KINGDOMCLASH

Report

Kane Abdoulaye
La Farciola Lorenzo
Francolini Marco
Sinani Silvi

Indice

Capitolo 1

Analisi

Il progetto consiste nella realizzazione di un gioco strategico ispirato in parte a un minigioco presente in Assassin's Creed Valhalla chiamato" Orlog".

1.1 Requisiti

Requisiti Funzionali

- Realizzazione di una campagna a vari livelli con difficoltà incrementale.
- Possibilità di costruire, potenziare la base e consentire al giocatore di gestire le proprie risorse per potenziare truppe o edifici.
- Combattimento strategico a turni tra giocatore e un bot, aggiungendo un fattore randomizzante.
- Possibilità di salvare la partita.
- Interfacce utente intuitive.

Requisiti non funzionali

- Integrazione di un menù.
- Musica di sottofondo.
- Pannelli informativi (info su come giocare).
- Possibilità di caricare salvataggi.
- Aggiunta di più battaglie.
- Aggiunta di più edifici.

1.2 Analisi e modello del dominio

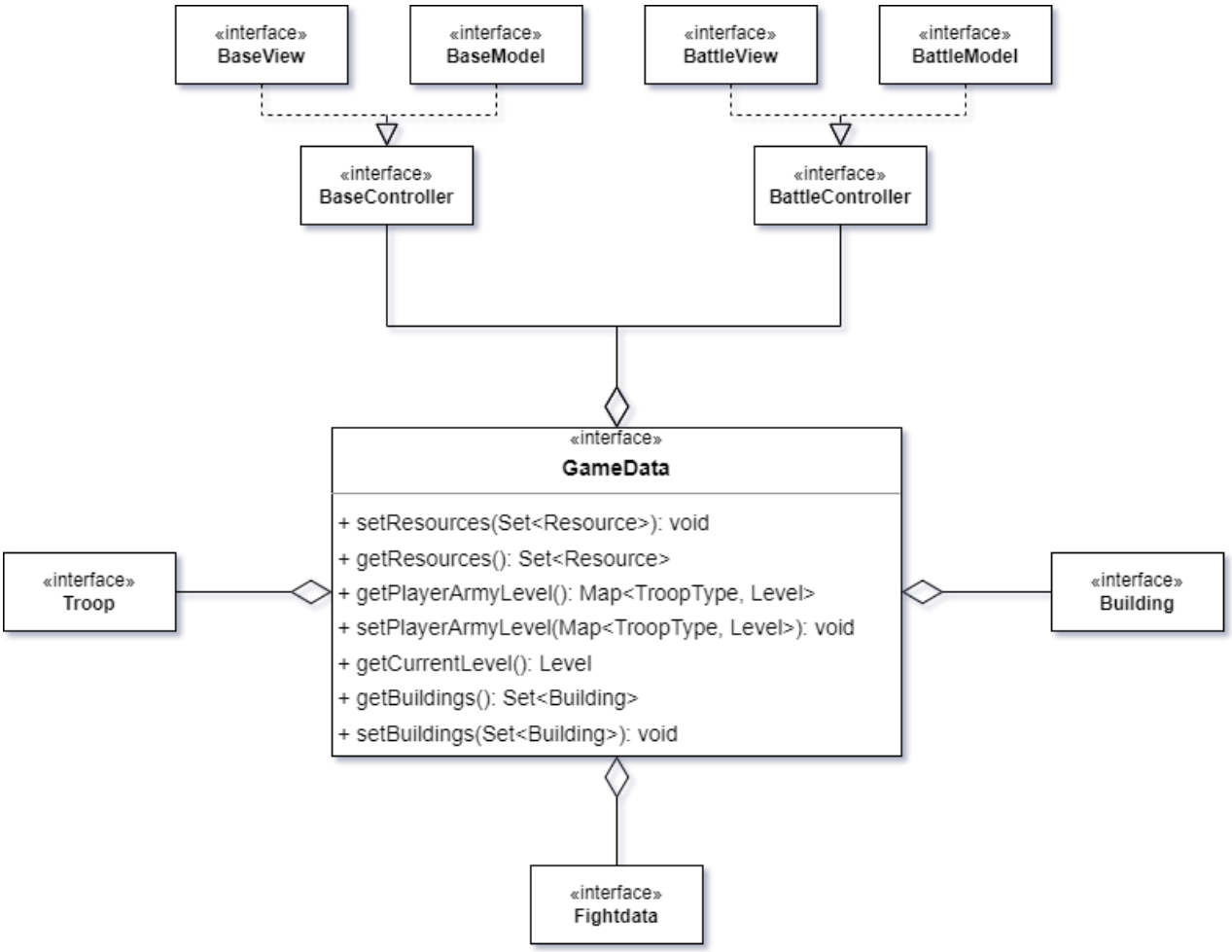

Figura 1.1: Schema UML dell'analisi del problema

Il gioco è costruito da due sezioni principali:

- Una parte dove si gestisce una base appartenente al giocatore, che permette di piazzare edifici e potenziare truppe
- Una parte di battaglia da vincere che permette di passare al livello successivo.

Ogni battaglia ha la sua difficoltà basata sui livelli delle truppe del bot. Il giocatore è libero di scegliere quali truppe potenziare nella sua base a seconda della strategia che ritiene più opportuna per vincere le battaglie.
Ogni edificio produce ciclicamente un quantitativo di materiali utile a potenziare le truppe o gli edifici stessi.
Ogni battaglia è composta da un numero indefinito di fasi. Le fasi di battaglia forniscono tre possibilità per cambiare le truppe non ancora schierate in modo randomico, permettendo al giocatore di scegliere al meglio quali truppe dispiegare nel campo di battaglia. Una volta che gli avversari hanno esaurito tutti i tentativi oppure schierato tutte le truppe si arriva alla fine della fase di battaglia e si ha un confronto tra le truppe schierate. Se uno dei due opponenti esaurisce completamente le vite, la battaglia giunge a un termine, altrimenti ricomincia una nuova fase.

Capitolo 2

Design

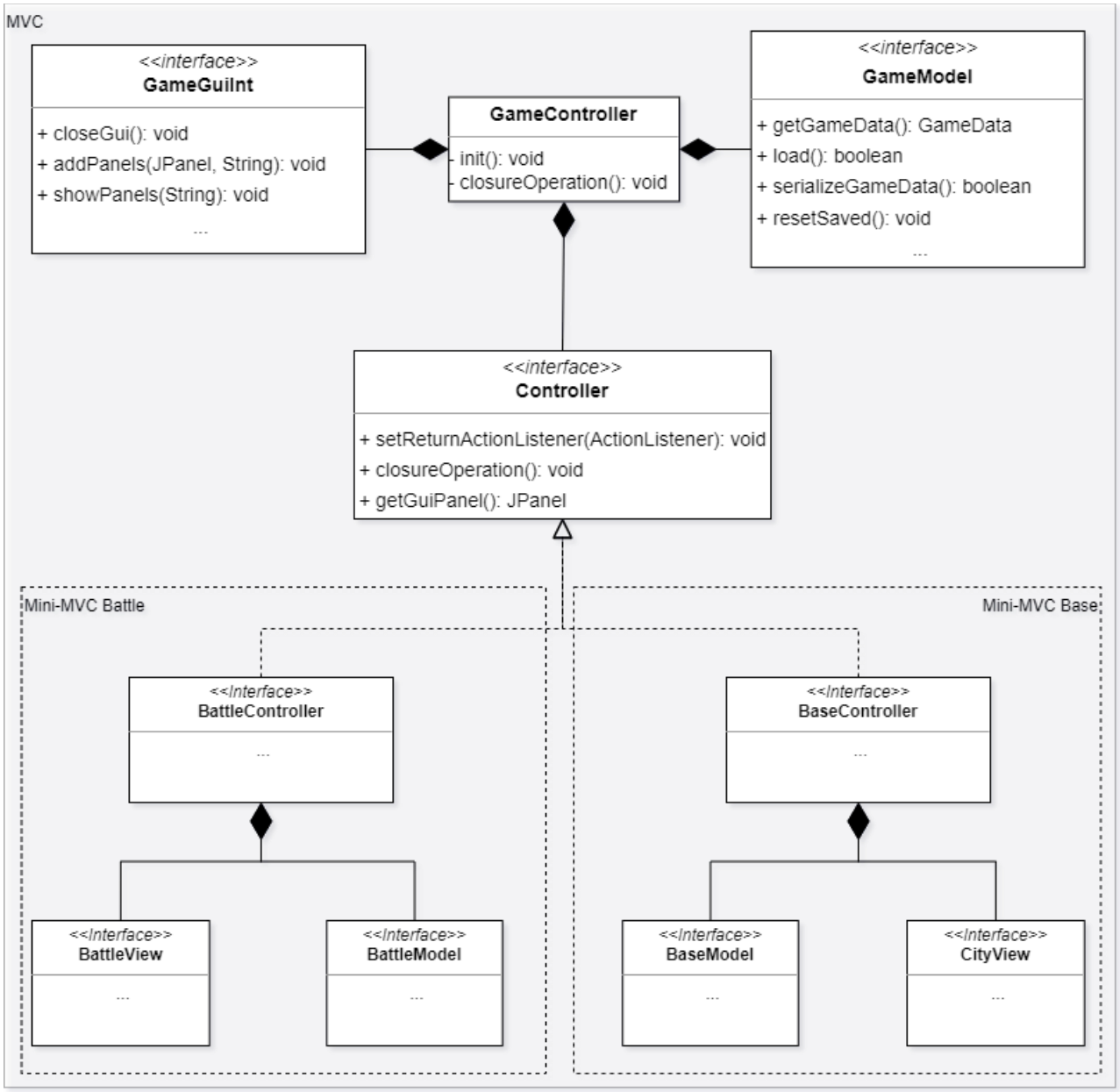

Figura 2.0: Schema UML – Architettura dell'applicazione.

2.1 Architettura

Il progetto è stato implementato seguendo la filosofia del pattern architetturale MVC (Model-View-Controller) utilizzandolo per suddividere le varie parti del gioco in servizi distinti fra loro.

View – GameGuiInt: la view consente la registrazione di nuovi pannelli, permettendo su richiesta di impostare la visibilità di un solo panello alla volta.

Model – GameModel: si occupa della logica di serializzazione e de-serializzazione fungendo anche come archivio per i dati di gioco.

Controller – GameController: il controller agisce come un nodo di comunicazione tra le diverse parti dell'applicazione. In particolare, rende possibile il flusso dei dati dal GameModel verso i servizi e registra nella GameGuiInt i pannelli provenienti da essi.

Definizione di servizi – (Battle/Base): Ciascun servizio viene costruito seguendo l'architettura MVC e rappresenta un'area specifica dell'applicazione, come la base del giocatore oppure il campo di battaglia. Questo approccio modulare permette di separare la responsabilità tra le varie logiche di gioco.

2.2 Design Dettagliato

Marco Francolini

Base Model

Il compito del BaseModel è di gestire le strutture, risorse e gli aggiornamenti delle truppe del giocatore, utilizzando le risorse prodotte dalle strutture durante la partita. Inoltre, il BaseModel deve assicurarsi che le azioni che vengono compiute sui vari elementi rispettino determinati criteri e nel caso in cui ciò non accada deve essere in grado di mandare una forma di feedback appropriato. Quindi per la sua natura, il base model è un problema articolato che va scomposto.

Gestione strutture, risorse e truppe

Uno dei primi problemi presentati in fase di progettazione è come permettere all'utente di costruire strutture e aggiornare truppe con caratteristiche che in futuro potrebbero cambiare o che in base a quello che avviene durante la partita possono essere modificate.

Per affrontare il problema delle costruzioni, l'utilizzo della builder pattern assieme alle classi dati è sembrata la scelta architetturale più adeguata portando alla creazione di Building e BuildingBuilder. In questo modo si possono creare delle strutture con valori predefiniti in maniera semplice per permettere di gestire l'evoluzione del costo e della produzione degli edifici durante la partita, oppure l'aggiunta di nuove costruzioni, lasciando una libertà sia di scegliere il modo in cui le strutture vengono aggiornate a livello implementativo che a livello di regole di gioco.

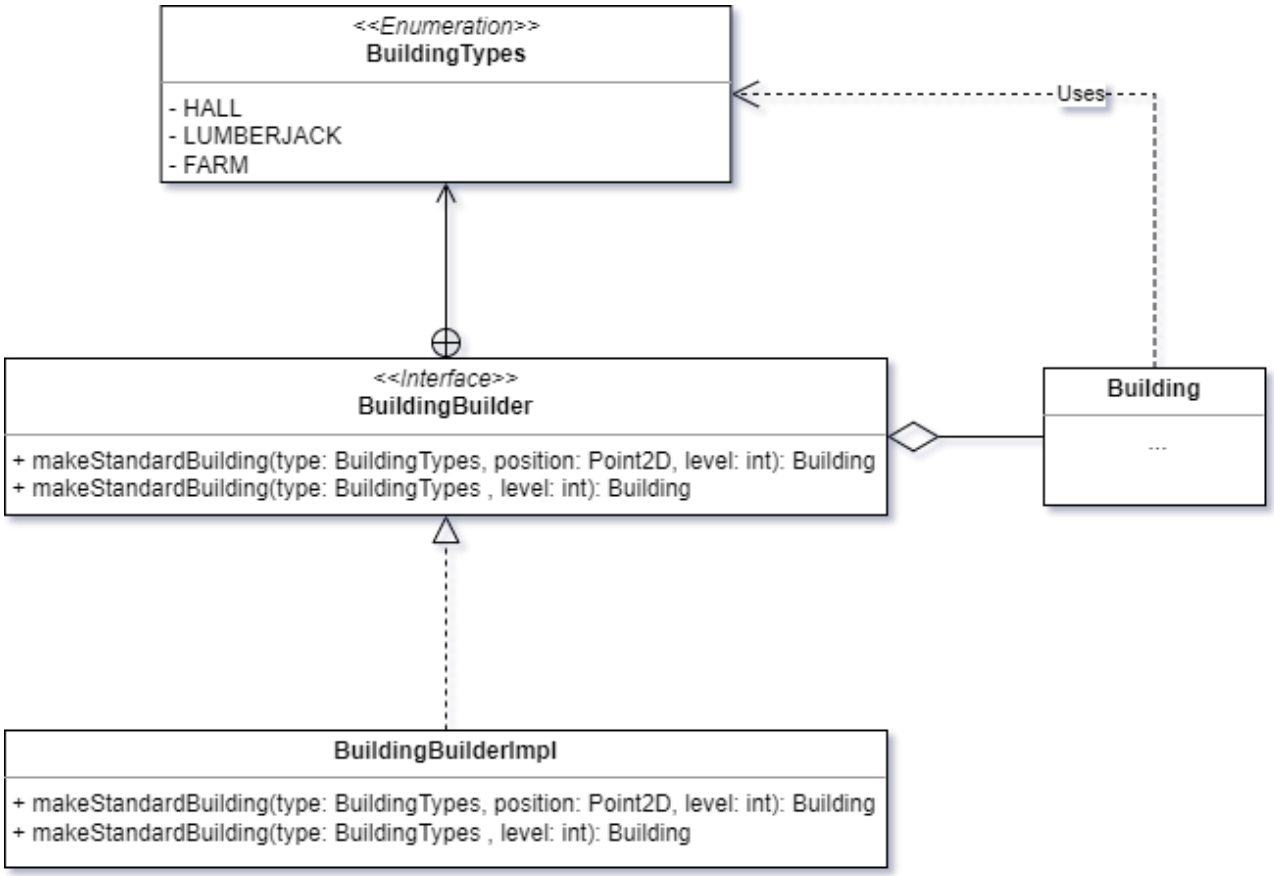

Figura 2.1: Schema UML della classe dati Building e l'interfaccia BuildingBuilder.

