



”The100DaysWar”
Relazione per il progetto di “Programmazione
ad Oggetti” A.A 2023/24

Balzani Riccardo
Bartolini Riccardo
Fabbri Gianmarco
Francalanci Filippo

10 Gennaio 2025

Indice

1 Analisi	3
1.1 Descrizione e requisiti	3
1.1.1 Requisiti funzionali	3
1.1.2 Requisiti non funzionali	4
1.2 Modello del Dominio	5
2 Design	6
2.1 Architettura	6
2.2 Design dettagliato	7
2.2.1 Balzani Riccardo	7
2.2.2 Bartolini Riccardo	17
2.2.3 Fabbri Gianmarco	25
2.2.4 Francalanci Filippo	34
3 Sviluppo	40
3.1 Testing automatizzato	40
3.1.1 Balzani Riccardo	40
3.1.2 Bartolini Riccardo	41
3.1.3 Fabbri Gianmarco	42
3.1.4 Francalanci Filippo	42
3.2 Note di sviluppo	43
3.2.1 Balzani Riccardo	43
3.2.2 Bartolini Riccardo	44
3.2.3 Fabbri Gianmarco	45
3.2.4 Francalanci Filippo	45
4 Commenti finali	47
4.1 Autovalutazione e lavori futuri	47
4.1.1 Bartolini Riccardo	49
4.1.2 Fabbri Gianmarco	49
4.1.3 Francalanci Filippo	50

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il software The100DaysWar è un gioco strategico a turni che simula uno scontro militare tra due giocatori su una mappa bidimensionale. Il gioco si svolge nell'arco di 100 giorni virtuali con scambio di turni durante i quali i giocatori devono gestire risorse, costruire torri difensive e pianificare strategie per ottenere il controllo delle caselle della mappa. Ogni giocatore inizia con uno spawn point e un budget iniziale di 250 monete. Durante il proprio turno, i giocatori possono reclutare soldati per l'attacco, costruire torri difensive, potenziare le proprie unità e pianificare movimenti sulla mappa. Il combattimento tra soldati viene risolto attraverso un sistema di dadi, dove il numero di dadi lanciati dipende dal livello dell'unità coinvolta, mentre tra soldati e torri e viceversa viene arrecato un danno proporzionale al livello. La mappa di gioco è composta da diverse tipologie di celle che influenzano le possibili azioni dei giocatori: celle edificabili per la costruzione di torri, celle non edificabili che rappresentano ostacoli, e celle speciali che possono fornire bonus di risorse quando occupate. Il gioco supporta il combattimento giocatore contro computer (PvE), implementando un sistema semplice per il bot che valuta e sceglie le azioni più appropriate in base al proprio stato attuale.

1.1.1 Requisiti funzionali

- Gestione dei turni alternati tra i giocatori, con un limite di tempo per turno
- Sistema di combattimento basato su dadi, dove il numero di dadi è proporzionale al livello dell'unità

- Gestione delle risorse con accumulo progressivo di monete ad ogni turno
- Sistema di costruzione e potenziamento delle unità:
 - Soldati mobili con capacità offensive (livello massimo 3)
 - Torri difensive stazionarie (livello massimo 3)
- Sistema di movimento delle unità sulla mappa con gestione delle collisioni e dei combattimenti
- Bot in grado di:
 - Valutare la propria situazione corrente.
 - Scegliere l'azione più appropriata tra acquisto, potenziamento e movimento delle unità
 - Adattare la strategia in base alle risorse
- Sistema di vittoria basato su due condizioni:
 - Conquista immediata dello spawn point nemico
 - Controllo della maggioranza delle celle alla fine dei 100 turni

1.1.2 Requisiti non funzionali

- Corretto funzionamento in ciascuno dei 3 sistemi operativi principali: Linux, Windows e MacOs.
- Interfaccia utente intuitiva per la gestione delle azioni di gioco
- Performance ottimizzate per garantire una rapida esecuzione dei turni
- Architettura modulare per facilitare l'estensione del gioco con nuove funzionalità
- Codice mantenibile e ben documentato per facilitare future modifiche e aggiornamenti

1.2 Modello del Dominio

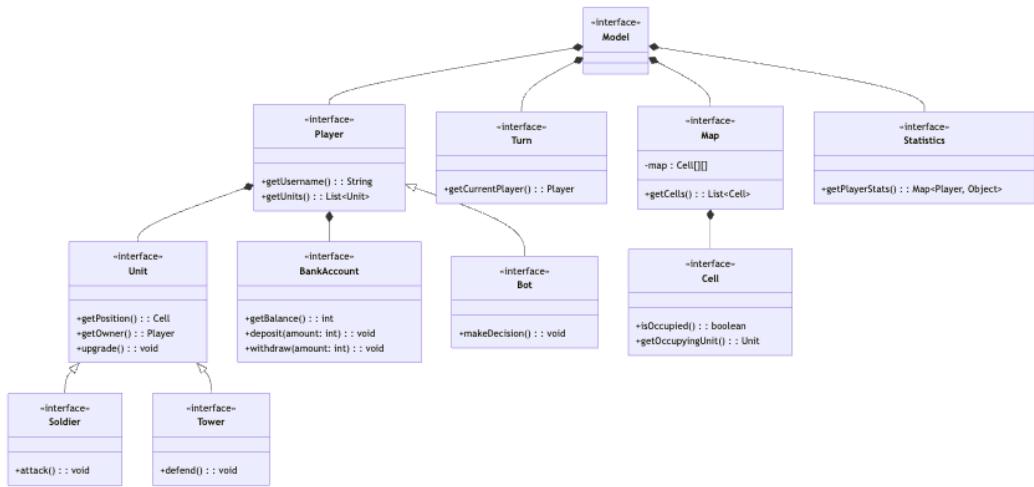


Figura 1.1: Dominio applicativo di The100DaysWar

Il gioco si caratterizza di entità Player. Ciascun player possiede delle Unit, divise tra Soldier e Tower, e un BankAccount per la gestione delle risorse. Bot sarà un'estensione di Player. La Map di gioco si compone di Cell. Saranno presenti dei Turn in cui il Player può svolgere le proprie azioni e delle Statistics visualizzabili.

Capitolo 2

Design

2.1 Architettura

Per la realizzazione del software si ‘e scelto, in fase di progettazione, di utilizzare il pattern architettonale MVC (Model-View-Controller). A livello implementativo, il punto cardine del Model ‘e l’interfaccia Model. Questa ha il compito di richiamare, effettuare i vari controlli sui metodi delle classi e di fornire un insieme di azioni del gioco. Per il Controller, il componente principale dell’architettura, ‘e l’interfaccia MainController, la quale rende possibile l’utilizzo dei controller minori, ciascuno dei quali si occupa di gestire uno specifico elemento della View. Ha anche il compito di inizializzare un’istanza del GameModel e di garantire il corretto avvio, termine, e salvataggio del gioco. A livello di View, il punto di entrata è StartMenuView che fornisce all’utente una label per inserire il proprio username e tramite il bottone “start” viene delegato al MainController la creazione e visualizzazione del reale elemento principale: GameView. Questo, istanzia i pannelli e i bottoni, che hanno il compito di rendere visibile lo stato attuale del model e di poterci interagire tramite i controller corrispondenti. Tramite il corretto utilizzo del pattern MVC, viene agevolata una possibile sostituzione della componente grafica, senza dover modificare le parti di Model e Controller, non essendoci dipendenze con esse.

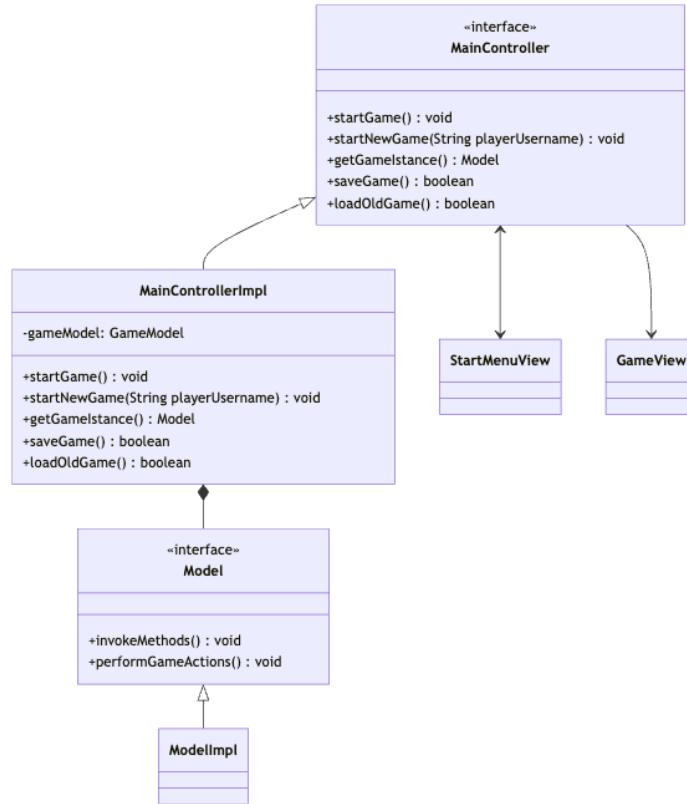


Figura 2.1: Implementazione del Pattern MVC

2.2 Design dettagliato

2.2.1 Balzani Riccardo

Torri

Problema

Nel gioco *The100DaysWar*, le torri sono unità statiche che infliggono danni ai soldati nemici all'interno del loro raggio d'azione. Devono essere:

- Costruibili
- Potenziabili
- Eliminabili

L’obiettivo è quello di creare un sistema semplice per la gestione delle torri, con un’interfaccia intuitiva per il giocatore e una struttura facilmente espandibile per supportare nuovi tipi di torri in futuro.

Soluzione

La progettazione si basa su un’interfaccia generale `Tower` (estensione di `Unit` della quale parleremo in seguito), che definisce i comportamenti comuni a tutti i tipi di torre, come infliggere danni o perdere vita. Questo approccio garantisce flessibilità e polimorfismo.

L’unica categoria di torri implementata nel gioco è quella delle torri normali, rappresentata grazie alla classe astratta `AbstractNormalTower`, che definisce proprietà standard come posizione, danno e resistenza. Questa classe è stata concretizzata in due varianti: `BasicTower` e `AdvancedTower`, che differiscono per costo, vita e danno inflitto.

Per semplificare la creazione e garantire espandibilità, è stato utilizzato il *Factory Pattern*. La classe `TowerFactoryImpl` centralizza la creazione delle torri, utilizzando un sistema che associa ogni tipo di torre al suo costruttore.

I tipi di torre implementati concretamente sono invece contenuti, ed associati al proprio prezzo, all’interno dell’enum `TowerType`.

Le caratteristiche delle torri non gestite in questa struttura, riguardanti eliminabilità e combattimento, sono state implementate, per scelte di progettazione, all’interno del `TurnManager` (descritto più avanti).

Vantaggi

- **Espandibilità:** Nuove categorie di torri (anche con abilità totalmente diverse dalle normal tower) possono essere aggiunte con facilità.
- **Semplicità:** L’aggiunta di nuove torri normali richiede solo l’implementazione di una nuova classe concreta.
- **Manutenzione:** La creazione è centralizzata in `TowerFactoryImpl`, garantendo encapsulamento e riducendo i rischi di errori.
- **Organizzazione:** Il codice modulare facilita la gestione di nuove funzionalità.

Svantaggi

- Con poche tipologie di torri, la struttura gerarchica e il *Factory Pattern* possono risultare eccessivamente complessi.

- L'uso del *Factory Pattern* può risultare ridondante in un contesto con poche torri concrete.

Grazie a questa progettazione, il sistema delle torri può evolversi facilmente, offrendo maggiore varietà e nuove meccaniche per arricchire l'esperienza di gioco.

