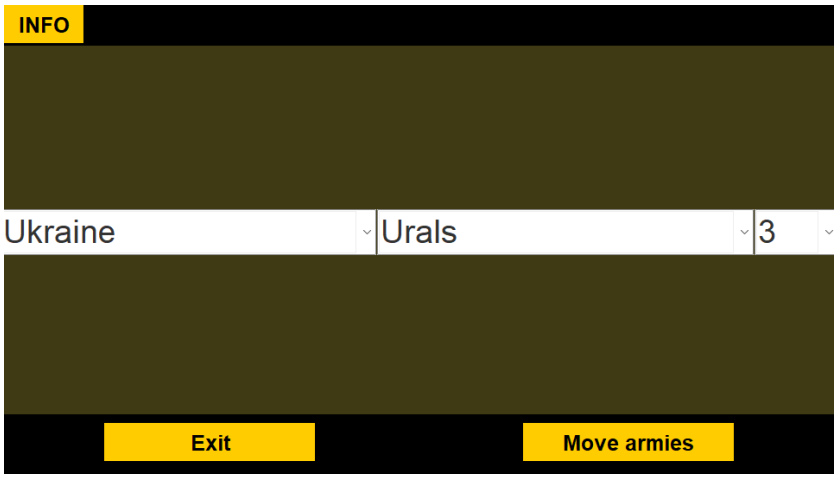

Figura A.6: Movements panel

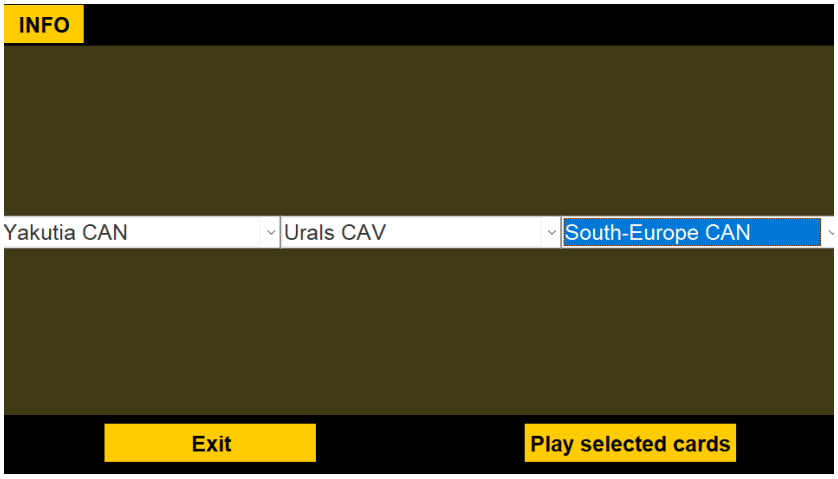

Figura A.7: Cards panel

A.0.6 Fine della partita

Quando un giocatore raggiungerà il proprio obiettivo, verrà visualizzata una schermata per segnalare la vittoria (Figura A.8).

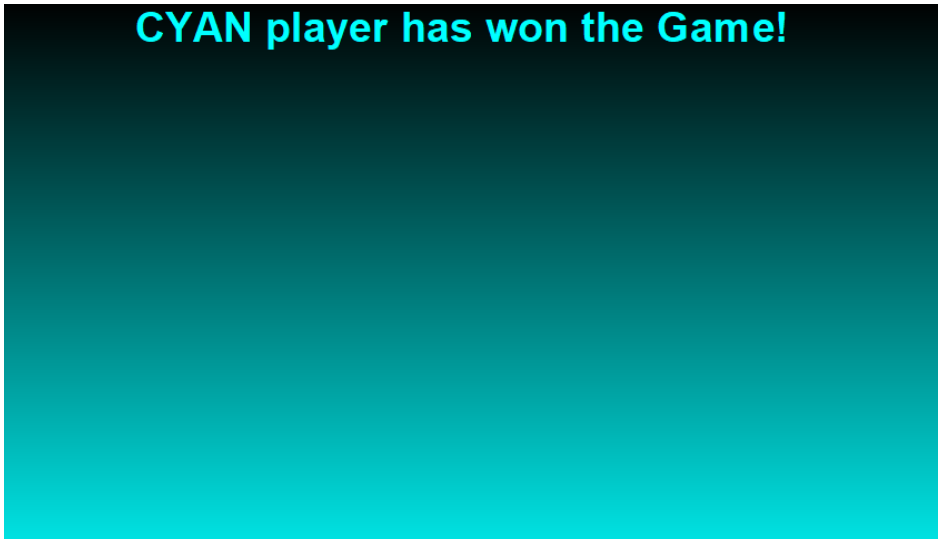

Figura A.8: Schermata di vittoria

Appendice B

Esercitazioni di laboratorio

B.0.1 michele.farneti@studio.unibo.it

B.0.2 manuele.dambrosio@studio.unibo.it

B.0.3 kelianenidele.nana@studio.unibo.it

- Laboratorio 08: `https://virtuale.unibo.it/mod/forum/discuss.php?d=148025#p210219`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=149231#p211485`
- Laboratorio 10: `https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212729`

B.0.4 anna.malagoli2@studio.unibo.it