Le risorse vengono trattate in una maniera molto simile, tuttavia essendo elementi più semplici, l'utilizzo della pattern builder non è stata ritenuta necessaria, la soluzione più efficace è risultata la creazione di una classe con un'enumerazione che ne definisce il tipo e un valore che ne rappresenta la quantità, implementando anche qualche funzione che ne semplifica la manipolazione.

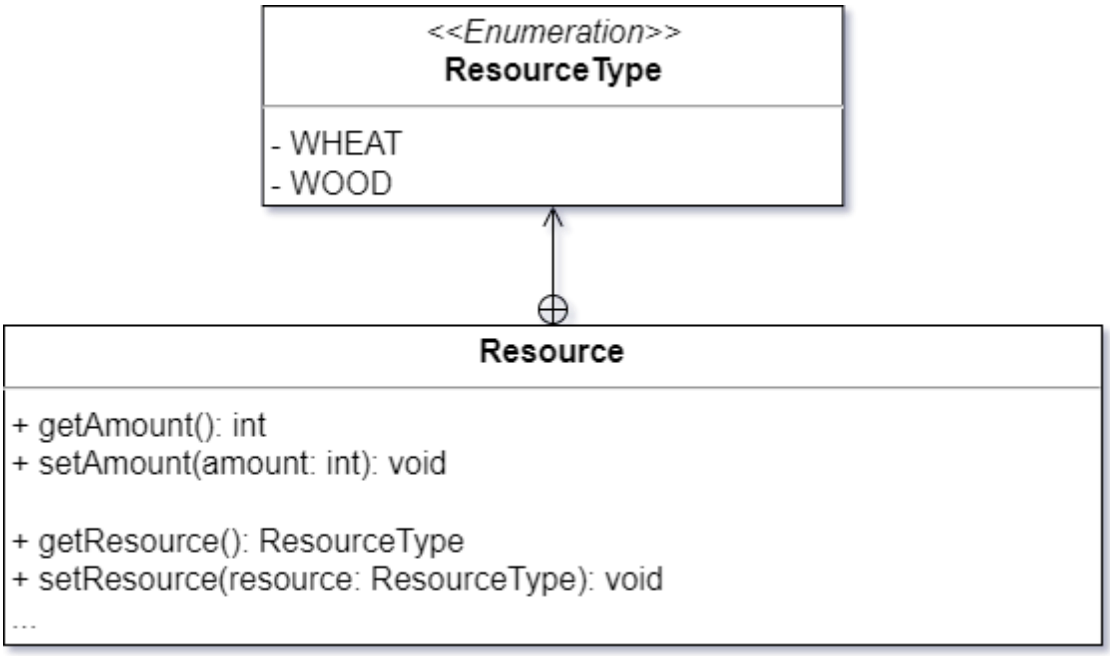

Figura 2.2: Schema UML della classe dati Resource e l'enumerazione che ne definisce il tipo.

In fine, le truppe vengono trattate in una maniera ancora più semplice, poiché l'utilizzo di queste all'interno del BaseModel è molto limitato, la soluzione è risultata sempre la costruzione di un'enumerazione che ne definisce il tipo.

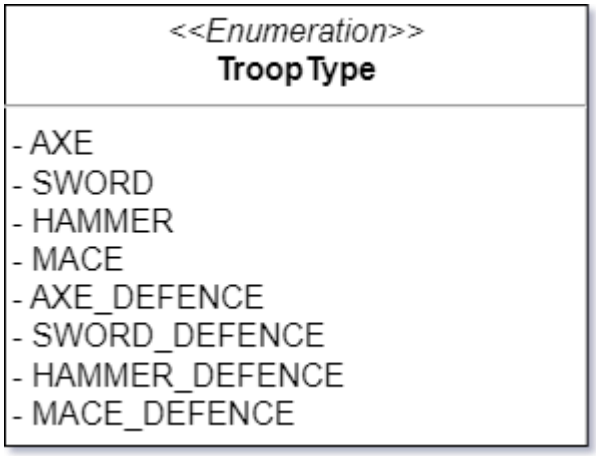

Figura 2.3: Schema UML dell'enumerazione TroopType che definisce il tipo di truppa.

Come si può notare, per motivi di semplicità, l'approccio di progettazione più comune è stato l'utilizzo di enumerazioni per definire le tipologie di strutture, truppe e risorse. Questa scelta lascia spazio all'implementazione di qualsiasi metodo interno per la gestione di questi elementi, ma allo stesso tempo limita la possibilità di cambiarne la quantità. Questo comportamento è intenzionale, in quanto l'introduzione di un maggior numero di tipologie potrebbe sbilanciare pesantemente le regole del gioco.

Sistema di Eccezioni

Ritornare una forma di feedback all'invocazione di una qualsiasi funzionalità che potrebbe

essere utilizzata in maniera scorretta ha presentato un problema importante, poiché ci sono diversi modi in cui l'utente può interagire con questa parte del gioco aprendo anche tantissime possibilità in cui si possono verificare errori che devono essere gestiti oppure fatti notare all'utente.

La soluzione scelta per questo problema è stato l'utilizzo delle eccezioni appoggiato da un sistema di ereditarietà per identificare gruppi di problemi che si possono verificare nello stesso contesto.
Questo metodo permette di identificare il problema o il gruppo che lo identifica in maniera corretta e cercare di gestire l'errore oppure notificare l'utente che si è verificato un determinato errore.

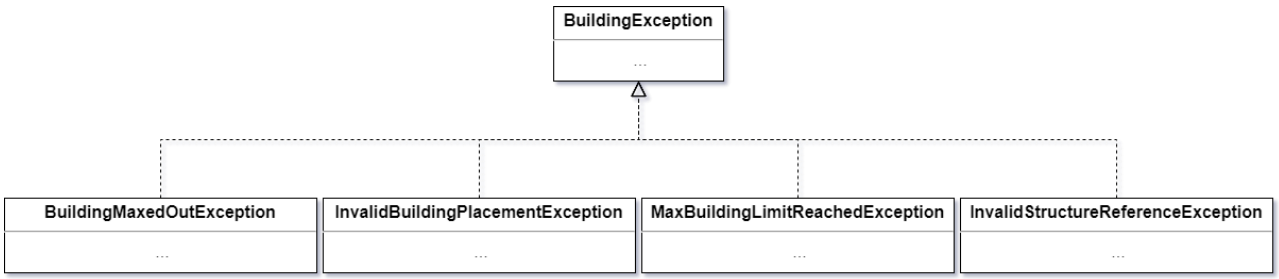

Figura 2.4: Schema UML del sistema di gerarchia delle eccezioni relative alle Building

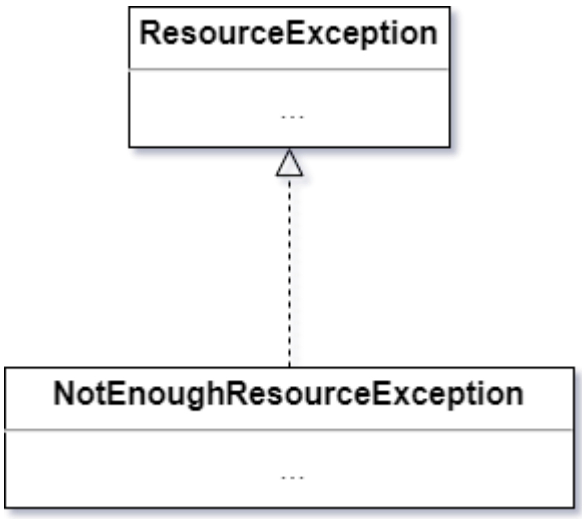

Figura 2.5: Schema UML del sistema di gerarchia delle eccezioni relative alle risorse

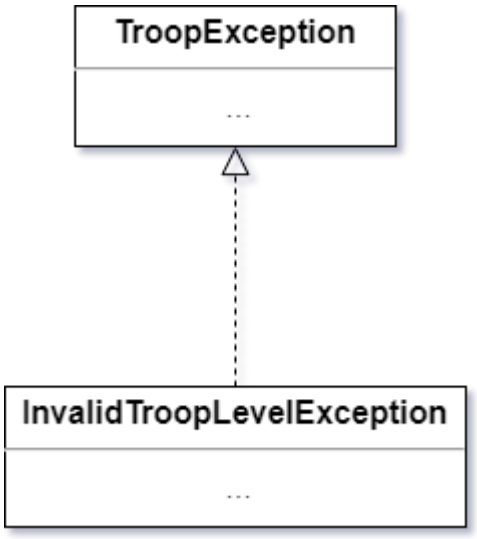

Figura 2.6: Schema UML del sistema di gerarchia delle eccezioni relative alle truppe

Al di fuori del BaseModel, queste eccezioni vengono prese dal controller per essere successivamente mostrate a schermo all'utente.

Thread Manager

La natura della di costruzione di una base del gioco implica la necessità di dover eseguire un numero indeterminato di operazioni potenzialmente cicliche dato un tempo limite, consentendo in ogni momento di mettere in pausa una qualsiasi operazione assegnata. Questo permette di creare l'illusione di un progressivo avanzamento di azioni specifiche.

Inizialmente, l'idea di utilizzare un game-loop per gestire queste operazioni cicliche, simili a intervalli di tempo, è stata presa in considerazione. Tuttavia, con l'evolversi della progettazione è diventato sempre più evidente che il gioco sarebbe stato event-driven.

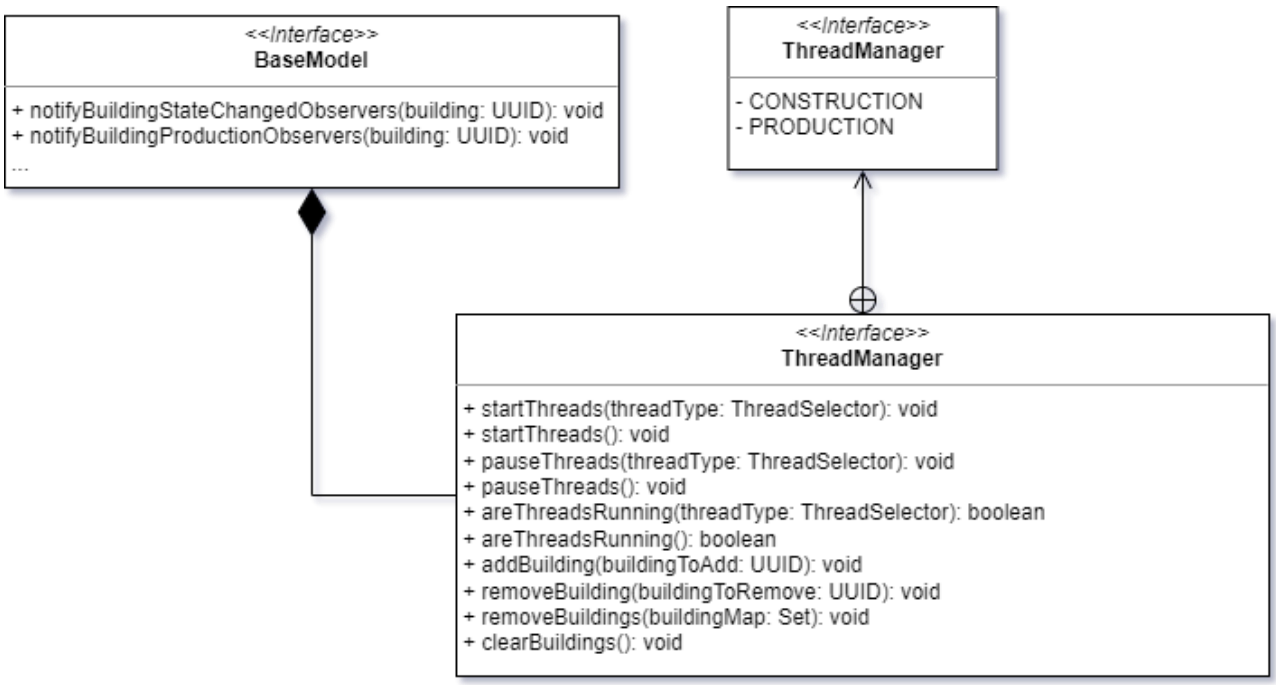

Figura 2.7: Schema UML che rappresenta la struttura del ThreadManager.

Il percorso di sviluppo si è quindi orientato verso l'adozione del pattern delle pool di thread.

Dal momento che i thread vengono utilizzati con lo scopo di gestire le strutture, esistono due tipologie di thread che corrispondono a due stati possibili delle strutture durante la partita: il "construction" thread che si occupa della costruzione delle strutture, e il "production" thread che si occupa di produrre periodicamente delle risorse da strutture già costruite, creando una pool di thread per ogni tipo.

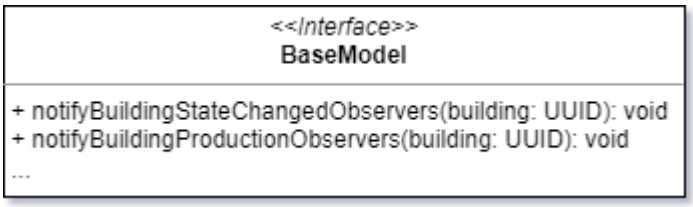

Figura 2.8: Schema UML rappresentante il sistema di notifica del BaseModel.

Per il sistema di notifica si utilizza il BaseModel come tramite per comunicare a chiunque sia in ascolto selezionando il metodo giusto definito alla thread pool alla quale il thread responsabile appartiene.