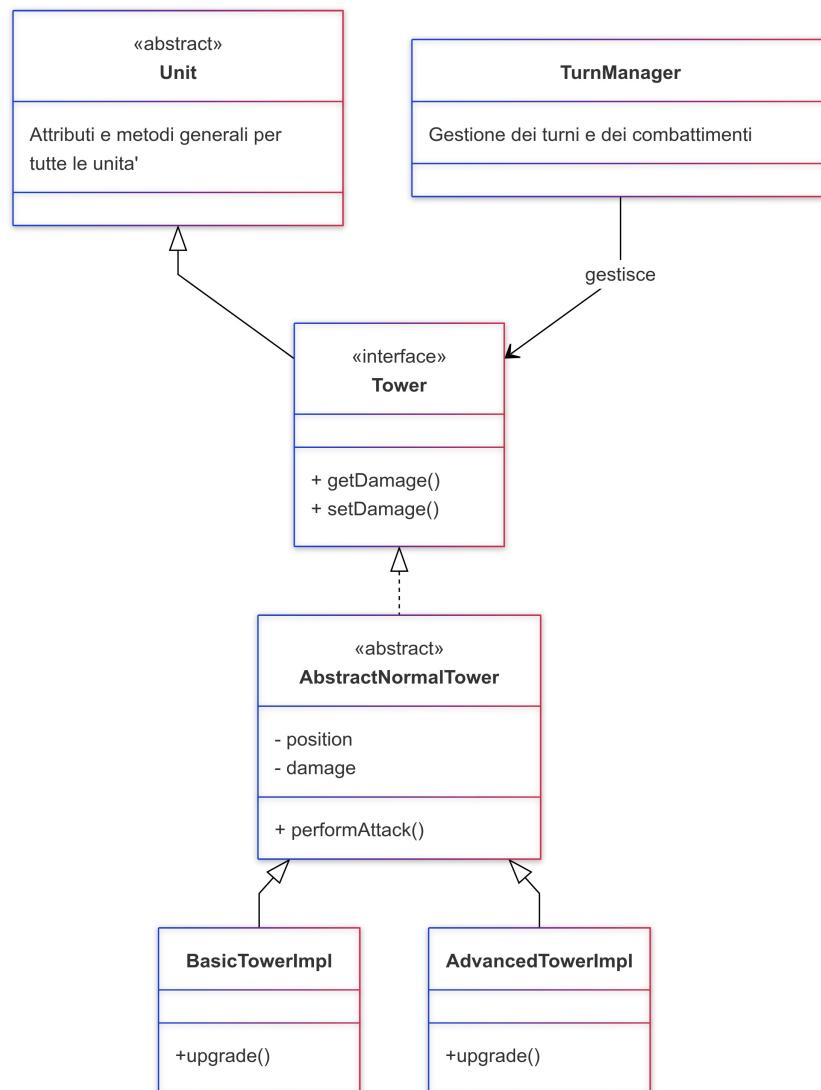


Figura 2.2: Schema UML della struttura di Tower

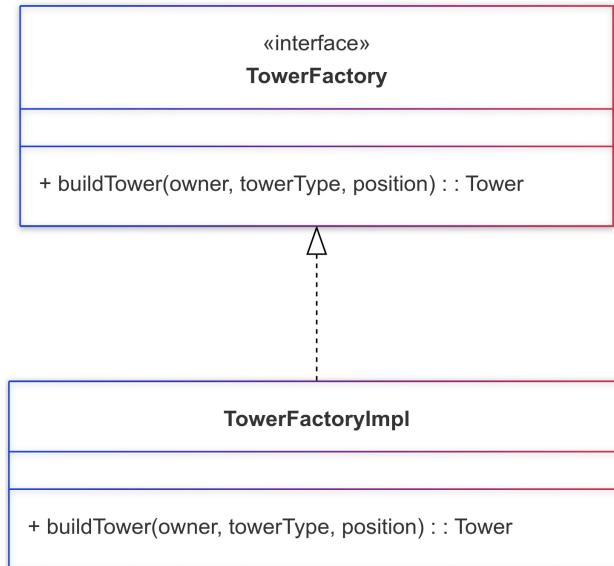


Figura 2.3: Shcema UML TowerFactory

Salvataggio dei Dati

Problema

In *The100DaysWar*, i giocatori devono poter salvare la partita corrente e riprenderla successivamente senza perdere alcuna informazione. È stato necessario sviluppare un sistema che raccogliesse e salvasse:

- Turno corrente
- Unità possedute
- Conto bancario
- Mappa

con tutte le relative caratteristiche.

Soluzione

Per gestire il salvataggio dei dati, sono state implementate le classi `GameDataImpl` e `GameSaverImpl`.

- **GameDataImpl**: Racchiude tutte le informazioni della partita corrente in un'unica struttura, facilitando l'accesso e la gestione dei dati necessari per il salvataggio.
- **GameSaverImpl**: Si occupa di salvare i dati di un'istanza di `GameDataImpl` sul file `saved_game.ser` tramite serializzazione. Il salvataggio avviene attraverso il metodo `saveGame()`, che serializza i dati utilizzando un `ObjectOutputStream`. Gli eventuali errori durante la scrittura vengono gestiti successivamente dal `MainController` e notificati tramite un sistema di logger.

Inoltre, `GameSaverImpl` permette di specificare un percorso personalizzato per il file di salvataggio, garantendo maggiore flessibilità. Se non specificato, il file viene salvato nel percorso predefinito.

Al livello del controller, inoltre, per garantire una gestione efficiente, il salvataggio delle partite viene eseguito in modo asincrono tramite un `ExecutorService` che esegue il task in un thread separato. Il metodo si avvale di un timeout per evitare eccessive attese e restituisce un risultato booleano per indicare il successo o il fallimento della scrittura dei salvataggi. Il risultato dell'operazione di salvataggio sarà notificato all'utente finale.

Vantaggi

- **Manutenibilità**: `GameDataImpl` centralizza tutti i dati necessari per il salvataggio, rendendo semplice aggiungere nuovi dati in futuro.
- **Flessibilità**: `GameSaverImpl` consente di salvare il file in un percorso definito dallo sviluppatore.
- **Robustezza**: La gestione delle eccezioni e di attese eccessive è solida e ben integrata nel sistema.

Svantaggi

- Tutte le classi presenti in `GameDataImpl`, e quelle utilizzate al loro interno, devono essere serializzabili, aumentando la complessità del sistema.
- L'utilizzo di un `ExecutorService`, soprattutto lavorando con piccoli salvataggi, può aggiungere complessità al codice.
- Il sistema attuale non supporta il salvataggio di partite multiple.

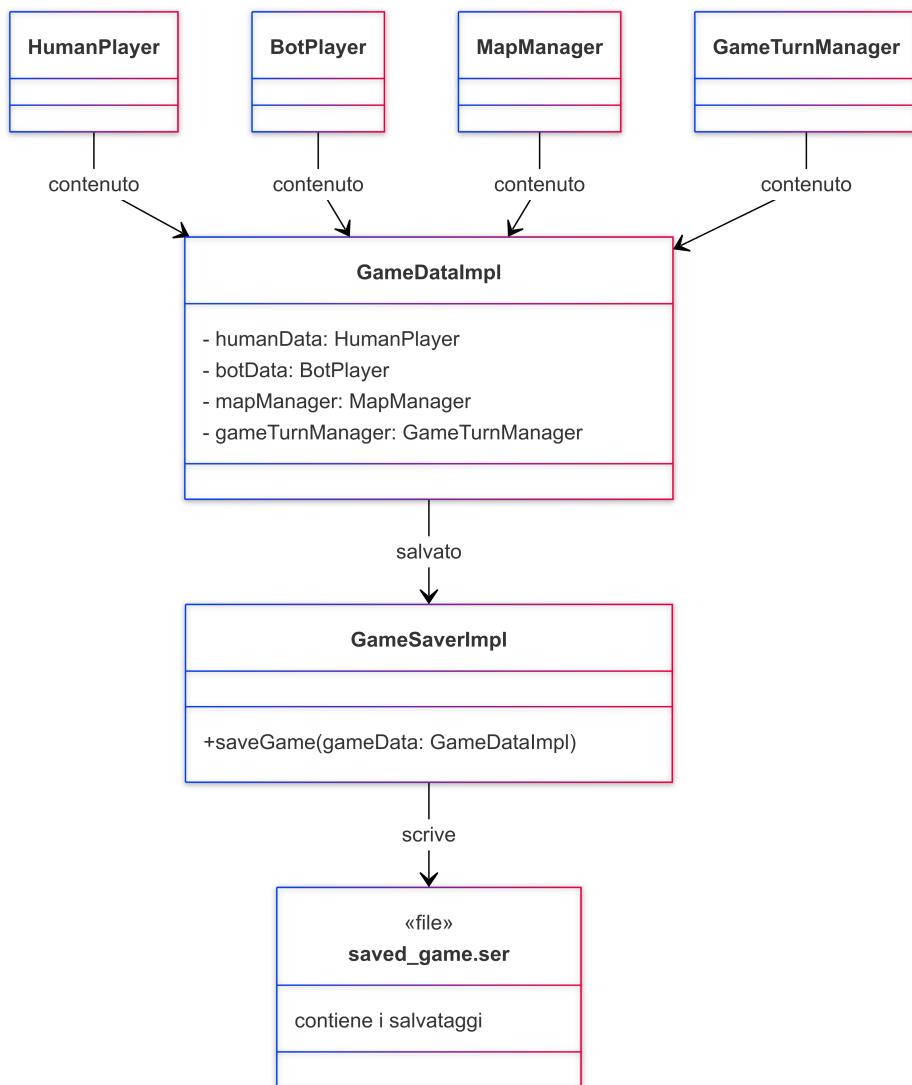


Figura 2.4: Shcema UML salvataggio dei dati sul file saved_game.ser

Caricamento dei Salvataggi in una Nuova Partita

Problema

Il sistema di salvataggio richiede un meccanismo per caricare i dati salvati, consentendo ai giocatori di riprendere i progressi accumulati senza perdita

di informazioni. In *The100DaysWar*, deve essere possibile selezionare dalla prima schermata l'opzione per riprendere l'ultima partita salvata.

Soluzione

Per gestire il caricamento, vengono utilizzate le classi `GameLoaderImpl` e `GameDataImpl` (descritta nella sezione precedente).

Soluzione

Per gestire il caricamento, vengono utilizzate le classi `GameLoaderImpl` e `GameDataImpl` (descritta nella sezione precedente).

- `GameLoaderImpl` legge il file `game_saved.ser` dal percorso specificato o, in assenza di specifiche, dal percorso predefinito. Il metodo principale, `loadGame()`, utilizza un `ObjectInputStream` per deserializzare i dati e caricarli in un'istanza dedicata di `GameDataImpl`.

L'istanza inizializzata è quindi utilizzata per rigenerare la partita salvata tramite un costruttore dedicato all'interno del model. In caso di errori durante la lettura del file, le eccezioni vengono registrate tramite un apposito sistema di logger, e il metodo `loadGame()` restituisce un `Optional.empty()`, gestito in modo appropriato dal controller.

Al livello del controller, inoltre, viene utilizzato il medesimo meccanismo utilizzato nel salvataggio (sezione sopraffante) che, come già spiegato, tramite `ExecutorService` consente di monitorare il tempo impiegato e migliorare l'efficienza. Anche in questo caso, il risultato finale del caricamento, sarà notificato all'utente finale.

Vantaggi

- **Affidabilità:** La gestione delle eccezioni tramite logger garantisce che eventuali errori durante il caricamento siano identificati e notificati senza interrompere il funzionamento dell'applicazione.
- **Manutenibilità:** La separazione tra la lettura dei dati (`GameLoaderImpl`) e la rigenerazione della partita (`GameDataImpl`) facilita l'estensione e la manutenzione del sistema.
- **Flessibilità:** La possibilità di specificare un percorso personalizzato per il file offre maggiore controllo e configurabilità per futuri sviluppatori.

Svantaggi

Gli svantaggi sono sostanzialmente gli stessi che ricorrono nel salvataggio.

- **Dipendenza dalla serializzazione:** Tutte le classi contenute in `GameDataImpl` e le loro dipendenze devono essere serializzabili, aumentando la complessità complessiva del codice.
- L'utilizzo di un `ExecutorService` può complicare il codice.
- **Caricamento limitato:** Il sistema attuale supporta il caricamento di una sola partita alla volta. Non è possibile gestire salvataggi multipli senza modificare la struttura.