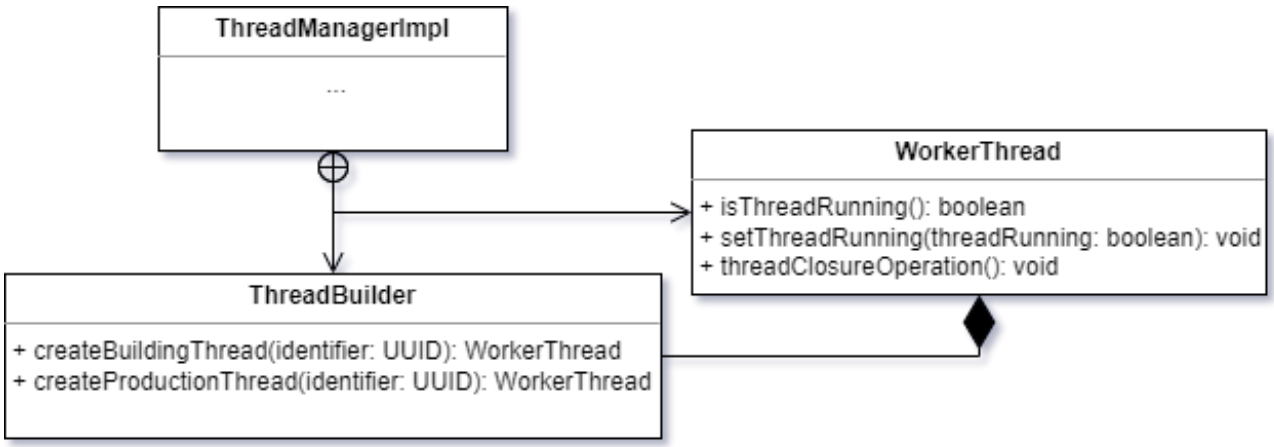

Figura 2.9: Schema UML che rappresenta la struttura interna alla implementazione del ThreadManager.

Dal momento che ogni thread ha un task specifico da eseguire su ogni tipo di struttura, è stato importante l'utilizzo della builder pattern creando la classe "ThreadBuilder" per generare dei thread specializzati prima di assegnarli alla thread pool.
La struttura principale del thread "WorkerThread" è molto semplice, dal momento che della funzionalità principale se ne occupa il builder, possiede solamente delle funzioni per manipolare e sondare il suo stato di esecuzione.

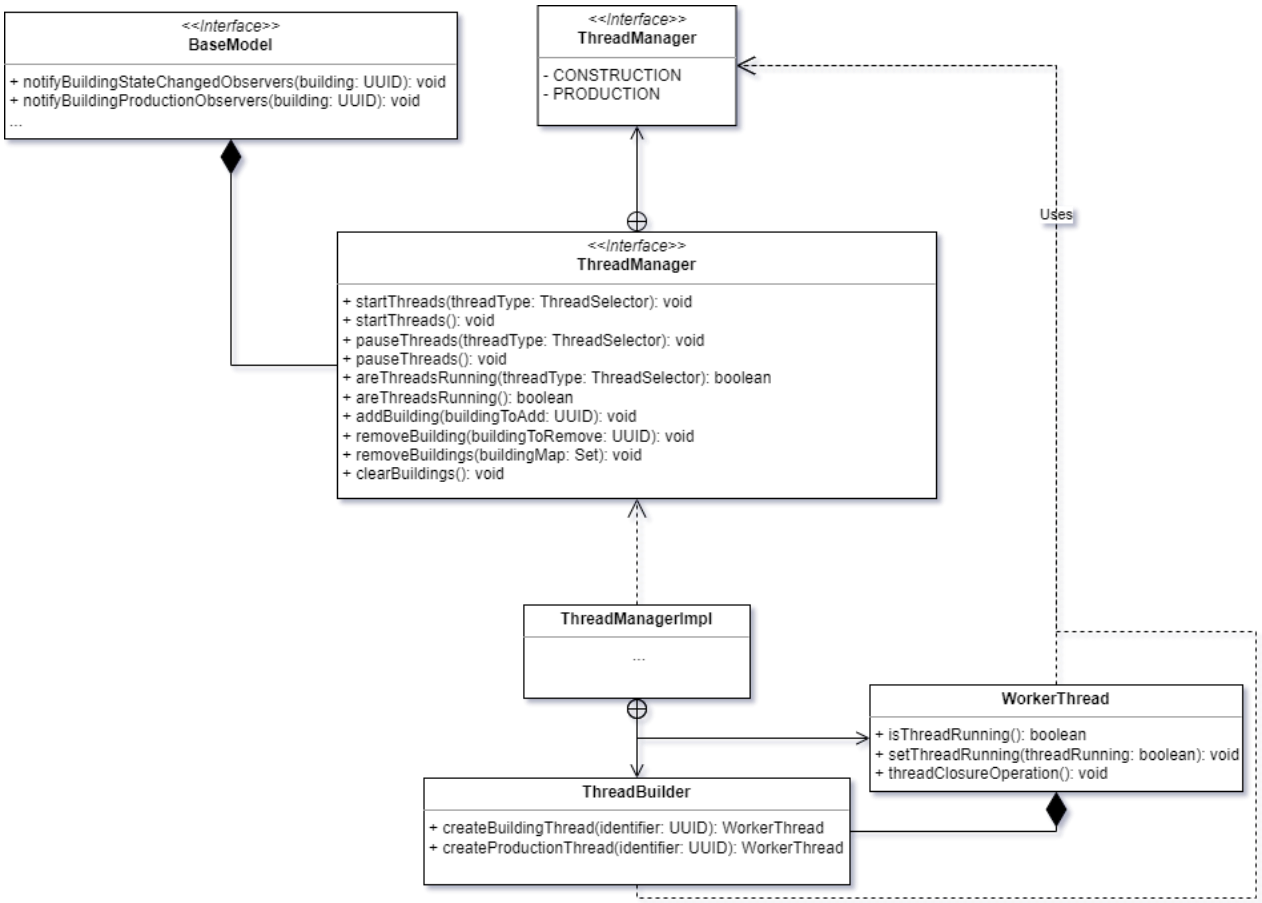

Figura 2.10: Schema UML completo del ThreadManager.

Il Thread manager stesso si è rivelato forse uno dei problemi più intricati da risolvere, poiché entrano in gioco diversi fattori, tra cui l'integrità dei dati con l'utilizzo di molteplici thread e la gestione di un sistema di thread che può essere interrotto in un qualsiasi momento.

BaseModel
Il BaseModel è un'aggregazione di tutti i pezzi sopra descritti, creando un sistema che sfrutta tutti gli elementi a sua disposizione per gestire correttamente la base del giocatore.

Rappresentando la logica di tutte le operazioni che avvengono durante la fase in cui il giocatore è in base, il BaseModel prende una parte importante della progettazione del gioco. Nonostante ciò, deve comunque seguire la filosofia di un model, quindi deve operare senza comunicare direttamente con i livelli superiori al model, ma al massimo può predisporre dei "ponti" di comunicazione con i livelli superiori utilizzando l'observer pattern, in questo modo ogni livello può registrarsi in modo da ascoltare gli eventi che avvengono all'interno del BaseModel.

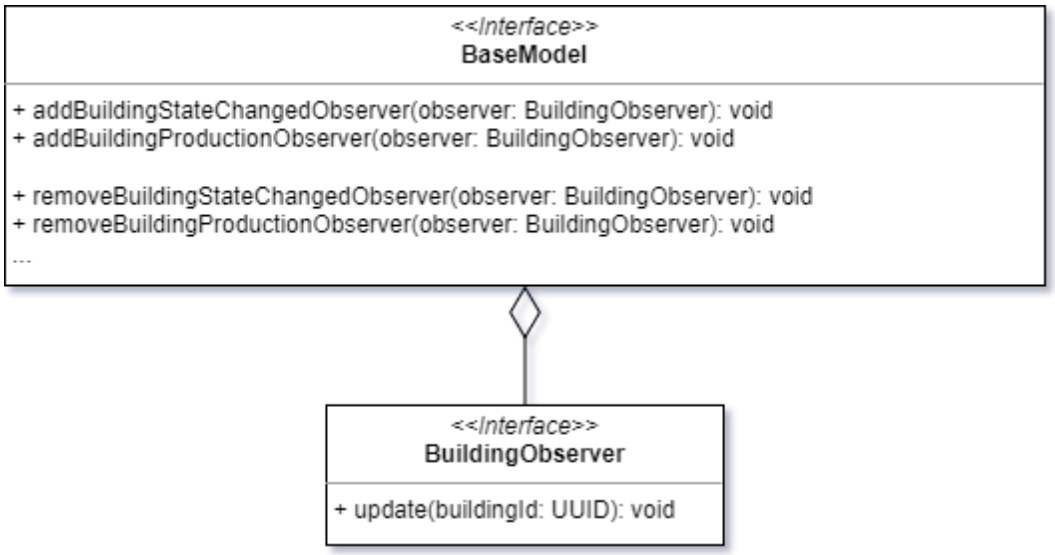

Figura 2.11: Schema UML che rappresenta la struttura dell'observer pattern all'interno del model.

Configurazione Gioco

Per garantire una chiara separazione tra la logica di gioco e la gestione dei dati si è scelto di creare un sistema di configurazione dei dati di gioco tramite la libreria GSON, che risulta particolarmente efficiente e facile da usare. Questo sistema ha reso più facile la modifica e la gestione degli aspetti di gioco senza dover intervenire direttamente nel codice. Nel file json di configurazione è possibile definire vari parametri come: il numero di basi nemiche, il numero di edifici piazzabili, il numero di vite e molto altro.
In aggiunta per evitare in modo sicuro problematiche in fase di installazione si è deciso di creare un setup iniziale di questo file tramite classi di configurazione.

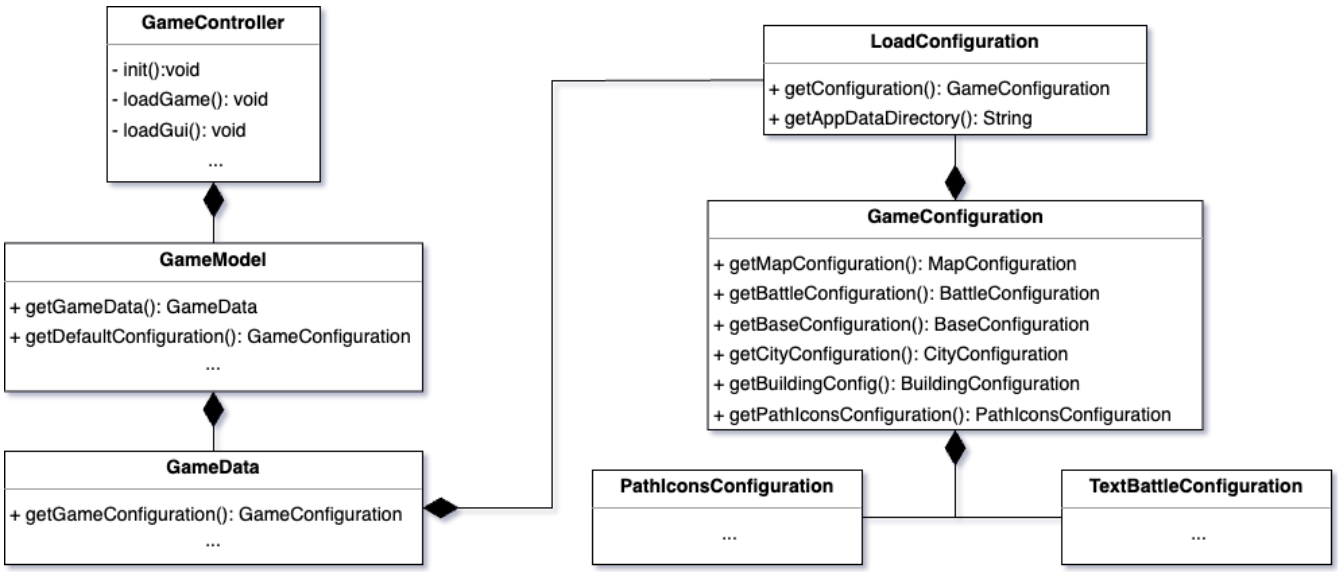

Figura 2.12: Schema UML della Configurazione del gioco

LoadConfiguartion
Problema Avere la sicurezza di caricare i dati di gioco correttamente in fase di setup.

Soluzione Nel caso di problemi creare i dati di gioco di default come specificato nelle classi di configurazione.

GameConfiguration
Problema Effettuare una corretta divisione logica fra dati e logica di gioco. Avere la possibilità di cambiare aspetti dell'applicazione senza intervenire nel codice. Evitare Hard Coding. Coordinare il setup delle Views e dei Model.

Soluzione Creazione di classi di configurazione custom per ciascuna entità dell'applicazione.

PathConfiguration
Problema Raggruppare i PATH delle immagini usate dalle view in un unico file in modo da velocizzare modifiche grafiche dell'applicazione e soprattutto senza modificare codice.

Soluzione Creazione di una classe di configurazione contenente i path delle immagini.

View – Battaglia

Il pannello della battaglia è stato sviluppato usando il layout manager CardLayout per poter switchare tra un pannello tutorial e il pannello di gioco. Il pannello di gioco risulta essere quello principale e viene gestito da un BorderLayout, in cui ciascuna parte è definita da un pannello unico con funzionalità ben precise. Questa implementazione ha permesso un elevato livello di incapsulamento e isolamento dei componenti, garantendo che ogni sottoparte offrisse dei servizi ben precisi a quelle superiori. Il controller che gestisce la view infatti interagisce solo con il pannello di primo livello avendo però la possibilità di controllare entità di qualsiasi livello della view.