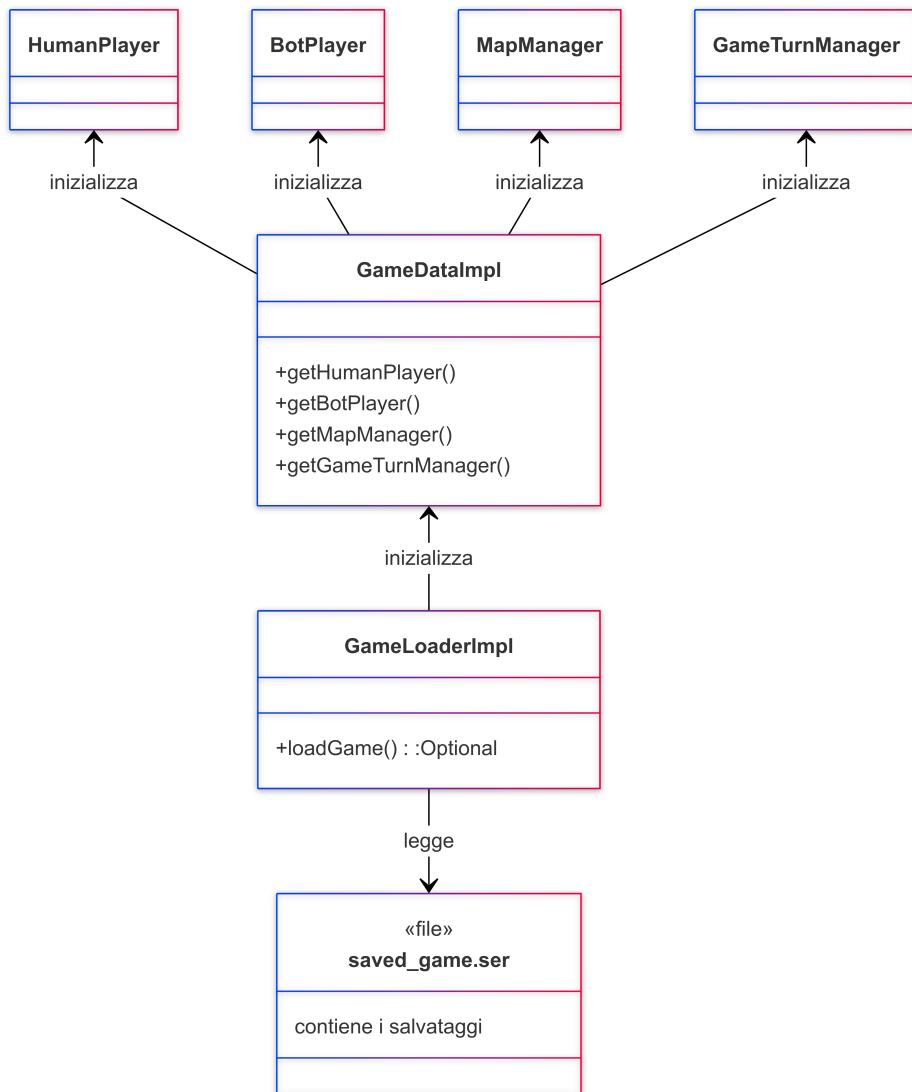


Figura 2.5: Schema UML lettura dei salvataggi dal file saved_game.ser

Dado

Problema

In *The100DaysWar*, le unità combattono simulando il lancio di dadi a sei facce. Il lancio deve garantire che ogni faccia abbia la stessa probabilità di uscire.

Soluzione

È stata implementata la classe `DiceImpl` per simulare il lancio di un dado. Il metodo principale, `roll()`, restituisce un numero casuale tra 1 e il numero di facce specificato nella costante statica `ACES` (sei in questo caso). L'uso di un generatore di numeri casuali garantisce l'equiprobabilità.

Per favorire l'estendibilità futura, ho introdotto l'interfaccia `Dice`, che permette di aggiungere varianti di dadi con più facce, proprietà speciali o probabilità personalizzate senza modificare il codice esistente.

Ogni volta che il dado viene lanciato, l'utente visualizza una piccola animazione che mostra questa azione.

Punti favorevoli

- Il numero di facce del dado è facilmente modificabile grazie alla costante `ACES`.
- La struttura progettuale favorisce l'aggiunta di nuove tipologie di dadi.

Punti sfavorevoli

- Attualmente è implementata solo la tipologia standard di dado a sei facce.
- L'animazione del dado non termina con la faccia con la quale esso atterra.

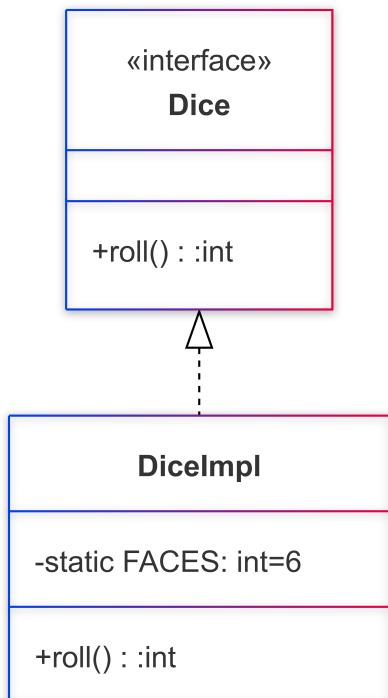


Figura 2.6: Shcema UML struttura Dice

2.2.2 Bartolini Riccardo

Diverse tipologie di celle

Problema

Per la nostra versione del gioco è necessario creare diversi tipi di celle che andranno poi a comporre la mappa in modo manutenibile e flessibile, ciò può comportare della duplicazione del codice e difficoltà nella manutenzione di esso.

Soluzione

Come soluzione viene modellata una sola classe **Cell** che modella al suo interno i vari aspetti compresi per le altre celle con caratteristiche particolari (**Spawn**, **Obstacle**) mentre per l'implementazione della **BonusCell** è stata implementata una classe che estende la classe **Cell** ma ha alcuni attributi e metodi dedicati alla gestione del bonus.

PRO:

- Semplicità: avendo una sola classe Cell che modella gli aspetti comuni a tutte le celle.
- Manutenibilità: con tutte le funzionalità principali centralizzate in una sola classe, le modifiche al comportamento comune delle celle possono essere apportate in un unico punto.
- Estendibilità: la soluzione permette l'aggiunta di nuovi tipi di celle in futuro estendendo Cell.

CONTRO:

- Responsabilità eccessiva: la classe Cell potrebbe diventare troppo complessa nel tempo.
- Rischio di Overhead: l'inclusione di metodi e attributi inutilizzati per alcuni tipi di celle.

Soluzione alternativa

Invece di estendere direttamente la classe Cell per creare celle con caratteristiche specifiche, si potrebbe utilizzare il Pattern Decorator per aggiungere dinamicamente comportamenti alle celle.

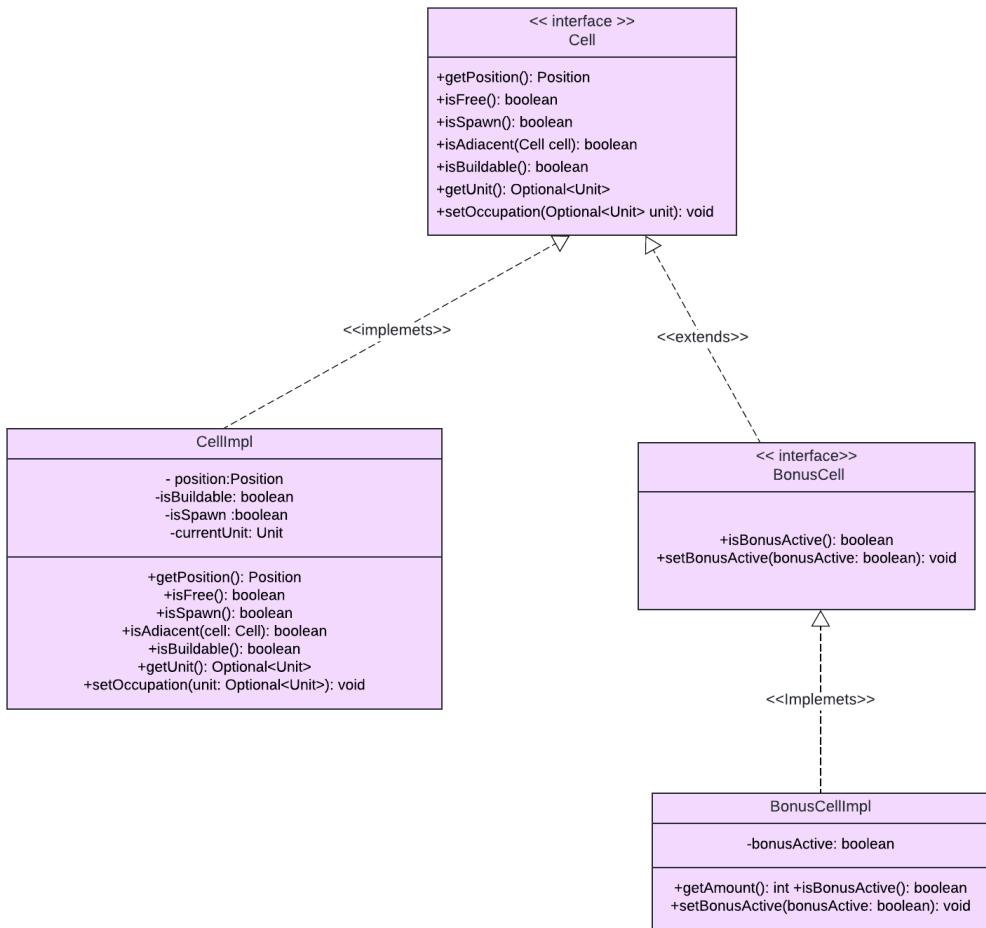


Figura 2.7: Schema UML per i tipi di Cell

Creazione della GameMap

PROBLEMA

Nel gioco è necessario avere una mappa di gioco composta da tutte le celle su cui i giocatori possono svolgere le proprie azioni come posizionare truppe, muovere i soldati, comprare truppe ecc... Tutti i componenti in essa devono essere in perfetta coesione per il corretto funzionamento in relazione con le altre classi che interagiscono con essa.

Soluzione

Come soluzione è stato utilizzato il pattern Builder ovvero il GameMapBuilder, attraverso il quale viene creata la mappa passo per passo attraverso i suoi metodi, ognuno dei quali dedicato ad una tipologia di cella differente per poi alla fine ottenere la mappa completa.

PRO:

- Flessibilità nella creazione: Il pattern Builder consente di costruire la mappa passo dopo passo, permettendo di configurare le celle e gli elementi della mappa in modo specifico.
- Manutenibilità: La logica di creazione è encapsulata all'interno del Builder, mantenendo la classe GameMap più pulita e focalizzata sull'uso della mappa anziché sulla sua creazione .
- Coesione e integrazione: La mappa viene costruita con metodi dedicati, garantendo che tutti i componenti siano perfettamente integrati e funzionino in relazione con le altre classi.
- Estendibilità: È possibile aggiungere nuovi tipi di celle o caratteristiche alla mappa senza modificare significativamente il codice esistente.

CONTRO:

- Complessità iniziale: L'implementazione del pattern Builder può introdurre una complessità iniziale non banale.
- Dipendenza dal Builder: l'uso del Builder introduce una dipendenza esplicita tra la classe che gestisce la mappa (MapManager) e il processo di costruzione.

Soluzione alternativa

In alternativa al pattern Builder, è possibile utilizzare il Pattern Factory, che si focalizza sulla creazione di oggetti complessi attraverso metodi statici o istanze dedicate.