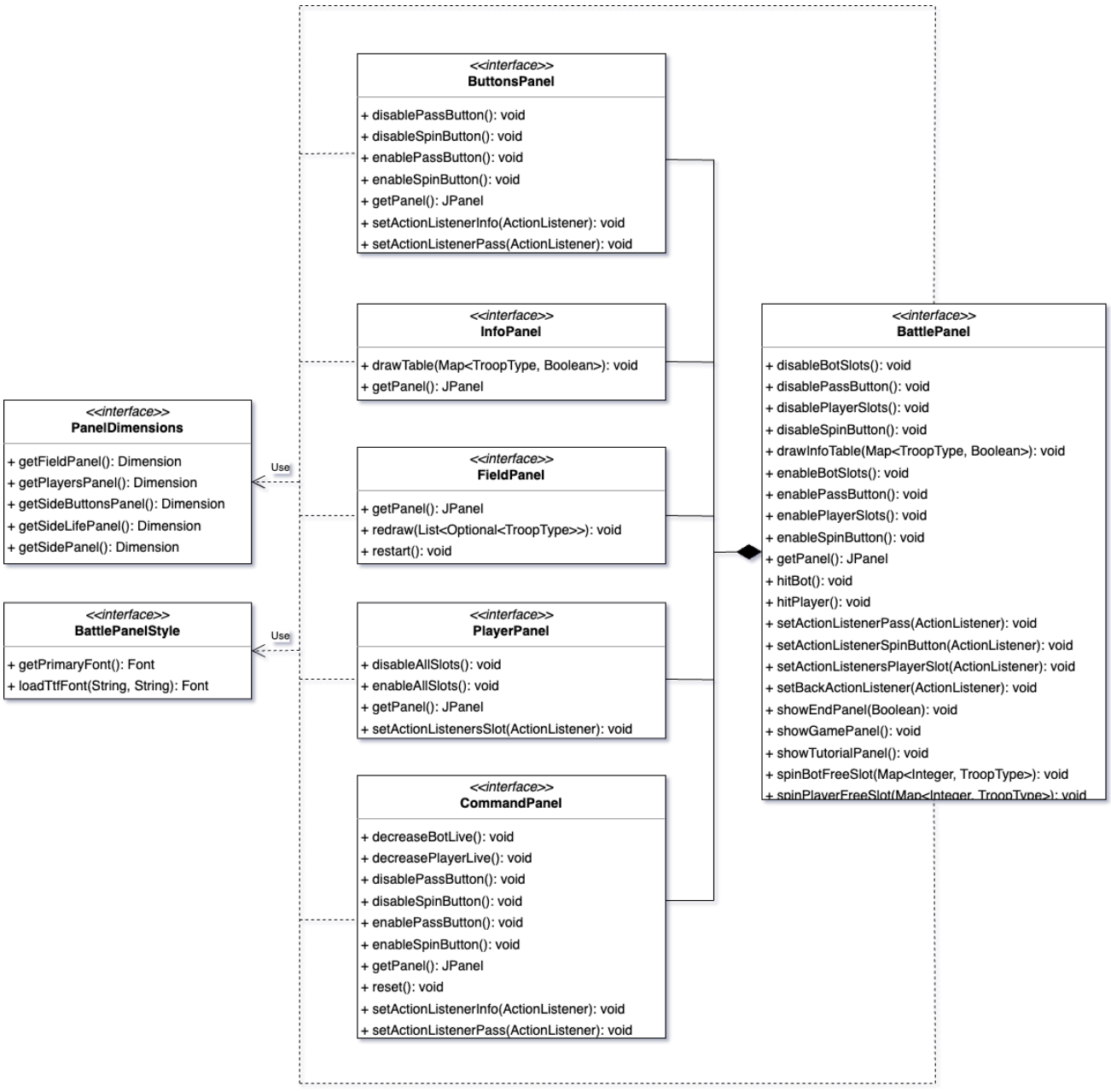

Figura 2.13: Schema UML view della battaglia.

BattlePanel
Problema Rispettare il design architetturale MVC e semplificare operazioni di manutenzione e di aggiornamento.

Soluzione Uso del Layout Manager BorderLayout ed interfacce definite delle sottoparti per suddividere le funzionalità della view. Uso di un incapsulamento corretto per rendere la view sicura ma allo stesso tempo flessibile da chi la controlla.

BattlePanelStyle
Problema Rendere i componenti della view graficamente uniformi e facili da modificare

Soluzione Uso di un'interfaccia che definisce colori e font usati nella view.

PanelDimensions
Problema Dimensioni dei pannelli dinamici.

Soluzione Uso di un'interfaccia che definisce le dimensioni dei sottopannelli tramite percentuali fisse rispetto l'intera risoluzione dello schermo.

Controller – Game

Il Controller ha il compito di registrare i vari pannelli ricevuti dai servizi nella View. Questa registrazione avviene durante l'inizializzazione del gioco e consente al Controller di avere un accesso diretto e organizzato sulla view. Il Controller gestisce inoltre la comunicazione tra la View e il Model, fornendo indicazioni su come gli elementi dei pannelli devono reagire a eventi specifici.

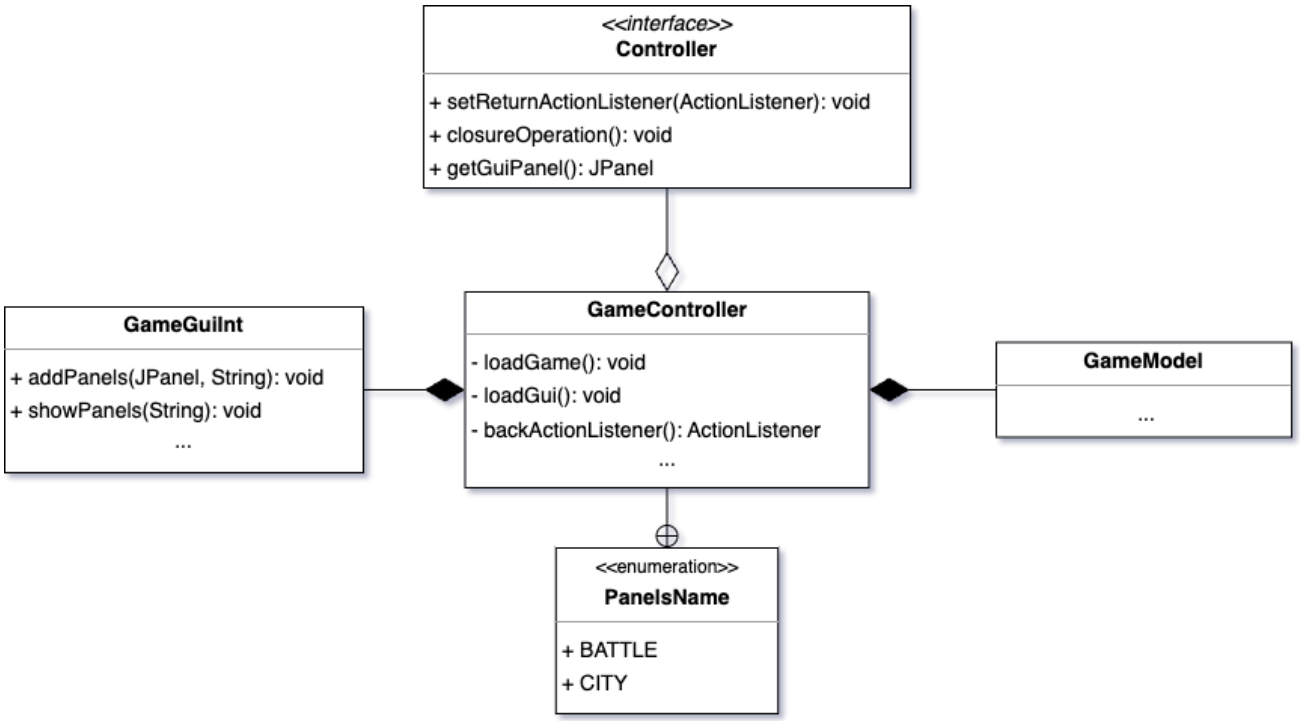

Figura 2.14: Schema UML Game Controller

Problema Registrazione e coordinamento della view.

Soluzione L'uso di un enum che definisce un identificativo per ogni pannello registrato nel CardLayout della View. E la definizione di un ActionListener per i pannelli capace di indirizzare sempre la View nel pannello principale.

Model – Game

Il Model è responsabile dell'organizzazione e l'integrità dei dati di gioco. Si occupa inoltre del salvataggio e caricamento del gioco. In esso è contenuta la logica di inizializzazione dell'applicazione.

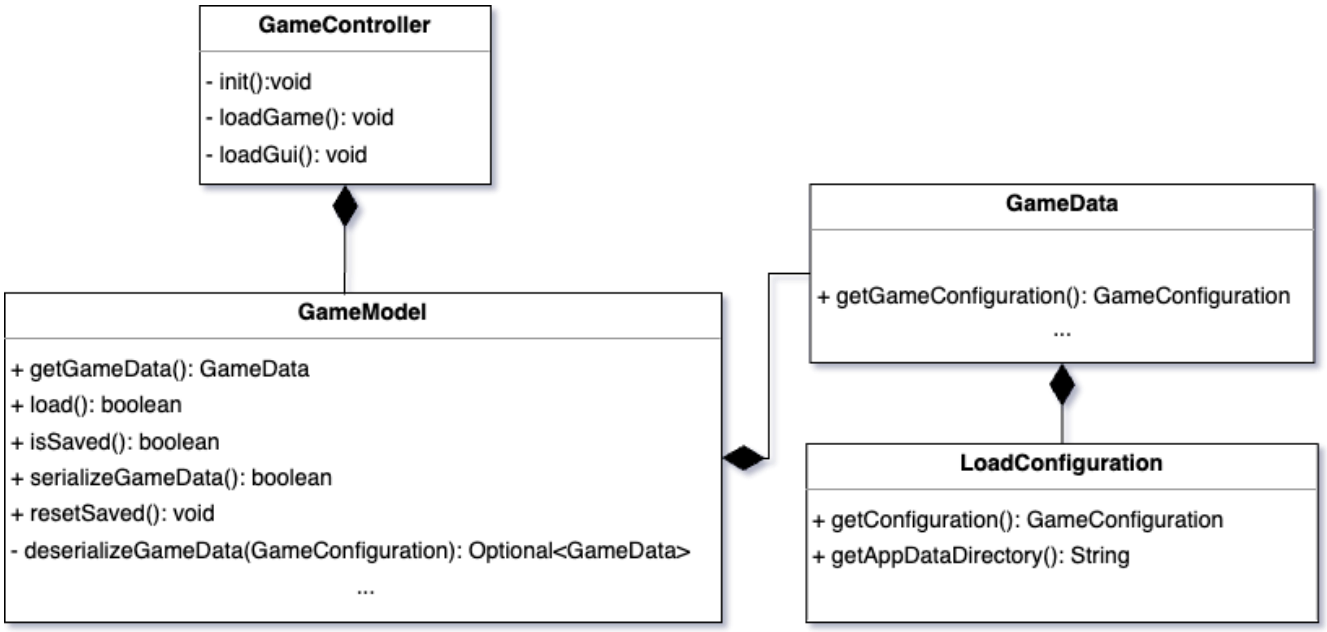

Figura 2.15: Schema UML Game Model

Problema Salvataggio.

Soluzione Uso della serializzazione. Si è deciso di serializzare anche la configurazioni dei dati di gioco per evitare che un salvataggio cambi nel caso cambi il file di configurazione.

Sound

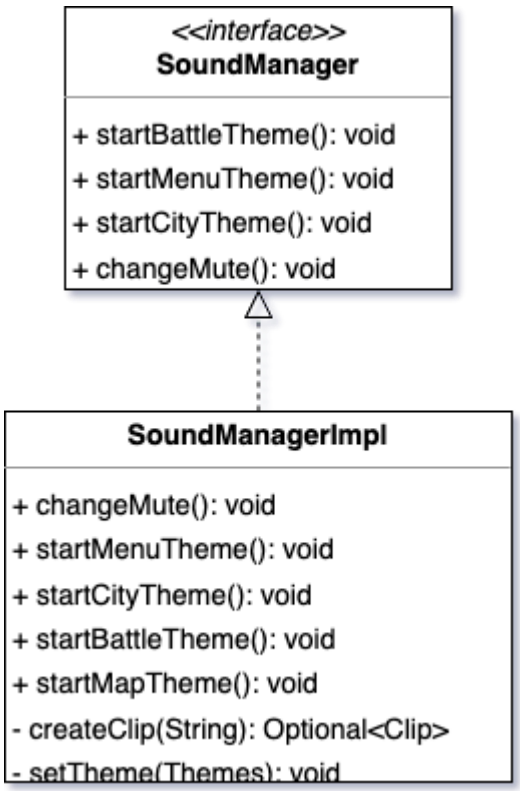

Figura 2.16: Schema UML Sound Manager

Problema Aggiungere musica di sottofondo al gioco.

Soluzione Creazione di un Sound Manager capace di cambiare la traccia, stopparla o farla ripartire.

Abdoulaye Kane

View-Città

Problema

durante la scrittura del codice bisognava trovare la struttura corretta per la realizzazione della parte view nell'mvc della city.

Soluzione

ho scelto di adottare un metodo che mostri un pannello principale appoggiandomi sempre all'mvc per la parte view. La challenge principale è stata l'utilizzo di più layout all'interno della view in modo che potessero svolgere le loro funzioni in contemporanea. Ho diviso la view principale in tre pannelli: due piccoli e uno grande. Ho disposto una piccola pulsantiera in alto che possa permettere all'utente di svolgere tutte le azioni possibili in maniera semplice e sono funzionali a quella view e al campo di gioco. Nella parte in basso si trova la pulsantiera che interagisce con il resto del gioco, mentre per dare più spazio alla parte principale della schermata lo spazio restante viene occupata dal campo di gioco.

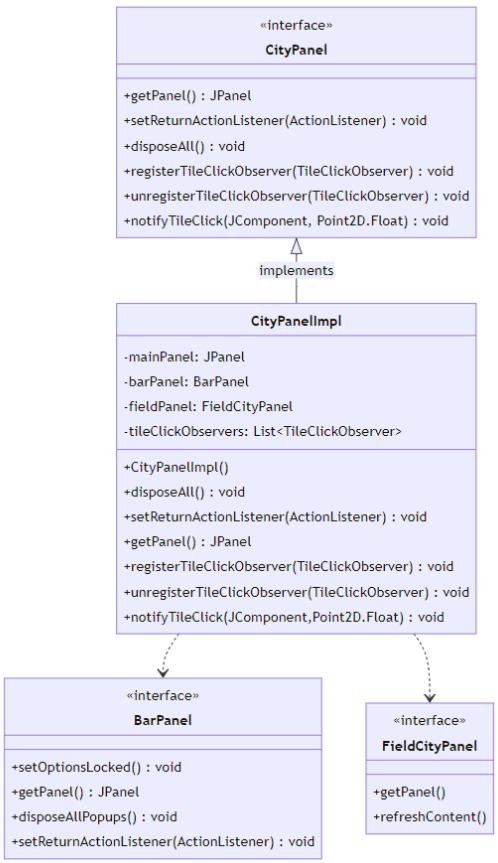

Figura 2.17: Schema UML view della città

Creazione dei popup e dei messaggi di avviso

Problema

Nella parte in alto della schermata sono presenti dei bottoni con le azioni che l'utente può compiere all'interno della view che oltre a potenziare e distruggere le strutture che forniscono risorse in base al tipo, c'è la possibilità di incrementare il livello delle truppe

Soluzione

per creare dei messaggi di avviso e piccole schermate che rendano il gioco più semplice e giocabile ho realizzato dei popup che permettono all'utente di svolgere delle operazioni all'interno del gioco o avvisano il giocatore di star compiendo azioni che non sono permesse e rimangono aperti all'interno del loro contesto e alla chiusura non interrompono il gioco.

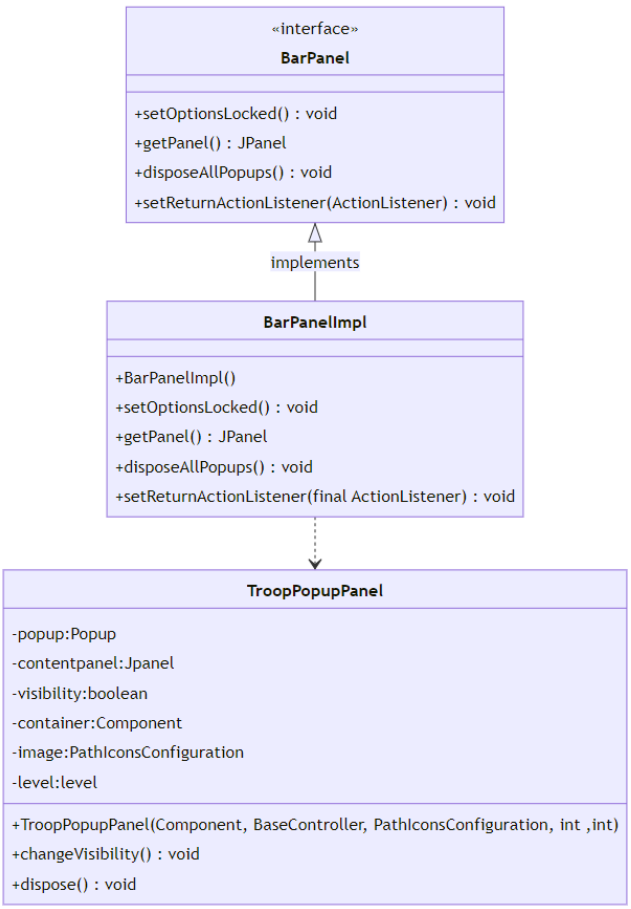

Figura 2.18: Schema UML view del Popup implementato nella BarPanel

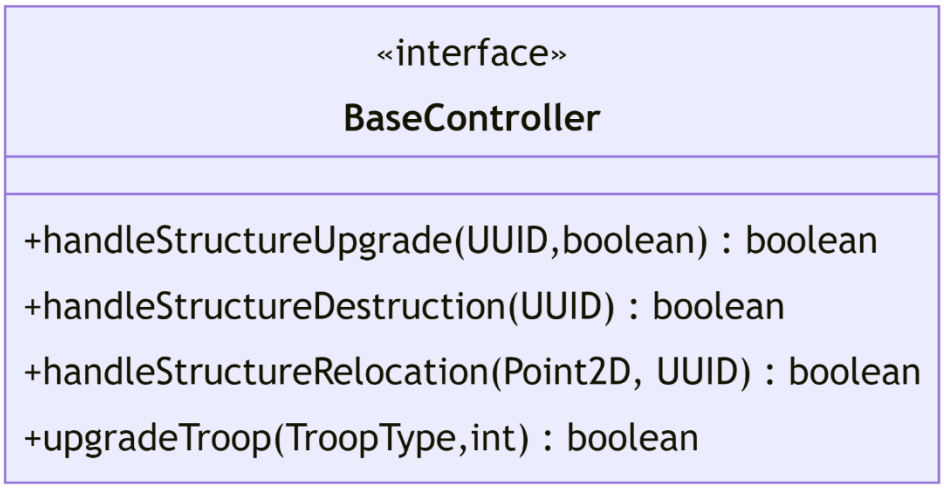

Figura 2.19: Metodi del BaseController che generano i messaggi di avviso all'utente

Configurazione delle texture

Problema

per fare in modo che nella mappa venga caricata la struttura desiderata era necessario caricare la texture ed assegnarla ad ogni UUID in modo che si potesse caricare sulla mappa la struttura corretta

Soluzione

All'interno della città c'è la possibilità di poter piazzare delle strutture che incrementano le risorse che vengono utilizzate in gioco. Ogni struttura è caratterizzata da una produzione differente di risorse a livello quantitativo. Per poter distinguere correttamente le struttura da ognuna di essa viene assegnato un identificatore in maniera tale che quando viene piazzata la struttura tramite essa venga mostrata quella corretta. Quindi ho assegnato il suo URL al tipo di struttura e poi la struttura stessa ad un'immagine che verrà associata al proprio identificatore.

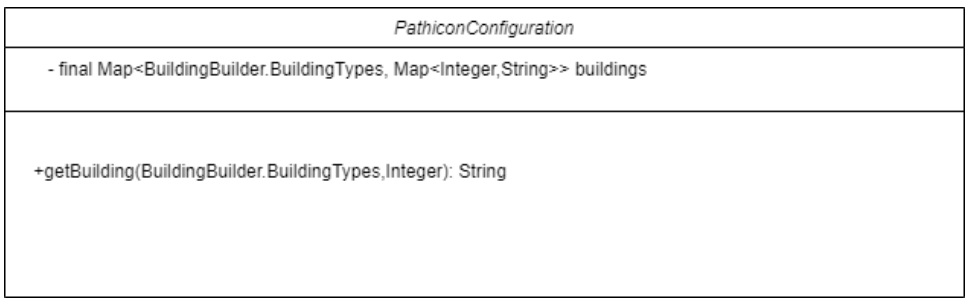

Figura 2.20: Implementazione delle texture della classe PathiconConfiguration

Lorenzo La Farciola

Funzionalità battaglia (entità)

Problema: Creare una modalità di battaglia contenente scelte sia casuali che strategiche. Elaborare un modo efficace per permettere alle entità di agire sul campo (manipolare i dati attraverso i metodi che hanno), immagazzinando le informazioni presenti su ogni cella.

Soluzione: Ho deciso di creare una classe "pattern" chiamata EntityData. Questa classe viene usata per il contenimento dei dati relativi alle entità sul campo di battaglia e sia il "bot" che il giocatore sfruttano la stessa. Contiene in particolare metodi per la manipolazione dei dati riguardanti, ad esempio, la selezione di truppe, metodi di ordinamento e rimozione o aggiunta di truppe in campo. EntityData fa operazioni su una mappa di interi (posizione) e di CellsImpl. Quest'ultima citata è una classe creata apposta per contenere le più basilari informazioni (che truppa è presente e se è stata cliccata).

Agendo sulla mappa, sono in grado di capire in ogni posizione del campo i dati inerenti alla precisa cella scelta.

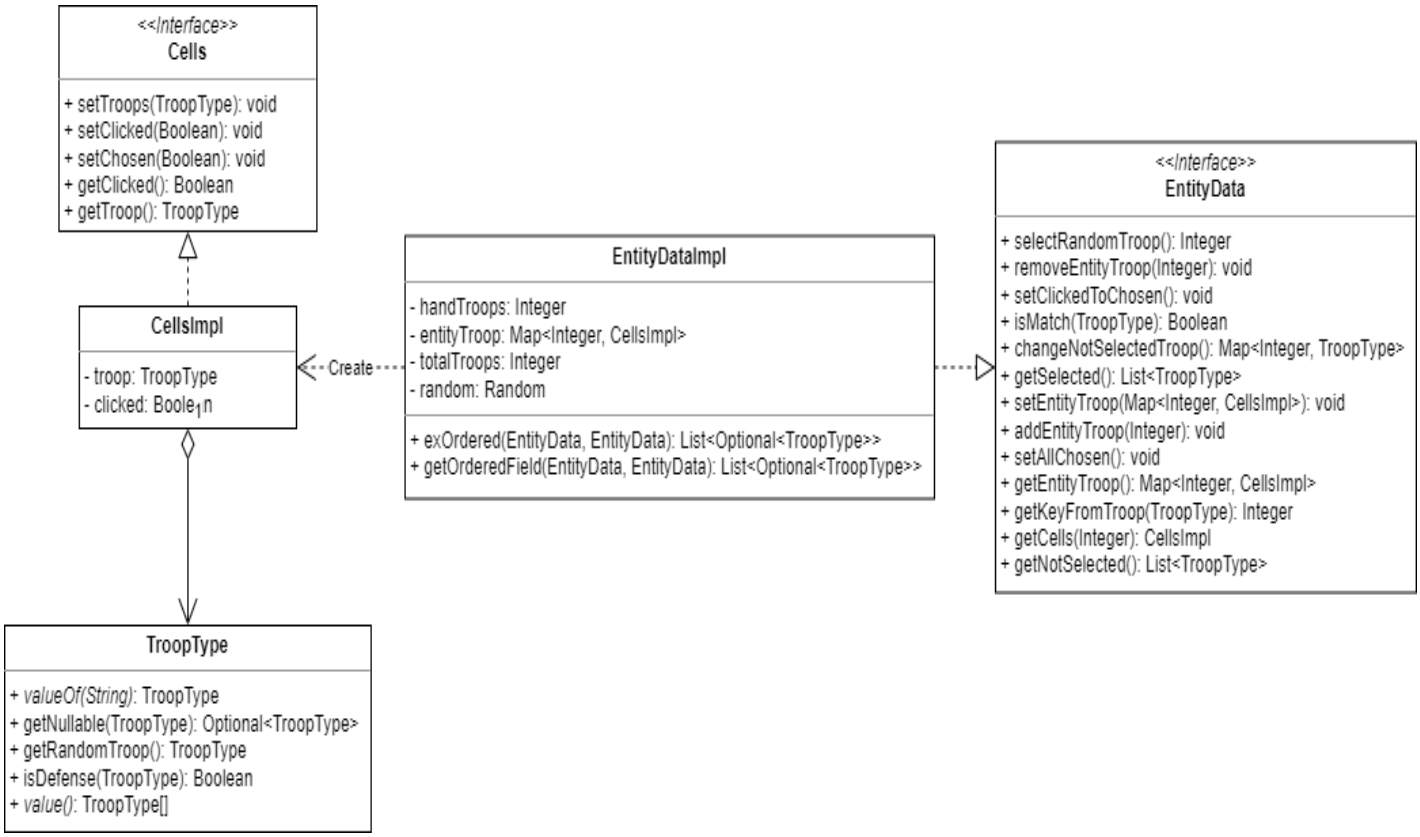

Figura 2.21 implementazione di EntityData e del suo sistema architetturale in UML.

Funzionalità battaglia (Model)

La classe model chiamata "BattleModel" si occupa delle operazioni generiche e di tutto ciò che avviene dietro ai pulsanti e alle scelte del bot. I pulsanti premibili dal giocatore sono parallelamente gli stessi che usa il bot (se l'utente può usare lo spin, anche il bot può). Per tanto il model contiene un metodo per ogni pulsante o azione che avviene in aggiunta a funzioni per il reset della battaglia e per la tabella di confronto truppe a sinistra.

Problema: Come fare agire il bot. Indecisione se assegnargli azioni di natura solo randomica oppure programmarlo in modo che un minimo decidesse cosa fare in base all'avversario.

Soluzione: Nella funzione pass, alla premuta quindi del giocatore, avviene tutto ciò che fa il bot in quel momento. Sono presenti confronti e chiamate ad altri metodi per vedere cosa il giocatore ha messo sul campo, e, in base anche a cosa ha in mano il bot, esso è obbligato a mettere determinate truppe. In caso di fallimento o di mancanza di truppe opposte a quelle del giocatore, le scelte avverranno casualmente.

Applicando il metodo scritto, il bot agisce sia con una logica impostata, sia casualmente.

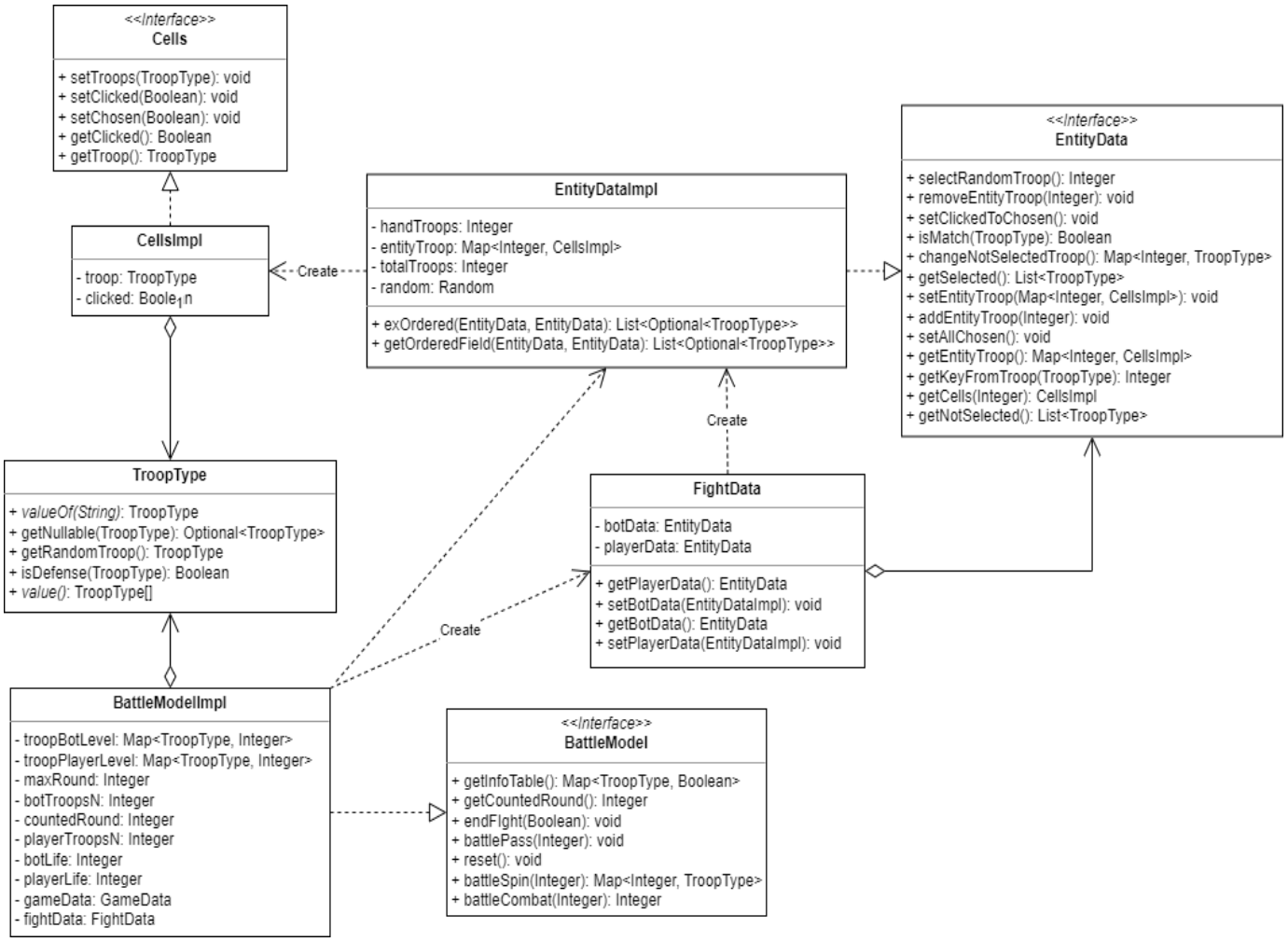

Figura 2.22 UML generale di ogni classe inerente al funzionamento del BattleModel.