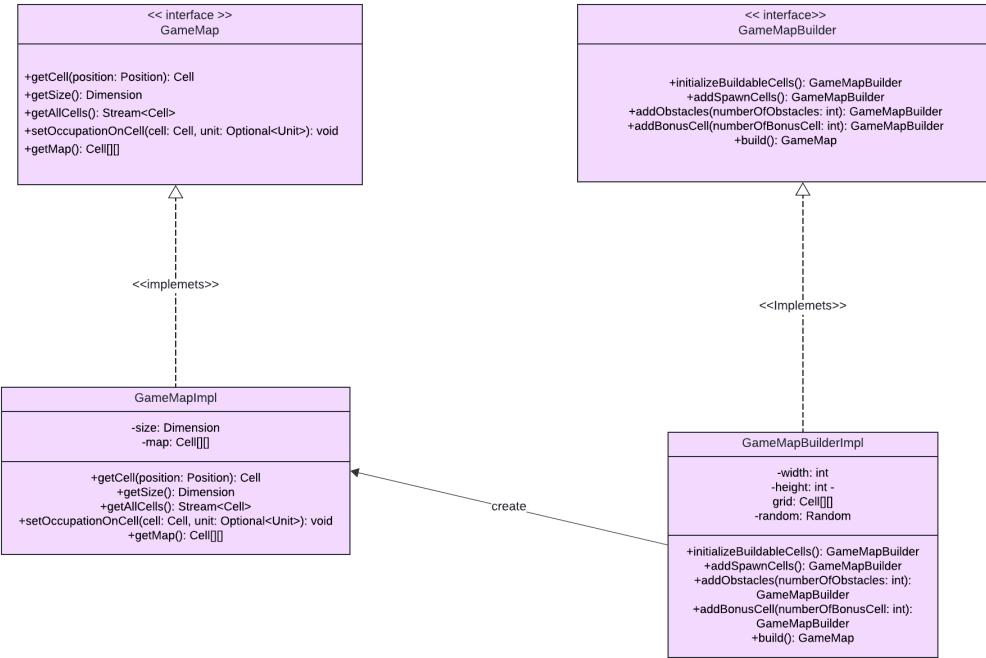


Figura 2.8: Schema UML per i tipi di Cell

Gestione della mappa e degli eventi su essa

Problema

All'interno del gioco è necessario che vengano gestiti tutti gli eventi che si presentano sulla mappa che è lo scenario principale di gioco, per esempio i movimenti sulle celle, posizionamento e combattimenti fra le unità dei due giocatori.

Soluzione

La soluzione proposta consiste nell'utilizzo del MapManager che fa gestore principale del gioco, attraverso i suoi metodi gestisce le varie interazioni fra gli oggetti presenti nella mappa, attraverso esso viene creata la mappa, con l'utilizzo del builder, in modo che possa agire direttamente su essa per effettuare modifiche a seguito di sviluppi della partita, per essere informato delle varie modifiche effettuate dalle altre classi è stato utilizzato il pattern Observer.

PRO:

- Centralizzazione della logica: Il MapManager agisce come gestore centrale di tutti gli eventi legati alla mappa, semplificando la gestione delle interazioni tra gli oggetti.
- Manutenibilità: Grazie al pattern Observer, il MapManager viene notificato automaticamente delle modifiche effettuate dalle altre classi, riducendo le dipendenze dirette tra i componenti e migliorando la modularità.
- Integrazione con il Builder: L'integrazione con il pattern Builder per la creazione della mappa semplifica l'inizializzazione della mappa e garantisce che il MapManager abbia accesso diretto agli oggetti della mappa.
- Estendibilità: È possibile aggiungere nuove tipologie di eventi sulla mappa senza modificare significativamente il codice esistente.

CONTRO:

- Dipendenza elevata: Il MapManager può diventare un collo di bottiglia del sistema, accumulando troppe responsabilità.
- Overhead del pattern Observer: L'uso del pattern Observer introduce un certo overhead, sia in termini di complessità di implementazione che di prestazioni, soprattutto in caso di notifiche frequenti.

Soluzione alternativa

In alternativa, è possibile gestire gli eventi sulla mappa utilizzando il Pattern Command, che separa le richieste (eventi) dall'esecuzione delle azioni.

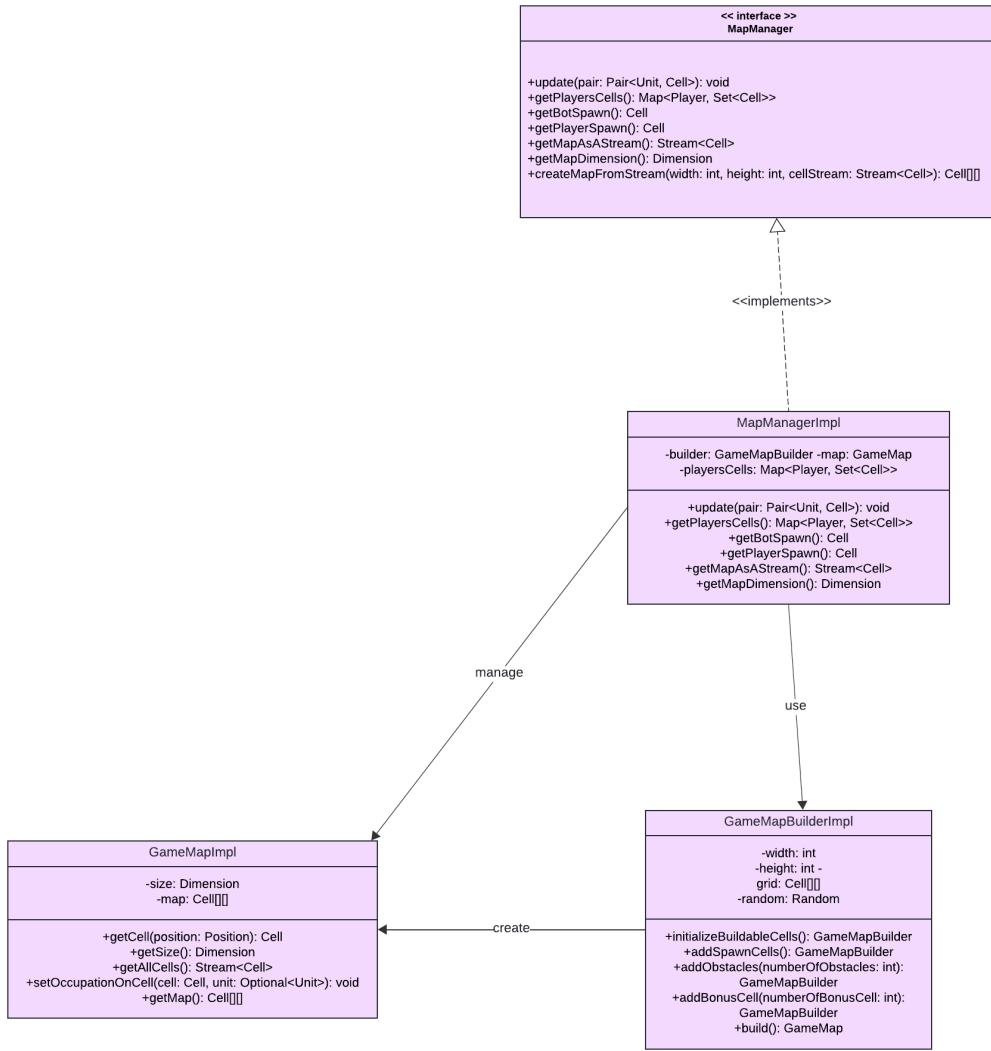


Figura 2.9: Schema UML per i tipi di Cell

Gestione delle statistiche di gioco

Problema

Durante la partita occorre avere le statistiche in tempo reale del gioco per ognuno dei due player in modo da essere a conoscenza di risorse, unità e percentuale di celle in proprio possesso.

Soluzione

Le statistiche sono state implementate utilizzando il pattern MVC, viene istanziata la classe GameStatisticsImpl che implementa l'interfaccia GameStatistics nel Model, attraverso lo StatisticController vengono aggiornati e passati in modo corretto tutti i vari dati alla StatisticsView. Le statistiche vengono trasformate da Map a coppie ordinate in modo decrescente per maggior pulizia e chiarezza.

PRO:

- Separazione delle responsabilità: l'uso del pattern MVC consente di separare chiaramente il calcolo e la gestione delle statistiche (Model) dalla loro rappresentazione grafica (View) e dalla logica di controllo (Controller).
- Aggiornamenti in tempo reale: grazie alla struttura del controller, le statistiche possono essere aggiornate in tempo reale durante il gioco, garantendo che la view mostri sempre i dati più recenti.
- Estendibilità: è semplice aggiungere nuove metriche alle statistiche o nuovi modi per visualizzarle, modificando solo il Model o la View.

CONTRO:

- Dipendenze multiple: L'uso del pattern MVC introduce un certo livello di complessità, poiché è necessario mantenere la sincronizzazione tra il Model, il Controller e la View.
- Overhead di aggiornamento: In presenza con aggiornamenti molto frequenti, l'aggiornamento continuo delle statistiche potrebbe causare un overhead, influendo sulle prestazioni.

Soluzione alternativa

In alternativa al pattern MVC, è possibile utilizzare il Pattern Observer per sincronizzare la View direttamente con il Model.

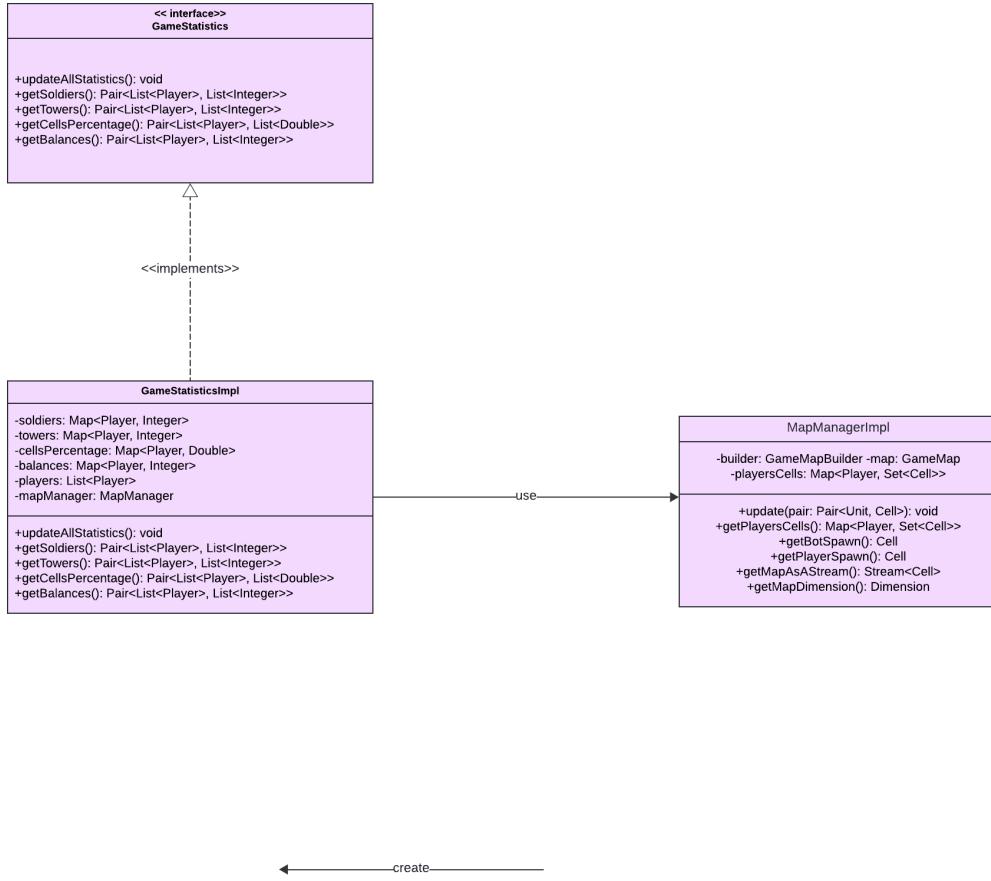


Figura 2.10: Schema UML per i tipi di Cell

2.2.3 Fabbri Gianmarco

Diverse tipologia di unità

Problema

Nel gioco The100DaysWar ci sono diverse tipologie di unità (soldati, torri) con caratteristiche comuni per cui è necessario centralizzare la creazione e mantenere il codice pulito evitando duplicazioni.