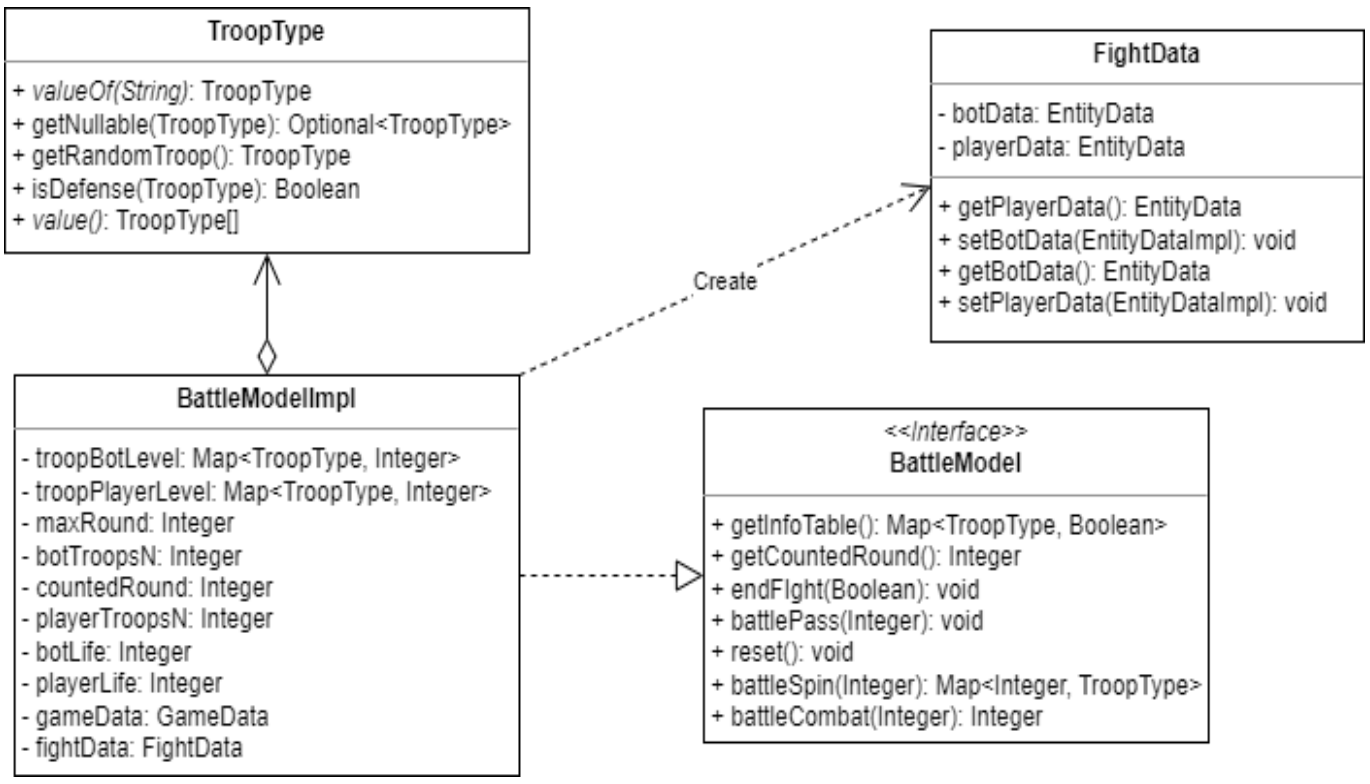

Figura 2.23 implementazione BattleModel e funzionamento in schema UML.

FightData

FightData è una classe che si occupa dell'inizializzazione delle istanze entità, contenendo, quindi, tutte le entità della battaglia.

Problema: C'è stata indecisione se fare FightData Serializzabile, di conseguenza se farla Optional oppure no. FightData si trova dentro la classe principale di contenimento dati "GameData", e quest'ultima viene serializzata al salvataggio e uscita del gioco. Ma se l'utente salva senza aver iniziato una battaglia? In questo caso non si ha mai la certezza dell'esistenza certa di FightData (da qui il motivo del perché usare l'Optional).

Soluzione: FightData viene creata al momento di inizio battaglia e l'utente non può salvare durante il combattimento, per tanto non ha bisogno di essere serializzata (quindi ad ogni battaglia viene ricreata e non è un Optional). Le uniche informazioni utili sulla battaglia vengono salvate in altre classi serializzate (punti vita del giocatore e del bot, livello truppe bot, numero massimo dei round).

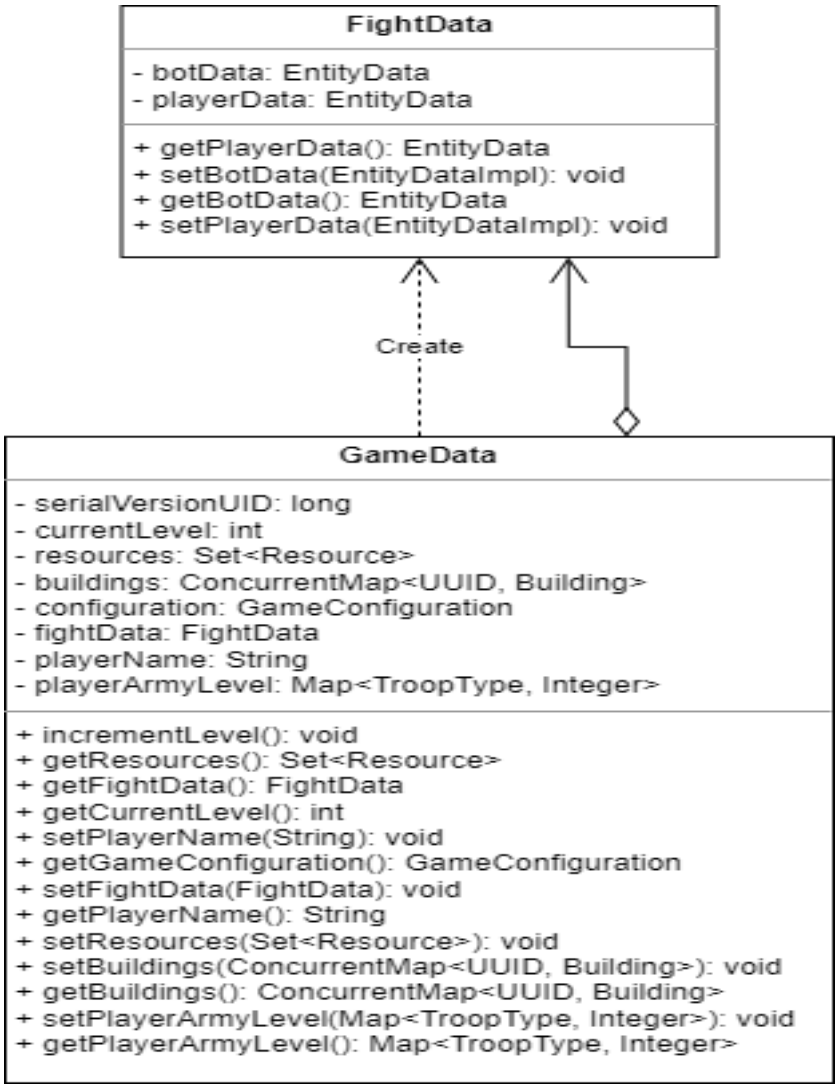

Figura 2.24 rappresentazione di FightData in UML.

Architettura GUI del gioco

Problema: Come gestire lo switch di finestre all'interno del gioco. Implementare un sistema dove la Gui deve solo avere i metodi utili per registrare le finestre e gli eventi, senza che però lo faccia direttamente.

Soluzione: La classe Gui principale denominata "GameGui" contiene metodi utili per la coordinazione e riuscita dello switch tra le finestre principali del gioco. La Gui prende la configurazione di gioco in ingresso e crea le istanze dei pannelli contenenti i bottoni del menu e delle varie altre View presenti nel gioco (praticamente è la View generale). Si è deciso di mantenere separata ogni singola classe rappresentante un pannello di un particolare menu nel gioco (menu di inizio, menu di info gioco, ecc.). La classe GameGui oltre ad inizializzare le istanze, ha diversi metodi che si occupano di settare gli eventi ai bottoni delle classi specifiche. Questi metodi vengono chiamati dal Controller generale di

gioco, il quale registra determinate finestre da cambiare (le varie View del gioco) o azioni da compiere per gran parte dei bottoni nel menu (la Gui offre i metodi al Controller).

Figura 2.25 implementazione GameGui UML.

Pannelli del menu separati

Oltre alle finestre principali come la mappa o la battaglia, nel gioco sono presenti pannelli di menu o di informazioni sul funzionamento di alcune dinamiche.

Le classi che contengono questi bottoni sono:

- SouthPanel: Pannello che si trova in basso ad ogni finestra una volta iniziato il gioco. Contiene pulsanti come “quit”, “save”, “music”, “menu”.

- InfoMenuPanel: Pannello che mostra le informazioni base di gioco nella schermata iniziale. Cliccando info nel menu che si apre, è possibile vedere un tutorial sul gioco.

- NamePlayerImpl: Pannello dove il player può inserire il suo nome. Una volta iniziato il gioco da capo, viene chiesto il nome del giocatore.

- GameMenuImpl: Pannello del menu appena si apre il gioco. Qui è possibile caricare un salvataggio, iniziare una partita, uscire o leggere il tutorial.

Queste classi vengono, come detto precedentemente, tutte usate dalla GameGui, che grazie al Controller, inserisce gli eventi dentro i bottoni. Per i bottoni richiesti sono stati creati due Enum che il Controller usa per identificarli. In questo modo si possono aggiungere bottoni facilmente e anche metodi per l'assegnamento di eventi, creando una semplice comunicazione tra View e Controller.

Le classi che non hanno un Enum dentro è perché possono essere tranquillamente settate dalla Gui senza che intervenga il Controller, visto che i bottoni si occupano solo di mostrare un pannello (InfoMenuPanel, NamePlayerImpl).

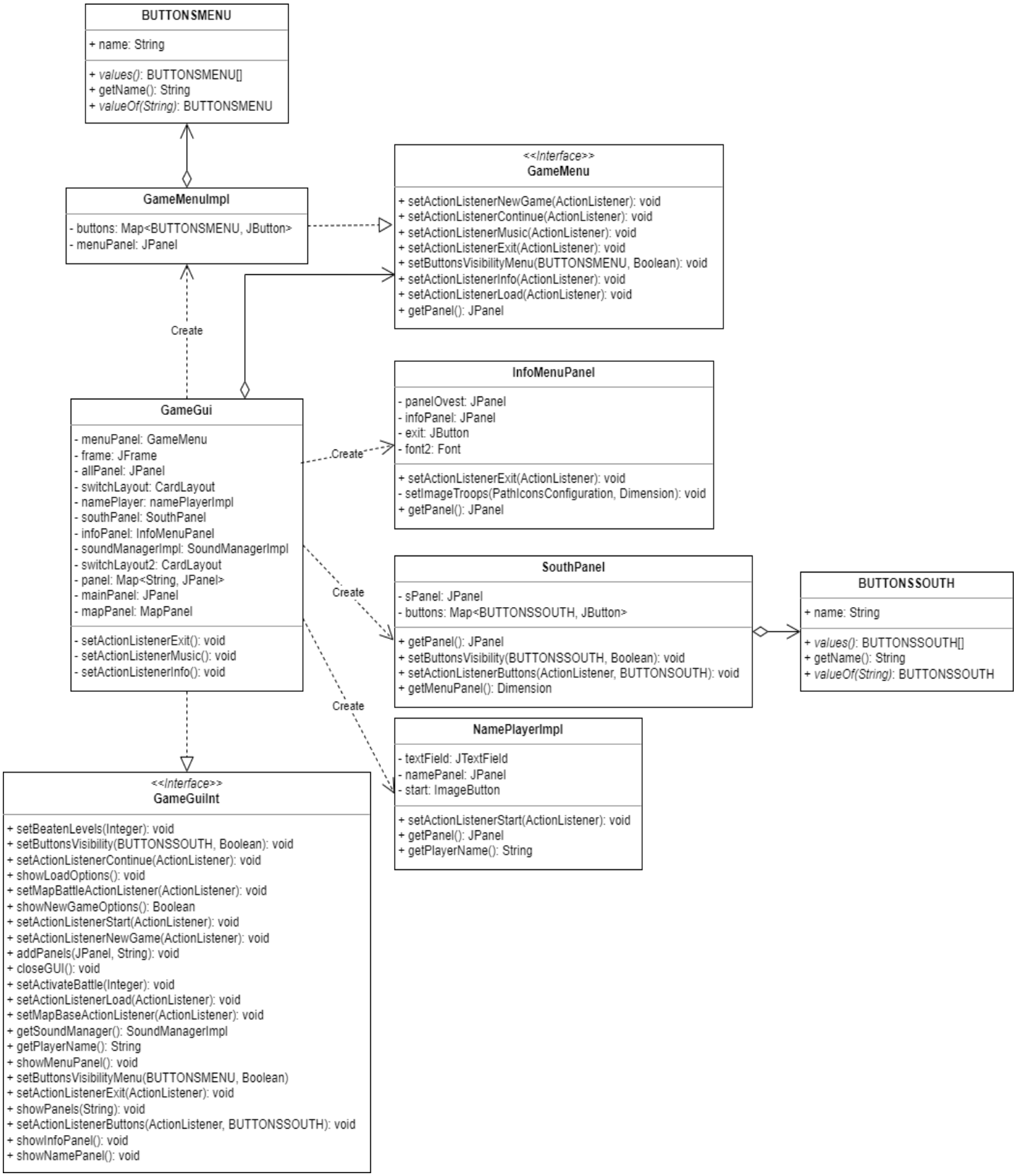

Figura 2.26 architettura completa della GameGui.