Soluzione

La soluzione da me offerta utilizza il **Factory Pattern** e l'uso di una **Abstract Class**. La classe astratta avrà il compito di incapsulare tutte le

caratteristiche comuni tra i soldati e le torri (livello, costi di acquisto / aumento di livello, vita variabile durante il gioco) in un'unica classe: UnitImpl. Il Factory Pattern permette di centralizzare la creazione delle tipologie di unità in un'unica classe: UnitFactory.

Note: per una corretta gestione del posizionamento delle Unit nella mappa e di movimento si è scelto in fase progettuale di utilizzare l'**Observer Pattern** e di rendere ciascuna Unit un Observable in modo da notificare ad un'entità di gestione della mappa (MapManager) la propria modifica di posizione.

PRO

- Centralizzazione della creazione: utilizzando il Factory Pattern, la creazione delle unità è centralizzata in un'unica classe, semplificando la gestione e la manutenzione del codice.
- Riduzione della duplicazione del codice: la classe astratta UnitImpl incapsula le caratteristiche comuni, eliminando la necessità di ripetere lo stesso codice per ogni tipologia di unità.
- Facilità di estensione: l'aggiunta di nuove tipologie di unità diventa più semplice, poiché è sufficiente creare nuove classi che estendono UnitImpl senza modificare il codice esistente della factory.

CONTRO

- L'implementazione del Factory Pattern e di una classe astratta richiede una struttura di classi più articolata, che può risultare eccessiva per progetti di dimensioni ridotte.
- L'utilizzo di una classe astratta impone una certa struttura alle classi derivate, limitando la flessibilità nel caso in cui le caratteristiche delle unità dovessero evolversi in modo non previsto.

Soluzione alternativa

Un'alternativa al Factory Pattern potrebbe essere l'adozione del **Pattern Prototype**. Questo approccio prevede la clonazione di oggetti esistenti per crearne nuovi, piuttosto che istanziarli direttamente tramite una factory.

PRO

- Clonare oggetti esistenti può essere più efficiente rispetto alla creazione di nuove istanze da zero.
- Aggiungere nuove tipologie di unità è più semplice, basta creare nuovi prototipi senza modificare la logica di clonazione esistente.

CONTRO

- Gestione dei prototipi: è necessario gestire correttamente i prototipi registrati per evitare clonazioni indesiderate o inconsistenti.
- Assicurarsi che la clonazione avvenga correttamente se le unità contengono riferimenti a oggetti mutabili, per evitare effetti collaterali.

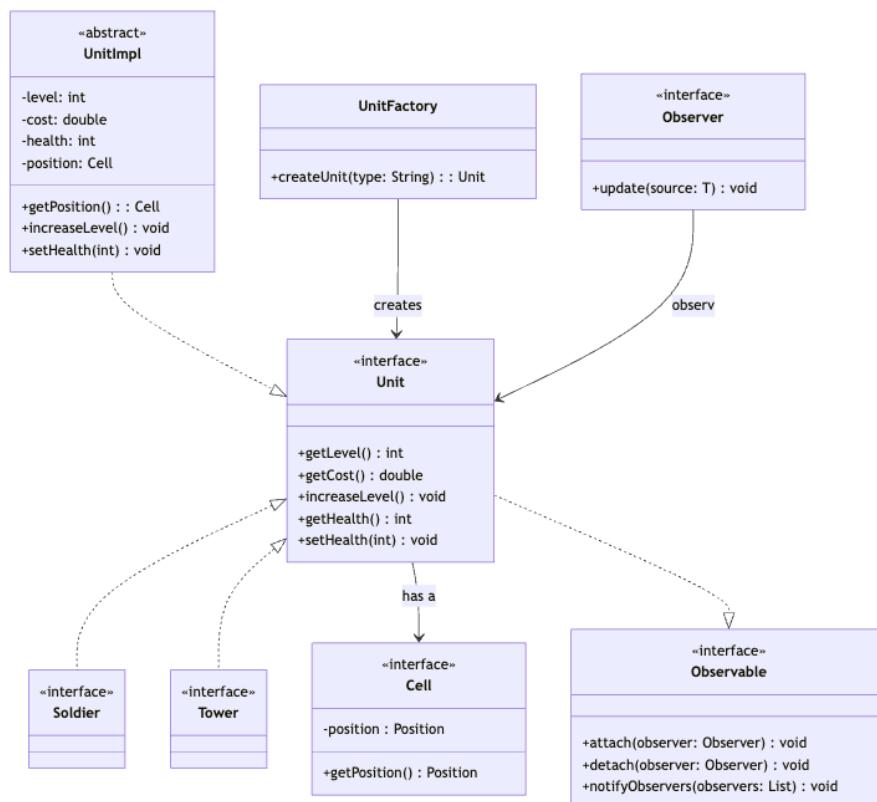


Figura 2.11: Schema UML per Unit e UnitFactory

Diverse tipologie di giocatori

Problema

Nel gioco The100DaysWar ci sono due tipologie di giocatori, umano e bot. L'obiettivo, come per le unità, è di evitare la duplicazione di codice.

Soluzione

Ho scelto di ricorrere di nuovo all'utilizzo di una **Abstract Class**: PlayerImpl. La classe astratta rappresenta una buona soluzione in quanto il bot dispone della struttura del giocatore umano con l'unica differenza che decide autonomamente quale azione svolgere.

Note: come per Unit, abbiamo deciso in fase progettuale di ricorrere all'**Observer Pattern** per gestire un aspetto dinamico del gioco: il guadagno di risorse. Ciò avviene in due maniere, quando un soldato appartenente al giocatore passa sopra una BonusCell e ad ogni incremento di GameDay. A questo proposito si è deciso di realizzare un interfaccia comune: ResourceGenerator che, da come suggerisce il nome, è un'estensione dell'interfaccia Observable dedicata alla notifica a Player per l'aggiunta di risorse.

PRO

- Utilizzando una abstract class le caratteristiche condivise tra giocatori umani e bot sono gestite in un'unica posizione, facilitando la manutenzione del codice.
- Facilità di estensione: è semplice aggiungere nuovi tipi di giocatori creando nuove classi che estendono PlayerImpl senza modificare il codice esistente.

CONTRO

- L'uso di una classe astratta impone una struttura fissa, rendendo difficile adattarsi a cambiamenti non previsti nelle caratteristiche dei giocatori.
- Introdurre una abstract class aggiunge un livello di astrazione che può rendere il sistema più complesso da comprendere.

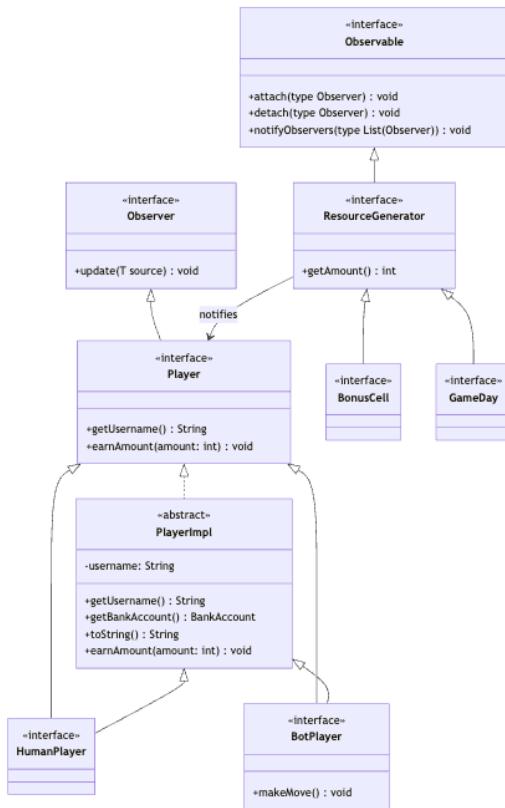


Figura 2.12: Schema UML per i tipi di player

Azioni dei giocatori

Problema

Nel gioco The100DaysWar i giocatori possono effettuare azioni durante lo svolgimento, identiche per le tipologie di giocatori.

Soluzione

In questo caso la soluzione utilizza il **Command Pattern** che permette di encapsulare un comando all'interno di una classe che disporrà di un'unico metodo per eseguirlo. La scelta di creare GenericPlayerCommand è stata presa in modo tale che ogni azione dei giocatori dovrà accettare il contratto di questa interfaccia.

PRO

- Utilizzando il command pattern, ogni azione del giocatore è gestita da una classe separata, facilitando la manutenzione e l'estensione delle funzionalità.
- Il pattern separa il richiedente dell'azione dall'esecutore, aumentando la modularità del sistema.

CONTRO

- Ogni nuova azione richiede la creazione di una nuova classe comando, aumentando la complessità del progetto.
- Implementare il command pattern introduce un livello di astrazione aggiuntivo, che può rendere il sistema più difficile da comprendere all'inizio.

Soluzione alternativa

Un'alternativa al command pattern potrebbe essere l'utilizzo di funzioni di callback o lambda per gestire le azioni dei giocatori. Questo approccio permette di definire le azioni direttamente all'interno delle classi dei giocatori senza crearne nuove separate per ogni azione.

PRO

- Utilizzare callback o lambda riduce il numero di classi necessarie, semplificando la struttura del codice.
- Definire nuove azioni è più rapido e non necessita la creazione di una nuova classe.

CONTRO

- Le azioni definite tramite callback possono essere meno riutilizzabili rispetto alle classi comando.
- Per azioni che richiedono logiche complesse l'uso di callback può rendere il codice meno leggibile e più difficile da mantenere.

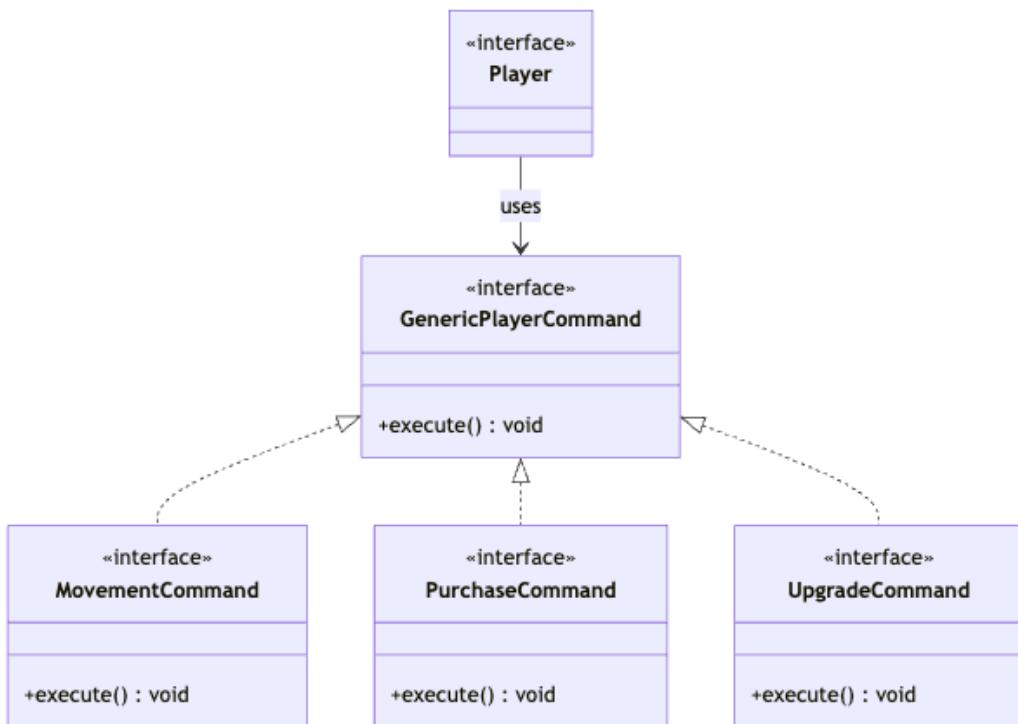


Figura 2.13: Schema UML per le azioni del player

Giocatore automatizzato

Problema

Nel gioco The100DaysWar è necessario che vi sia un giocatore automatizzato per gestire le azioni dei bot, mantenendo il codice pulito e modulare.