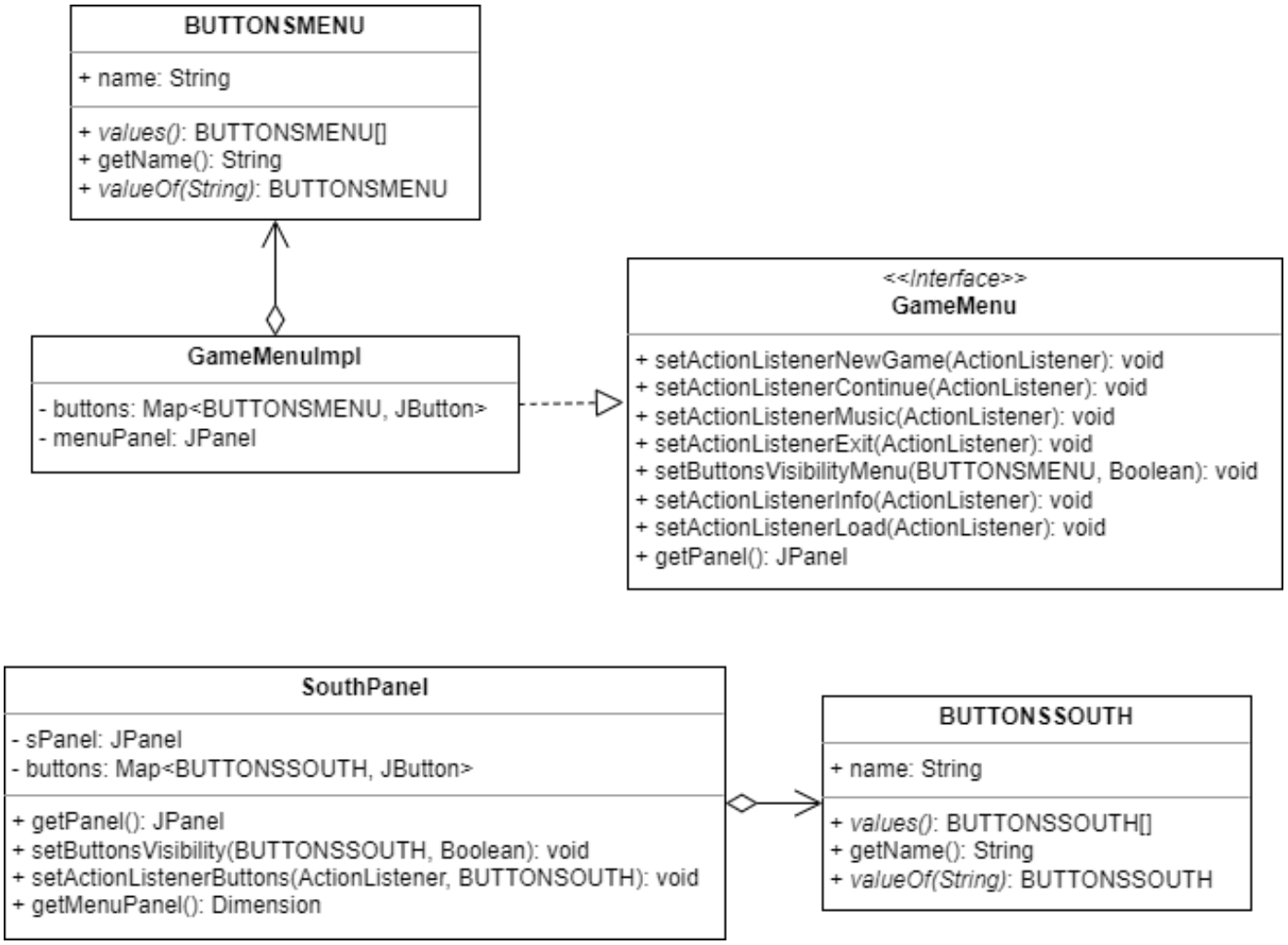

Figura 2.27 Rappresentazione degli enum nelle classi.

Controller

Problema: Durante lo sviluppo del controller, si era indecisi se usare un pattern Observer per la registrazione di funzioni, oppure se semplicemente scrivere dei metodi nel Controller che chiamassero direttamente sia il Model che la View.

Soluzione: È stata adottata la seconda modalità, in quanto più semplice, rapida e intuibile. Si è cercato lo stesso di mantenere una certa modularità, infatti le aggiunte di funzioni nel modello MVC della battaglia, non vanno a intaccare le altre. In caso di modifiche, basterà chiamare le giuste funzioni sia nella View che nel Model. La maggior parte dei metodi nel Controller vengono chiamati tra di loro quando premuti determinati pulsanti (pass, spin, ecc.).

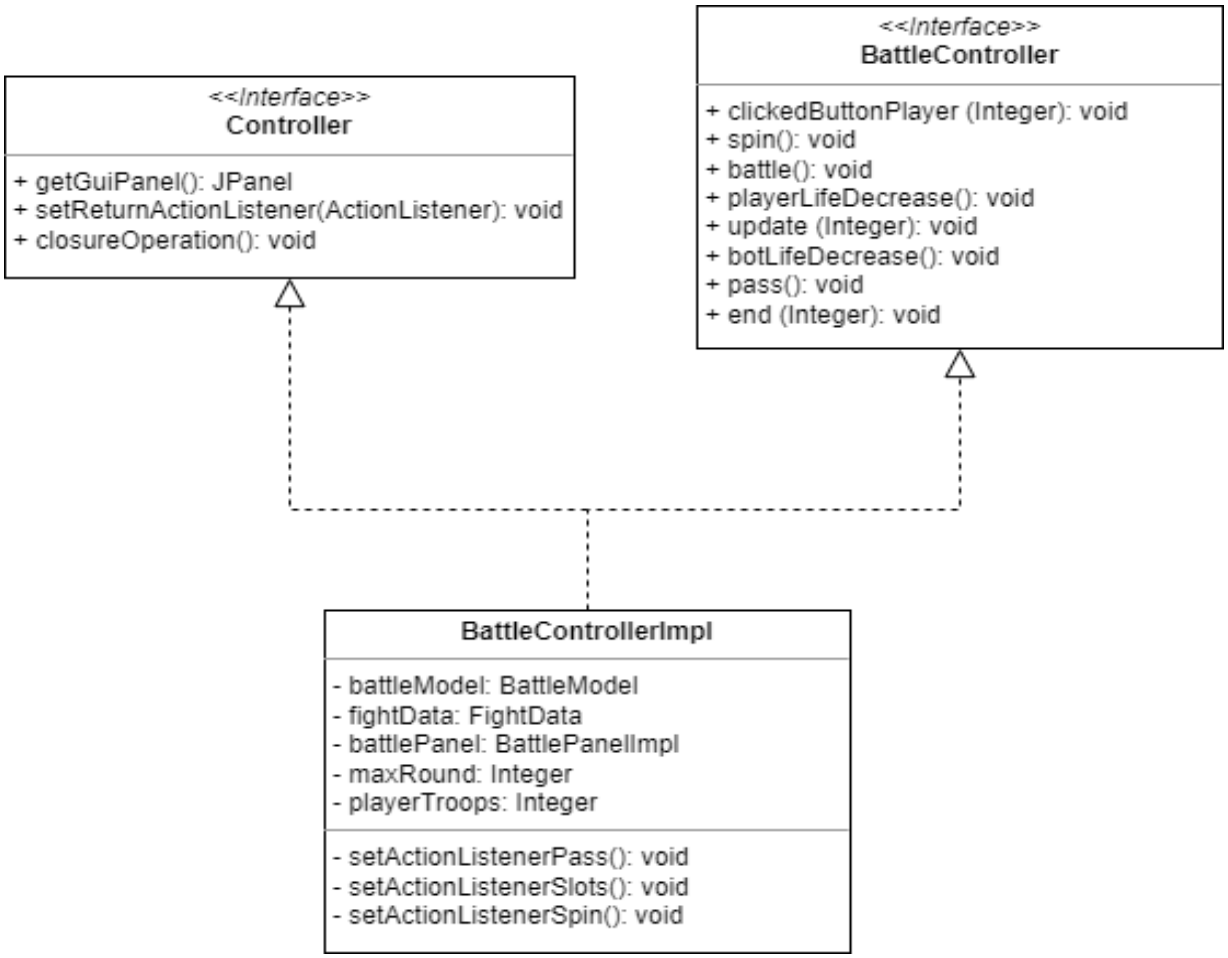

Figura 2.28 Rappresentazione BattleController.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

- EntityDataTest: Controlla la correttezza di ciò che avviene nelle funzioni principali delle entità di gioco.
- BattleModelTest: Si occupa di testare le funzioni del Model principali che utilizzano le entità.

- BaseModelImplTest: Performa i test su ogni singola funzionalità del BaseModel controllando se i dati vengono gestiti correttamente anche in caso di errore.
- ExceptionsTest: Testa se il messaggio di errore viene costruito in maniera corretta in alcune eccezioni.
- ThreadManagerImplTest: Testa se il tempismo e gli esiti delle operazioni del thread sono corretti.
- BaseControllerTest: Testa come i dati vengono incapsulati all'interno del controller.

3.2 Metodologia di lavoro

Inizialmente il nostro gruppo ha elaborato degli schemi UML per dividerci il lavoro a vicenda in modo all'incirca equo. Ci siamo suddivisi il lavoro in maniera tale da permettere che ognuno facesse sia la propria parte di Model-Controller, sia la propria parte di View. Separati in due gruppi, una metà si è occupata della base e delle sue funzionalità, e l'altra della battaglia.

3.2.1 Lorenzo La Farciola

Mi sono occupato in autonomia di:

- Creazione architetturale del Model "BattleModelImpl". Implementazione della classe per le entità della battaglia "EntityData". Creazione della classe "FightData" per la semplificazione nell'uso e creazione delle entità. Creazione di una classe di appoggio "CellsImpl" che si occupa di fornire le informazioni base su cosa è presente in ogni posizione del campo.
- Creazione e implementazione del funzionamento della classe BattleControllerImpl per l'unione di View e Model.
- Separatamente dalla battaglia, creazione della GameGui generale per l'interazione del gioco e di tutti pannelli presenti contenenti bottoni di menu, musica, salvataggio, info e uscita utili al giocatore per una migliore esperienza ("InfoMenuPanel", "NamePlayerImpl", "GameMenuImpl", "SouthPanel").
- Infine, creazione di classi di appoggio per l'estensione di JButton, JTextArea e JPanel, utili alla modifica di immagini in background e altri attributi.

3.2.2 Marco Francolini

Mi sono occupato in autonomia di:

- Progettazione struttura interna model e controller della parte della base (BaseModel e BaseController).
- Implementazione model e controller della parte di costruzione della base nel gioco.
- Implementazione ulility grafiche della classe GraphicUtils.
- Implementazione della view della mappa di selezione tra base e battaglia.
- Implementazione della classe Resource con tutte le sue utlility.

- Implementazione configurazioni utilizzate dai moduli sopreaelencati.
- Inserimento di Pair nel progetto.

3.2.3 Abdoulaye Kane

Mi sono occupato in autonomia di:

- Implementazione della view della città
- Implementazione delle classi di utilità per la view della città (TroopPopup e Cityconfiguration)
- Generazione dei messaggi di avviso all'utente
- Caricamento delle texture per le strutture inserite nella classe pathIconsConfiguration

3.3.4 Silvi Sinani

→ Implementazione del controller principale e gestione dell'inizializzazione dell'applicazione
  - package it.unibo.controller
    - GameController
    - LoadConfiguration

→ Implementazione del modello principale del gioco e gestione del salvataggio.
  - package it.unibo.model
    - GameModel

→ Implementazione delle classi di configurazione del gioco
  - package it.unibo.kingodmclash.config
    - PathIconsConfiguration
    - TextBattleConfiguration
    - GameConfiguration

→ Implementazione della view della battaglia
  - package it.unibo.view.battle.*

→ Implementazione di classi di utilità
  - package it.unibo.view.utilities
    - ImageIconSupplier
    - BattlePanelStyle

→ Gestione della musica.
  - package it.unibo.sound.*

3.3 Note di Sviluppo

3.3.1 Lorenzo La Farciola

- Utilizzo di Stream: https://github.com/RealSilvi/KingdomClash/blob/e62e9db9084bd9ce4b79a40c96ed68f30088e04e/src/main/java/it/unibo/model/battle/entitydata/EntityDataImpl.java#L173C13-L183C32
- Utilizzo di Optional: https://github.com/RealSilvi/KingdomClash/blob/e62e9db9084bd9ce4b79a40c96ed68f30088e04e/src/main/java/it/unibo/model/battle/entitydata/EntityDataImpl.java#L167C9-L167C80
- Utilizzo di Lambda: https://github.com/RealSilvi/KingdomClash/blob/e62e9db9084bd9ce4b79a40c96ed68f30088e04e/src/main/java/it/unibo/model/battle/BattleModelImpl.java#L69C9-L70C34
- Utilizzo di Java Swing: https://github.com/RealSilvi/KingdomClash/blob/e62e9db9084bd9ce4b79a40c96ed68f30088e04e/src/main/java/it/unibo/view/menu/InfoMenuPanel.java#L62C9-L71C49

3.3.2 Marco Francolini

- Utilizzo di Stream, accoppiato all'utilizzo di Optional e Lambda.
- Implementazione della classe Pair fornita durante le esercitazioni.