Soluzione

La soluzione proposta utilizza il **Pattern Strategy**: BotStrategy. La valutazione di quale mossa eseguire è gestita da DecisionMaker, che utilizza

un'enumerazione delle tipologie di azione: `ActionType`. Ogni `ActionType` ha metodi associati per valutarne l'efficacia e per eseguirla. In questo modo, `DecisionMaker` seleziona l'azione più appropriata basandosi sulle condizioni di gioco correnti e delega l'esecuzione all'azione scelta. Per il type `MOVE_SOLDIER` è stato necessario introdurre: `PathFinder` ossia un'interfaccia che calcola il percorso da una cella partenza ad una cella arrivo della mappa. Nel caso specifico ho deciso di implementarla tramite un algoritmo BFS che avevo già familiarizzato in precedenza. Un ultimo appunto, all'interno dell'enumerazione è presente una classe statica privata: `ActionNotifier` che implementa l'interfaccia `Observable` per gestire l'aspetto di notifica quando una mossa viene eseguita e quindi adattare il bot all'architettura creata. Seppur il bot fornito presenta una sola strategia, ho comunque deciso di mantenere lo strategy pattern per facilitare un'estensione futura del codice.

PRO

- Il pattern strategy permette di cambiare facilmente il comportamento del bot senza modificare la sua struttura di base.
- Viene semplificata l'aggiunta di nuove strategie di gioco creando nuove classi che implementano `BotStrategy`.

CONTRO

- L'implementazione del pattern strategy introduce ulteriori classi e interfacce, aumentando la complessità del progetto.
- Coordinare e gestire diverse strategie può complicare l'architettura del sistema.

Soluzione alternativa

Un'alternativa al pattern strategy potrebbe essere l'utilizzo del **Pattern State**. Questo approccio permette al bot di cambiare il suo comportamento in base allo stato interno, gestendo le diverse modalità di azione attraverso stati distinti.

PRO

- Il pattern state consente al bot di cambiare comportamento dinamicamente in base allo stato corrente del gioco.

- Ogni stato è gestito da una classe separata, rendendo il codice più modulare e facile da mantenere.

CONTRO

- Aumento del numero di classi: implementare diversi stati richiede la creazione di numerose classi.
- Gestire le transizioni tra stati può complicare l'architettura del sistema.

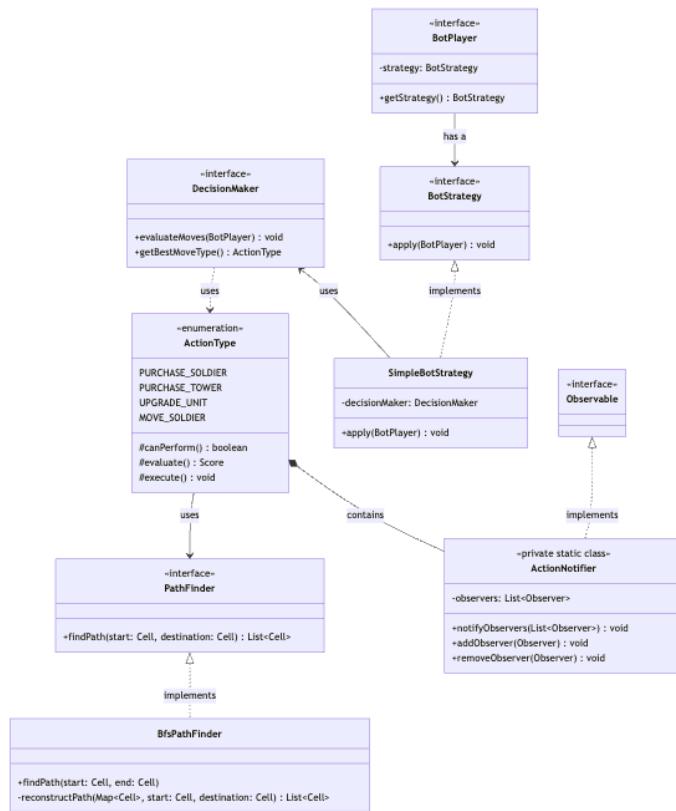


Figura 2.14: Schema UML per la gestione del bot

2.2.4 Francalanci Filippo

Gestione dei Giorni

Problema

Quando il giorno aumenta nel GameDay ogni giocatore dovrebbe ricevere un determinato ammontare di soldi nel loro bank account

Soluzione

Utilizzando l'**Observer Pattern** è possibile notificare i Player. GameDay estende l'interfaccia ResourceGenerator, che a sua volta estende l'interfaccia Observable. Il player estende l'interfaccia Observer. Tramite questo sistema quindi ad ogni incremento di GameDay viene lanciata una notifica alla lista di Player registrata, ogni Player richiama il metodo update dedicato al guadagno di risorse.

PRO:

- Vengono garantite l'efficienza e la velocità di un'azione che viene chiamata spesso.
- Viene separata la logica di aggiornamento dello stato del gioco da quella del giocatore.
- Migliora la leggibilità del codice.

CONTRO:

- Maggiore difficoltà nell'implementazione e di comprensione per uno sviluppatore esterno.

Soluzione Alternativa

Creare una classe centralizzata che ingloba i player e con un metodo di aggiornamento per il bankaccount che viene chiamato quando il giorno viene incrementato.

PRO:

- Più semplice l'implementazione.

CONTRO:

- La logica rimane in un blocco unico senza scorporare le dipendenze e i ruoli.

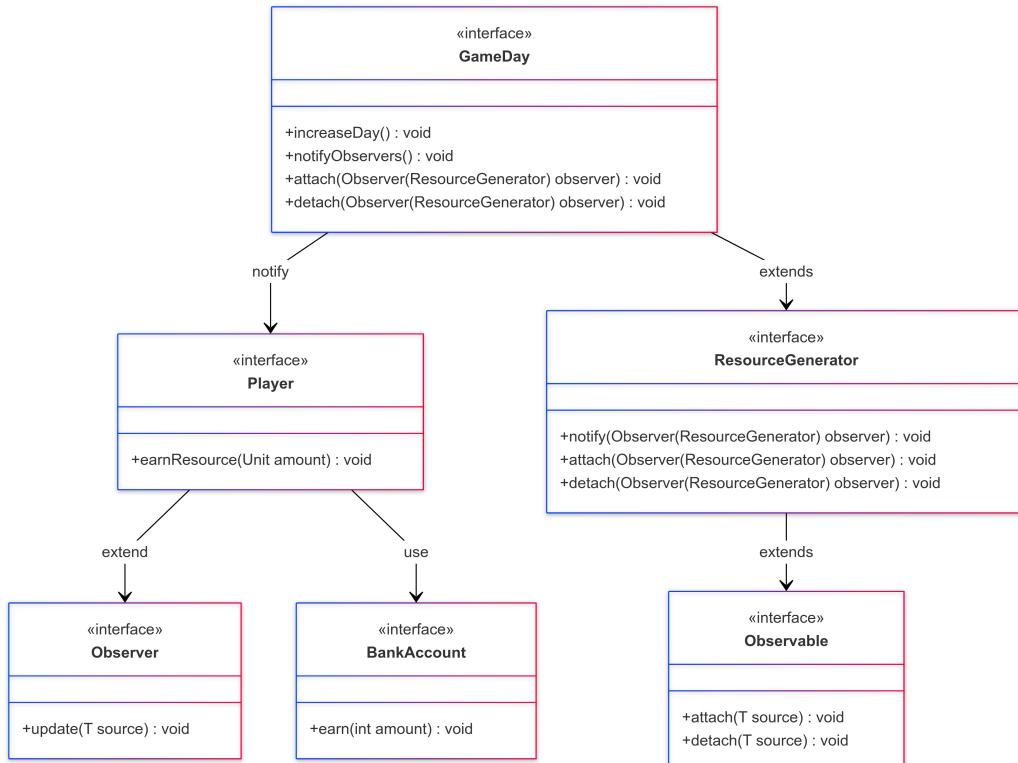


Figura 2.15: Schema UML per la gestione dei giorni

Gestione del cambio dei turni

Problema

Nel gioco c'è la necessità di gestire in un'unica classe le varie funzioni del gioco quali il cambio turno, incremento giorno e gestire gli eventi in gioco.

Soluzione

Ho creato una classe **GameTurnManagerImpl** dove vengono inglobate le classi necessarie per la gestione corretta dell'avanzare dei giorni virtuali e della transizione di turni tra i due Player, inoltre si garantisce l'attacco automatico delle torri ogni proprio turno. Viene utilizzato l'**Observer Pattern**

per l'aggiornamento della lista delle unità di Player quando vengono eliminate e rimosse dal gioco. Viene compreso anche GameDay per l'incremento periodico del giorno a intervalli regolari utilizzando le classi di della libreria java.util: **Timer** e **TimerTask**.

PRO:

- Centralizzata e quindi facile ottenere le risorse necessarie perché basta passare per quella classe
- Automazione utilizzando Timer
- Grazie al pattern Observer, la rimozione delle unità viene separata e aggiornata automaticamente in modo veloce e funzionale

CONTRO:

- Molte dipendeze con le altre classi. Se le altre classi cambiano, è possibile che ci sia da cambiare qualcosa anche nella classe dipendente.
- Se la classe diventa troppo grande può diventare complessa da gestire.

Soluzione Alternativa

Possibilità di utilizzare il Singleton Pattern in quanto c'è solo un gestore dei turni, quindi avere un'istanza unica e garantita.

PRO:

- Risparmio di risorse e quindi di memoria.
- Un'unica istanza globale che tutti possono usufruire.

CONTRO:

- Complica la creazione di test unitari introducendo una dipendenza globale.
- Può essere pericoloso in casi di gestione non accurata.

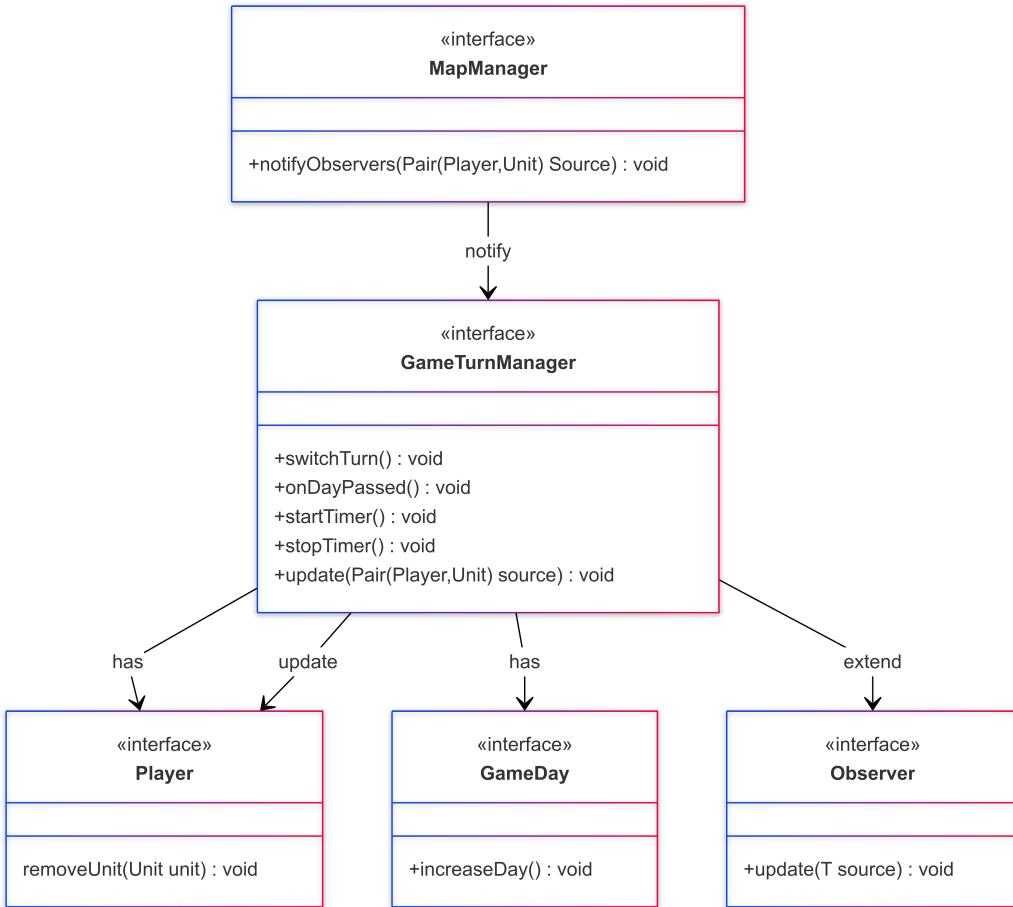


Figura 2.16: Schema UML per le gestione dei turni

Gestione della Battaglia

Problema

Il gioco necessita una gestione delle battaglie tra i tipi di unità. Devono quindi essere gestite le battaglie tra soldati e quelle tra le torri e soldati in ambo i casi di attacco e difesa.

Soluzione

La soluzione da me proposta è quella di usare due pattern, **Factory Pattern** e **Command Pattern**. Ho scelto di utilizzare questa accoppiata per il bisogno di gestire tipi di battaglie differenti, ognuna con una logica propria.

- La BattleFactory ha il compito di creare la battaglia adatta in base alle tipologie di attaccante e difensore rappresentate dall'interfaccia Combatant.
- Il pattern Command realizzato tramite l'interfaccia BattleCommand e successivamente implementata da GenericBattleCommand gioca un ruolo fondamentale in quanto si riesce ad incapsulare in un solo metodo la battaglia sfruttando quanto detto in precedenza.

PRO:

- L'uso del Factory Pattern consente di creare dinamicamente diverse battaglie in base ai tipi di unità coinvolte, il che rende il sistema facilmente estensibile per nuovi tipi di unità o battaglie aggiuntive.
- Il Command Pattern permette di eseguire correttamente il comportamento della battaglia in un singolo comando, quindi garantisce l'incapsulamento.

CONTRO:

- Un design più complesso e difficile da comprendere ad una prima lettura.

Soluzione Alternativa

Una soluzione alternativa poteva essere utilizzare una classe BattleManager. Utilizzando un unico metodo che analizza le caratteristiche degli attaccanti e difensori.

PRO:

- Implementazione e comprensione più immediate.

CONTRO:

- Estensibilità ridotta.

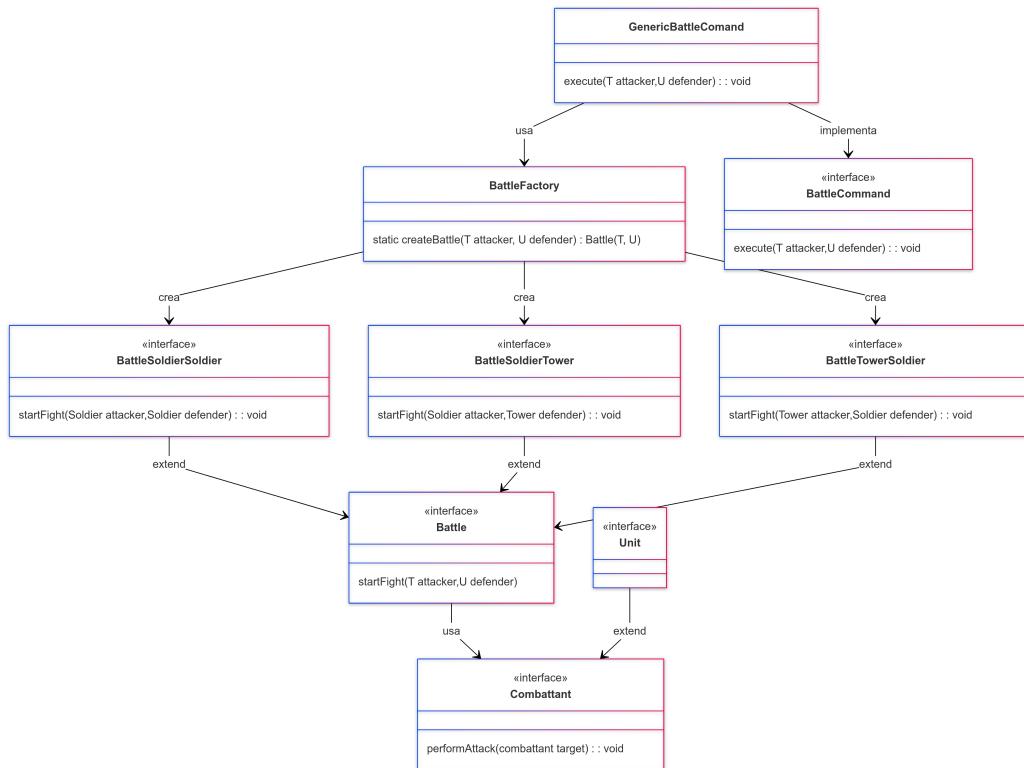


Figura 2.17: Schema UML per le gestione della battaglia

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per verificare il corretto funzionamento di ciascuna delle parti del progetto, in particolare del model, abbiamo creato dei test automatizzati con l'utilizzo di JUnit.

3.1.1 Balzani Riccardo

Tower Tests

- Verificano il corretto istanziamento di tutti i tipi di torre concreti.
- Controllano che le torri, una volta istanziate, abbiano tutte le caratteristiche attese.

Dice Tests

- Controllano che il dado produca esclusivamente valori compresi tra 1 e FACES.
- Verificano che il dado possa generare tutti i valori compresi tra 1 e FACES.

SaveData Tests

- Verificano che `GameDataImpl` contenga una deep copy di tutti gli oggetti che contiene.
- Controllano che il gioco venga salvato nella posizione specificata.

- Verificano che le eccezioni vengano effettivamente lanciate quando necessario.

LoadData Tests

- Controllano che il file dei salvataggi venga letto dal percorso specificato.
- Verificano la gestione della lettura da un file corrotto o inesistente.
- Controllano che tutti i dati letti siano coerenti con quelli precedentemente scritti sul file.

MainController Test

- Verifica che il processo di salvataggio/caricamento della partita si interrompa una volta superato il timeout stabilito.

Il metodo `cleanUp()` assicura che tutti i file utilizzati nei test di lettura o scrittura vengano eliminati al termine di ogni test, garantendo un ambiente pulito e privo di residui.

3.1.2 Bartolini Riccardo

CellsTest

- **CellTest:** Testa che gli attributi vengano inizializzati correttamente e che l'occupazione della cella gestita con gli Optional non dia errori in fase di costruzione.
- **BonusCellTest:** Testa che la BonusCell inizialmente sia 'settata' con i parametri corretti e che l'amount(bonus) sia corretto per l'attivazione.

MapTest

- **GameMapTest:** Testa che la mappa venga costruita nel modo corretto con l'utilizzo del Builder e che l'attivazione della BonusCell funzioni al passaggio o piazzamento di una unità sopra di essa.
- **MapManagerTest:** Testa il corretto setUp iniziale del MapManager e che vengano aggiunte alla lista di celle di proprietà di ogni Player le spawnCell all'inizio; testa che al momento del movimento da parte di un soldato verso un'altra cella vengano cambiate in modo corretto le occupazioni delle celle; testa l'impossibilità di comprare un nuovo

soldato se ne è già presente uno nella spawnCell e non è ancora stato mosso e ovviamente l'impossibilità di muoversi o piazzare unita in celle ostacoli ed infine il corretto match delle spawn cell per ogni player .

3.1.3 Fabbri Gianmarco

- **SoldierTest:** Verifica l'inizializzazione corretta dei soldati, inclusa la corretta assegnazione del proprietario e della posizione. Testa inoltre le funzionalità di aggiornamento del livello, gli attacchi tra soldati e torri e tra due soldati, assicurando che le modifiche agli attributi avvengano come previsto.
- **PlayerTest:** Controlla l'inizializzazione dei giocatori, inclusi nome, punto di spawn e saldo iniziale. Testa la gestione delle risorse tramite metodi di guadagno e spesa, l'aggiunta e rimozione di unità, nonché l'acquisto e l'upgrade delle unità. Verifica anche l'implementazione dei metodi ‘equals’ e ‘hashCode’, e il corretto funzionamento del costruttore di copia.
- **BotTest:** Assicura che il bot venga inizializzato correttamente con le unità vuote e il saldo iniziale appropriato. Testa la valutazione delle mosse del bot utilizzando il DecisionMaker, verifica la scelta della mossa migliore e controlla l'esecuzione delle azioni del bot, come l'acquisto di torri, garantendo che le unità vengano aggiunte correttamente e che le celle siano occupate senza errori.

3.1.4 Francalanci Filippo

- **GameTurnTest:** Verifica il funzionamento del cambio di turni, quindi quando la mano di gioco passa da un giocatore all'altro con conseguenza aumento di turno. Viene testato anche il cambio di turno automatico quando il giocatore non effettua mosse entro 4 giorni che sono gestiti da un timer.
- **GameDayTest:** Viene testato il funzionamento dell'aumento dei giorni e l'aumento del bilancio utilizzando l'observer pattern dove quando il giorno aumenta vengono notificati i player
- **BattleTest:** Assicura che funzioni correttamente il GenericBattleCommand tra Soldato-Soldato, Soldato-Torre e Torre-Soldato. Tra soldato e Soldato viene controllato che uno dei due soldati (o entrambi in caso di parità) vengano eliminati. Nel test tra Soldato e Torre viene testato

se il soldato infligge danno alla torre. Per la battaglia tra Torre e Soldato viene controllato se il soldato viene colpito e si trova nel raggio di azione della truppa

3.2 Note di sviluppo

3.2.1 Balzani Riccardo

Utilizzo di Optional

Gli `Optional` sono stati utilizzati in diversi punti del progetto. Un esempio è disponibile al seguente link: <https://github.com/Gianmarco-Fabbri/OOP23-100DaysWar/blob/main/src/main/java/it/unibo/the100dayswar/model/loaddata/impl/GameLoaderImpl.java>.

Utilizzo di Stream

Gli `Stream` sono stati impiegati in una sola occasione, durante la fase di test: <https://github.com/Gianmarco-Fabbri/OOP23-100DaysWar/blob/main/src/test/java/it/unibo/the100dayswar/model/loaddata/GameLoaderTest.java>.

Utilizzo di Lambda Expression

Le *lambda expressions* sono state usate in vari punti. Un esempio è visibile in: <https://github.com/Gianmarco-Fabbri/OOP23-100DaysWar/blob/main/src/main/java/it/unibo/the100dayswar/model/tower/impl/TowerFactoryImpl.java>.

Utilizzo di ExecutorService e Callable

`ExecutorService` e `Callable` sono stati utilizzati esclusivamente nel controller: <https://github.com/Gianmarco-Fabbri/OOP23-100DaysWar/blob/main/src/main/java/it/unibo/the100dayswar/controller/maincontroller/impl/MainControllerImpl.java>.

Risorse Esterne Utilizzate

Durante la fase di test, ho dovuto affrontare il problema di terminare le operazioni di salvataggio e caricamento del gioco nel caso in cui il tempo di attesa fosse eccessivo. Questa necessità è nata dall'esigenza di evitare che

l'utente finale, in caso di comportamenti inattesi, rimanga in uno stato di attesa infinita, bloccando il programma.

Per risolvere questo problema e notificare il fallimento all'utente, ho preso ispirazione dalla seguente discussione su StackOverflow: <https://stackoverflow.com/questions/16121176/skip-function-if-it-takes-too-long>.

Tuttavia, la mia implementazione si è discostata significativamente dalla versione proposta nel forum. Le principali modifiche includono:

- **Gestione più corretta delle eccezioni:** Attraverso l'uso di `Logger` per registrare eventuali errori.
- **Separazione tra definizione ed esecuzione del task:** Utilizzando `Callable`, ho garantito una maggiore modularità del codice.
- **Gestione del timeout:** In caso di superamento del limite di tempo, il task contenuto in `Callable` viene terminato e le risorse utilizzate vengono correttamente liberate.