3.3.3 Abdoulaye Kane

- Stream: https://github.com/RealSilvi/KingdomClash/blob/402ea1d8a1edbcfaa14d6f1ea7d0edcb56ab2b21/src/main/java/it/unibo/view/city/utilities/TroopPopupPanel.java#L70
- Optional: https://github.com/RealSilvi/KingdomClash/blob/11d2011e7ab776b72ff83848d112eed45aff43c3/src/main/java/it/unibo/view/city/panels/impl/BarPanelImpl.java#L252C2-L252C2
- Java Swing: https://github.com/RealSilvi/KingdomClash/blob/583149c5203a19403a5d515a5ef039ed362b3dee/src/main/java/it/unibo/view/city/CityPanelImpl.java#L64

3.3.4 Silvi Sinani

- Utilizzo della libreria GSON:
  https://github.com/RealSilvi/KingdomClash/blob/e6d1b7c81439b8755c84f24a6b5d27283aa97bf6/src/main/java/it/unibo/controller/LoadConfiguration.java#L21
- Utilizzo della libreria Java Sound Sampled:
  https://github.com/RealSilvi/KingdomClash/blob/e6d1b7c81439b8755c84f24a6b5d27283aa97bf6/src/main/java/it/unibo/controller/sound/SoundManagerImpl.java#L18
- Utilizzo di Stream:
  https://github.com/RealSilvi/KingdomClash/blob/e6d1b7c81439b8755c84f24a6b5d27283aa97bf6/src/main/java/it/unibo/view/battle/panels/entities/impl/LifePanelImpl.java#L50
- Utilizzo di Lambda:
  https://github.com/RealSilvi/KingdomClash/blob/e6d1b7c81439b8755c84f24a6b5d27283aa97bf6/src/main/java/it/unibo/view/battle/panels/impl/InfoPanelImpl.java#L47

Capitolo 4
Commenti Finali

4.1 Autovalutazione e lavori futuri

Lorenzo La Farciola

Mi ritengo soddisfatto del lavoro svolto in questo progetto di gruppo. Non avevo mai partecipato o affrontato difficoltà del genere, e proprio per questo ho cercato di mettermi in gioco al massimo. Ho appreso nuove cose e ho imparato a lavorare in un sistema di suddivisione di lavori per lo sviluppo di un progetto di questo calibro. Sicuramente non nego che è stato inizialmente difficile regolare le parti e riuscire perfettamente a comunicare tra di noi, ma successivamente capendoci siamo riusciti ad arrivare all'obiettivo. In generale è stato complicato riuscire a gestirsi bene il tempo, sia per differenze di puntualità tra i membri nel completare determinate parti, sia per scelte personali. Ho usato molto del mio tempo concentrandomi sul Model, trascurando magari alcuni piccole parti. Avrei potuto sicuramente porre più attenzione riguardo l'ordine e l'efficacia di alcuni pezzi del codice ma, nonostante questo, direi che sono riuscito comunque a cavarmela, anche se ovviamente avrei potuto fare di più.

Marco Francolini

Nonostante le parecchie difficoltà incontrate, questa esperienza mi ha consentito di comprendere diversi aspetti che compongono un lavoro di un piccolo team di sviluppo. La suddivisione di un

problema complesso in sotto problemi, e successivamente in piccoli task da dividere in sotto parti del gruppo di lavoro, è una delle abilità sviluppate che ritengo più importanti, poiché consente di scalare il carico di un progetto equamente per i membri del gruppo. Grazie a questa esperienza sono riuscito a comprendere l'importanza di pattern di programmazione ben definite nei progetti, permettendo di comprendere più facilmente il meccanismo di alcune strutture viste in parti di codice anche al di fuori del progetto.

Abdoulaye Kane

Per il progetto avrei potuto dare un contributo maggiore. È stato il primo lavoro di gruppo di questo genere e nonostante avessi delle lacune ho cercato di mettermi in gioco con questo progetto. Ho ritenuto abbastanza complesso il lavoro tra la divisione dei compiti a trovare una struttura corretta che potesse funzionare con le parti dei miei compagni di gruppo. Ci sono state alcune difficoltà di comunicazione iniziale e di tempistiche che hanno portato a un ritardo nella consegna ma alla fine siamo riusciti a svolgere bene il lavoro. Ho imparato molto dal progetto sia dalla parte di progettazione a quella di scrittura del codice. Avrei potuto imparare molto altro ma ho appreso l'importanza degli strumenti che può fornire Java e per progetti futuri potrebbero rendere più efficiente il codice.

Silvi Sinani

L'esperienza mi è stata molto utile. In primis mi ha fatto capire che oltre l'importanza delle hard-skills che una persona può possedere, altrettanto importanti sono le soft-skills. Soprattutto quando si lavora in gruppo. Infatti all'interno del team purtroppo abbiamo riscontrato delle difficoltà, soprattutto per quanto riguarda il rispetto delle deadline interne. Nonostante tutto però mi ritengo soddisfatto del lavoro svolto. Credo di essere cresciuto molto poiché questo progetto mi ha insegnato ad avere un punto di vista più ampio della nostra professione. Per quanto riguarda il linguaggio Java e la filosofia di OOP mi sento di avere costruito delle basi molto forti ma purtroppo ho visto che nel mondo del lavoro si richiedono molte altre skills di contorno. L'unico aspetto amaro è quello di non essere riuscito a padroneggiare l'uso dei pattern; infatti, mi rendo conto che il mio problem-solving non "ragiona a pattern" nonostante ne conosca diversi.

4.2 Difficoltà incontrate e commenti per i docenti

Marco Francolini
Complessivamente mi ritengo molto soddisfatto di questa esperienza e del supporto fornito sui forum del corso; tuttavia, avrei una critica sulle specifiche del progetto. A parer mio, l'utilizzo delle deadline fisse, nonostante ne comprendo lo scopo, non credo raggiungano il loro obbiettivo pienamente, poiché per la natura del progetto, non essendo nel contesto di un lavoro vero e proprio, ma di un elaborato, dove studenti possono anche essere fuori sede, rischiando di fare saltare anche un concetto di affidabilità di disponibilità dei vari componenti del gruppo, viene difficile potersi organizzare in orari e gruppi di lavoro sempre costanti.

Silvi Sinani
Si potrebbe anticipare nel percorso accademico concetti non direttamente collegati con il corso come i DVCS oppure i sistemi delle automazioni delle build. Risulta difficile comprenderli a pieno durante il corso poiché contemporaneamente avviene l'introduzione di altri molti concetti specifici del corso di importanza maggiore.

Bisogna dare la possibilità di implementare il progetto in singolo. Vi sono studenti pendolari e lavoratori. Sono cosciente che in singolo il progetto perda obbiettivi d'insegnamento come team-working e project-management che sono importantissimi ma bisogna tenere conto che non sono il focus del corso.

> Una possibile idea potrebbe essere:
>
> Dare la possibilità allo studente di scegliere se implementare il progetto in singolo oppure in gruppo quando si hanno delle valide circostanze.
>
> Nel caso di implementazioni in singolo attribuire un progetto allo studente e invertire il peso di voto tra esame di laboratorio e progetto. Oppure attribuire più peso ai laboratori consegnati durante il corso.

Appendice A
Guida Utente

Visuale Città interna

La città è il luogo dove il giocatore può decidere strategicamente la gestione di edifici e potenziamenti. Si è liberi di decidere che edifici costruire, considerando un massimo di 4 edifici posizionati contemporaneamente sul campo. Le costruzioni diverse sono 3:

- Municipio: costruzione di base già costruita, permette la produzione di legno e grano.

Figura 2.29 Municipio livello 1.

- Casa: costruzione che permette la produzione di grano.

Figura 2.30 Casa livello 1.

- Segheria: costruzione che permette la produzione di legno.

Figura 2.31 Segheria livello 1.

Figura 2.32 Risorse.

Il giocatore può costruire gli edifici che vuole in base alle sue esigenze, perché grazie a questi si è in grado di potenziare le truppe. In caso la costruzione sia stata messa per sbaglio o in generale vuole essere rimossa, chi gioca può tranquillamente distruggerla.

Figura 2.33 Attiva modalità demolizione delle strutture.

Gli edifici costruiti, dopo un po' di tempo che sono presenti sul campo, si potenziano. Potenziandosi producono maggiormente e cambia la texture leggermente.

Figura 2.34 Potenziamenti truppe.

Visuale Dall'alto Città - Accampamento

La visuale dall'alto è la prima cosa che vede il giocatore quando carica una partita o ne inizia una nuova. Qui si possono fare fondamentalmente solo due cose, ovvero, scegliere di tornare sulla città per eventuali upgrade o modifiche, oppure attaccare l'accampamento avversario.

Le basi del bot sono tre e il giocatore è obbligato a battere quella che viene mostrata. Inizialmente è agibile solo la prima base e le altre due sono nascoste. Una volta battuta la prima, chi gioca vedrà l'accampamento distrutto e quello nuovo sbloccato e disponibile per essere attaccato.

Figura 2.35 Visuale dall'alto con prima base disponibile e le altre nascoste.

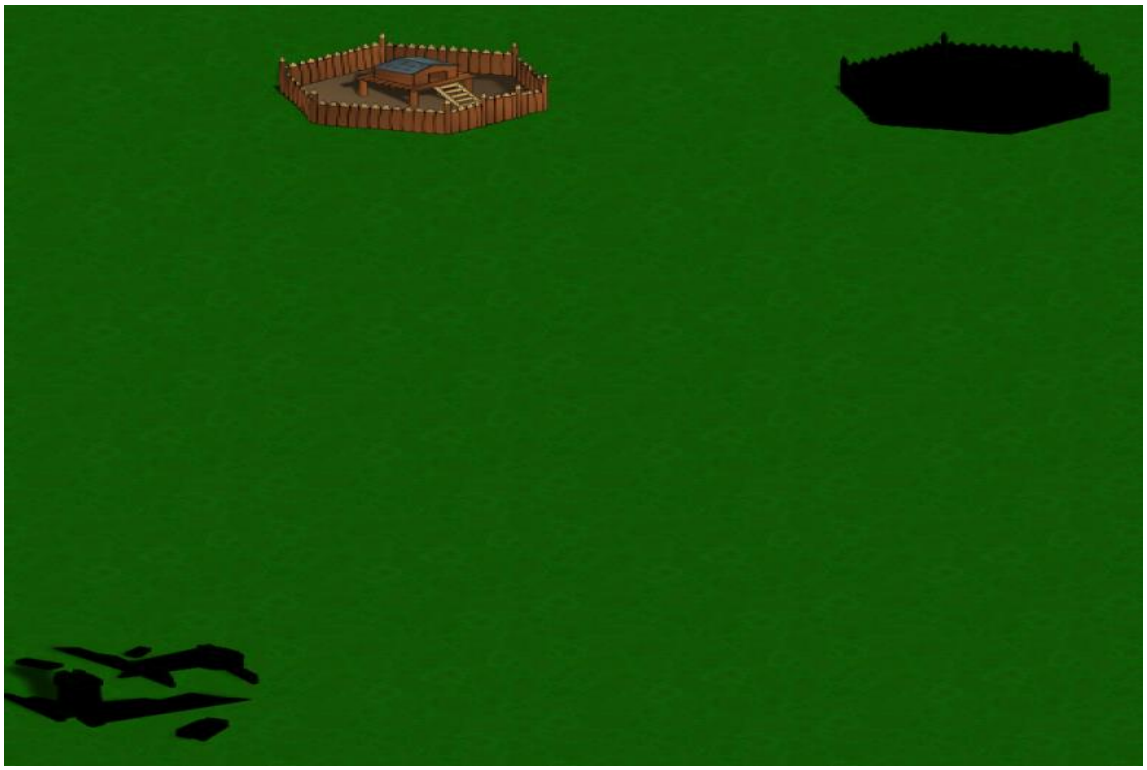

Figura 2.36 Base bot distrutta, seconda base sbloccata.

Ad ogni accampamento il livello di truppe del bot sale di uno, costringendo quindi il giocatore a potenziarsi prima di combattere o ad uno svantaggio in caso voglia affrontare il bot con truppe più deboli. Essendo il massimo di basi avversarie uguale a tre, anche il livello massimo di ogni truppa è tre. Le truppe nel gioco sono un totale di otto (tutte diverse), divise in due gruppi:

Truppe di attacco

- Spada
- Ascia
- Mazza
- Martello

Truppe di difesa

- Scudo per difesa dalla spada
- Scudo per difesa dall'ascia
- Scudo per difesa dalla mazza
- Scudo per difesa dal martello

Ogni truppa richiede determinate risorse per essere potenziata e può far danno solo se l'avversario non mette il suo opposto in campo. In caso la truppa di attacco è di un livello maggiore della truppa opposta di difesa, sfonderà lo scudo e farà danno a prescindere. Gli scudi, quindi, difendono i colpi solo se sono di livello pari o superiore alla truppa opposta.

Campo di battaglia Player – Bot

Premuto sull'accampamento, inizia la battaglia. Per ogni accampamento la battaglia è sempre uguale: viene girata la ruota, appaiono truppe casuali, entrambi scelgono quali schierare.

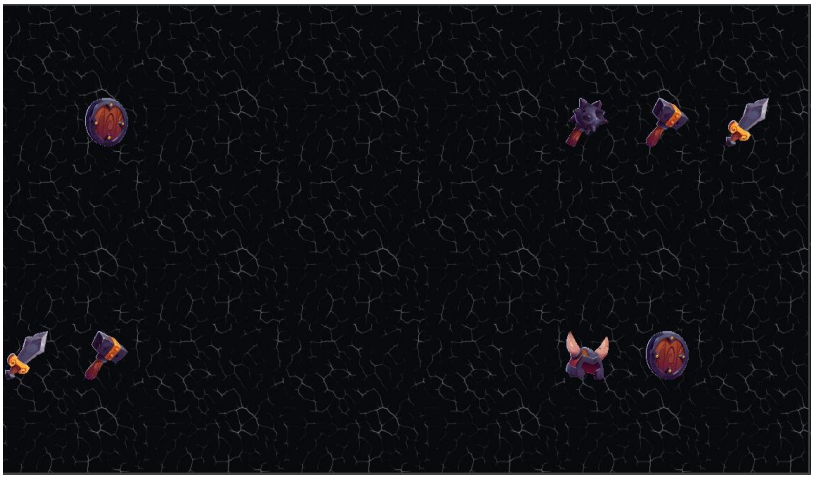

Figura 2.37 Campo battaglia

Le truppe in mano sono cinque, e una volta selezionata una truppa e passato il turno, quella stessa truppa non può essere ritirata dal campo.

Figura 2.38 Truppe in mano

I turni in totale sono tre, quindi dopo aver passato tre volte, spetta l'ultimo turno del bot e la battaglia inizia. Il giocatore può vedere di turno in turno le possibili sorti del combattimento, perché allo schieramento di una truppa il campo viene riordinato posizionando le equivalenti e opposte truppe, l'una di fronte all'altra. Nel campo, visivamente, le truppe di attacco del giocatore sono a sinistra mentre le difese sono a destra. Arrivati all'ultimo turno, se entrambi non hanno posizionato tutte le truppe, quelle rimaste deselezionate vengono obbligatoriamente buttate in campo. Una volta che quindi il giocatore passa il suo ultimo turno, il bot fa le sue mosse, e infine si potrà vedere per qualche secondo le truppe sparire affrontandosi a vicenda e i cuori diminuire. Le vite di entrambi si trovano sulla destra della griglia, mentre sulla sinistra è presente una tabella di confronti che mostra se le truppe del giocatore possono vincere contro quelle avversarie oppure no.

Figure 2.39 e 2.40 Tabelle dei confronti (a sinistra), delle vite e dei bottoni (a destra).

L'avversario può fare le stesse cose del giocatore (quindi usare lo spin e il pass). In caso il giocatore selezioni tutte e cinque le truppe al primo turno, il bot sarà comunque libero di fare i suoi tre turni.

La battaglia finisce quando uno dei due perde tutte le vite e, come citato prima, le vite vengono perse ogni tre turni, durante il combattimento.