Per il caricamento della partita corrente in memoria, ho tratto ispirazione dalla seguente discussione su StackOverflow: <https://stackoverflow.com/questions/28948315/save-game-data-java>.

Anche in questo caso, la mia soluzione si è differenziata in diversi aspetti. Il codice che ho fornito implementa:

- **Gestione più corretta delle eccezioni:** Con l'aggiunta di `Logger` per migliorare la tracciabilità.
- **Supporto per percorsi personalizzati:** Permettendo di specificare un *path* custom per il file di salvataggio.
- **Container migliorato:** Ho progettato `GameDataImpl` in modo più completo, garantendo una gestione coerente e modulare dei dati salvati.

3.2.2 Bartolini Riccardo

- **Utilizzo di Optional:** Utilizzati in vari punti , in particolare in relazione alla occupazione delle celle da parte di una unità. Esempio: <https://github.com/Gianmarco-Fabbri/OOP23-100DaysWar/blob/e9cc675a6a29c57f6src/main/java/it/unibo/the100dayswar/model/map/impl/GameMapImpl.java#L95-L110>
- **Utilizzo di Stream:** Utilizzati diverse volte per ciclare le celle della mappa , in particolare per avere uno Stream di Cell per poter fare più

comodamente controlli dal mapManager piuttosto che farli su una griglia di Cell. Esempio:<https://github.com/Gianmarco-Fabbri/OOP23-100DaysWar/blob/e9cc675a6a29c57f687ae683cc1e503a337207de/src/main/java/it/unibo/the100dayswar/model/map/impl/GameMapImpl.java#L85-L93>

- **Utilizzo di Lambda Expression:** Utilizzate anche esse in vari punti. Esempio:<https://github.com/Gianmarco-Fabbri/OOP23-100DaysWar/blob/e9cc675a6a29c57f687ae683cc1e503a337207de/src/main/java/it/unibo/the100dayswar/model/map/impl/MapManagerImpl.java#L185-L191>

3.2.3 Fabbri Gianmarco

- **Utilizzo di Generic Type:** Utilizzati solo in una classe: GenericPlayerCommand. <https://github.com/Gianmarco-Fabbri/OOP23-100DaysWar/blob/b025e0c9c4d4c38e1ab31cf93273cf4950fe1c37/src/main/java/it/unibo/the100dayswar/model/playeraction/api/GenericPlayerCommand.java#L7C1-L21C2>
- **Utilizzo di librerie esterne:** Utilizzate solo in una classe: ActionType. <https://github.com/Gianmarco-Fabbri/OOP23-100DaysWar/blob/b025e0c9c4d4c38e1ab31cf93273cf4950fe1c37/src/main/java/it/unibo/the100dayswar/model/bot/impl/ActionType.java#L13>
- **Utilizzo di Stream:** Utilizzati in diversi punti del codice, di seguito un esempio. <https://github.com/Gianmarco-Fabbri/OOP23-100DaysWar/blob/b025e0c9c4d4c38e1ab31cf93273cf4950fe1c37/src/main/java/it/unibo/the100dayswar/model/bot/impl/ActionType.java#L197C17-L200C60>
- **Utilizzo di Optional:** Utilizzati in una sola classe: DecisionMakerImpl. <https://github.com/Gianmarco-Fabbri/OOP23-100DaysWar/blob/b025e0c9c4d4c38e1ab31cf93273cf4950fe1c37/src/main/java/it/unibo/the100dayswar/model/bot/impl/DecisionMakerImpl.java#L42C9-L46C41>
- **Utilizzo di Lambda Expression:** Utilizzate in vari punti del codice, di seguito un esempio. <https://github.com/Gianmarco-Fabbri/OOP23-100DaysWar/blob/b025e0c9c4d4c38e1ab31cf93273cf4950fe1c37/src/main/java/it/unibo/the100dayswar/model/bot/impl/ActionType.java#L181C13-L187C16>

3.2.4 Francalanci Filippo

- **Utilizzo di Generic Type:** utilizzati nelle classi di battle in particolare nella factory e ne genericBattleCommand. Esempio: <https://github.com/Gianmarco-Fabbri/OOP23-100DaysWar/blob/b025e0c9c4d4c38e1ab31cf93273cf4950fe1c37/src/main/java/it/unibo/the100dayswar/model/bot/impl/ActionType.java#L13>

github.com/Gianmarco-Fabbri/OOP23-100DaysWar/blob/e9cc675a6a29c57f687ae683cc1e503a337207de/src/main/java/it/unibo/the100dayswar/model/fight/impl/GenericBattleCommand.java#L12

- **Utilizzo di Lambda:** Utilizzate in alcuni punti nelle classi che gestito.
Esempio: <https://github.com/Gianmarco-Fabbri/OOP23-100DaysWar/blob/e9cc675a6a29c57f687ae683cc1e503a337207de/src/main/java/it/unibo/the100dayswar/model/turn/impl/GameTurnManagerImpl.java#L179-L184>

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Balzani Riccardo

Valutazione del Lavoro

Valuto il mio contributo al progetto *The100DaysWar* in maniera estremamente positiva, non solo per il codice sviluppato, ma soprattutto per l'impegno e la dedizione investiti in ogni fase del progetto. Non avrei mai immaginato di dedicare così tanto tempo ed energie a un progetto universitario, ma sono pienamente soddisfatto della scelta fatta. Durante lo sviluppo, mi sono trovato frequentemente a rivalutare e rivedere le idee iniziali, un processo che ha richiesto ulteriori risorse in termini di tempo ed energia, ma che ha portato a un risultato finale di cui sono orgoglioso.

Il progetto mi ha insegnato molto, non solo dal punto di vista tecnico della programmazione, ma anche in termini di organizzazione e gestione. Ho avuto un assaggio di cosa significhi sviluppare un software con l'obiettivo di garantire espandibilità, manutenzione e compattezza.

Ruolo e Contributo

A livello tecnico, mi sono occupato di:

- Implementare le caratteristiche principali delle torri.
- Sviluppare i meccanismi di salvataggio e caricamento delle partite.
- Realizzare il sistema di lancio dei dadi.

- Collaborare alla scrittura delle classi `Model` e `MainController`.
- Contribuire significativamente alla parte `View` e alla creazione di classi di utility.

Nonostante in alcuni casi avessi preferito implementare certe funzionalità in modo diverso, sono soddisfatto del risultato complessivo.

Competenze Acquisite

Questo progetto mi ha permesso di sviluppare e consolidare diverse competenze, tra cui:

- Lavorare in gruppo in maniera efficace.
- Ricercare informazioni utili per implementare funzionalità nuove e inizialmente sconosciute.
- Utilizzare strumenti di versionamento come Git/GitHub.
- Organizzare il codice in maniera professionale, seguendo buone pratiche di design.

Piani Futuri

Se avessi l'opportunità di continuare a lavorare su questo progetto, mi piacerebbe:

- Implementare una logica multithreading per rendere la gestione della GUI più fluida.
- Aggiungere diversi livelli di difficoltà per il bot, migliorando così l'esperienza di gioco.

Difficoltà Affrontate

Le principali difficoltà incontrate durante lo sviluppo del progetto sono state:

- Suddividere il lavoro iniziale.
- Utilizzare GitHub in modo efficace.
- Implementare classi altamente estendibili.
- Integrare le funzionalità sviluppate dai miei compagni nel mio codice.

- Mantenere una comunicazione costante con i miei compagni.

Per quanto riguarda quest'ultima difficoltà, desidero sottolineare che è stata causata principalmente dal fatto che ho svolto l'intero progetto a distanza. Da settembre, infatti, sono in Erasmus e, di conseguenza, non ho potuto confrontarmi dal vivo con i miei compagni. Inoltre, gli orari di routine e universitari spesso non erano compatibili, complicando la comunicazione. In ogni caso penso di aver lavorato in sintonia con i miei compagni.

Nonostante queste sfide, sono molto soddisfatto di come, come gruppo, siamo riusciti a gestire queste complicazioni e a portare a termine il progetto con successo.

4.1.1 Bartolini Riccardo

Per me questa è stata la mia prima esperienza in un lavoro di gruppo per un progetto di questa portata, nonostante ciò ritengo di essermi impegnato al massimo per svolgere nel modo migliore possibile la mia parte in modo che poi fosse più facile anche per gli altri componenti del gruppo integrare le loro parti per l'assemblaggio poi di tutte le parti nel progetto. Le difficoltà maggiori che personalmente ho riscontrato all'inizio sono state sicuramente la gestione del progetto in condivisione tramite GitHub poiché per quanto sia utile e comodo all'inizio ho avuto bisogno di prenderci un po' la mano soprattutto con i vari branch, l'altra più grande difficoltà è stata la suddivisione equa dei ruoli nelle varie parti del progetto poiché a priori è stato difficile stimare la quantità di carico di lavoro che avrebbe avuto ciascuna parte. I miei più grandi punti di debolezza li riconosco soprattutto nella gestione iniziale che, a mio avviso, è la parte fondamentale, nella suddivisione e progettazione del tutto partendo da zero nella quale mi sono un po' bloccato non sapendo bene dove mettere le mani, però sono molto soddisfatto della piega che ha preso il lavoro di gruppo appena impostate le 'fondamenta' del progetto. Nonostante le diverse sfide che ci sono state, mi ritengo personalmente molto soddisfatto del risultato ottenuto e di come ha lavorato tutto il gruppo in coesione per arrivare alla fine del progetto, è stata sicuramente una ottima occasione per imparare oltre che ovviamente alcuni lati della programmazione ad oggetti magari per noi sconosciuti anche come poter dare al meglio il proprio contributo all'interno di un team.

4.1.2 Fabbri Gianmarco

Essendo la prima esperienza in un progetto di questa mole le difficoltà sono state molteplici, una fra le principali la suddivisione del carico di lavoro

a monte, in quanto, non avendo esempi pratici precedenti, stimare quanto tempo avrebbe impiegato l'ideazione e l'implementazione di ciascuna parte ha richiesto tempo per poi non assicurare una divisione prettamente equa. Di seguito l'uso di git inizialmente l'ho trovato personalmente ostico mentre in una seconda parte è stato sempre più facile utilizzarlo anche se con qualche errore. Oltre alle difficoltà ci sono molte note positive, la parte di architettura del dominio è stata una delle più stimolanti, lavorare in team per quanto crei qualche discordanza alla fine dei conti si è rivelato divertente e produttivo in quanto, tutti i membri si sono sostenuti a vicenda quando si sono incontrate delle complicanze. Tirando le somme mi ritengo ampliamente soddisfatto del lavoro eseguito negli ultimi mesi e del risultato finale.

4.1.3 Francalanci Filippo

Questa è stata la mia prima esperienza in un progetto di tale durata. Inizialmente, la fase di modellazione delle classi si è rivelata particolarmente complessa, soprattutto per garantire che fossero efficienti e coerenti. I compiti sono stati distribuiti in modo equo tra i membri del team. Ho incontrato alcune difficoltà nel trovare pattern adeguati per le mie classi, senza renderle troppo complesse. Tuttavia, con il proseguire del progetto, è diventato più facile comprendere il funzionamento delle classi e come gestirle al meglio. La parte relativa al controller e alla view, che inizialmente pensavo fosse la più difficile, si è invece rivelata abbastanza intuitiva. Lavorare in team, nonostante possa essere a volte impegnativo e stressante, ha offerto anche aspetti positivi: ci siamo supportati a vicenda e aiutati nei momenti di difficoltà. Alla fine, l'esperienza si è rivelata molto gratificante e il risultato finale è stato decisamente soddisfacente.

Appendice A

Guida utente

Tutorial di The100DaysWar

Menu Iniziale

All'avvio dell'applicazione, ti troverai davanti al menu iniziale, che presenta quattro pulsanti principali:

- **Start**: Avvia una nuova partita.
- **Resume**: Riprende l'ultima partita salvata.
- **Rules**: Mostra le regole dettagliate del gioco.
- **Exit**: Esce dall'applicazione.

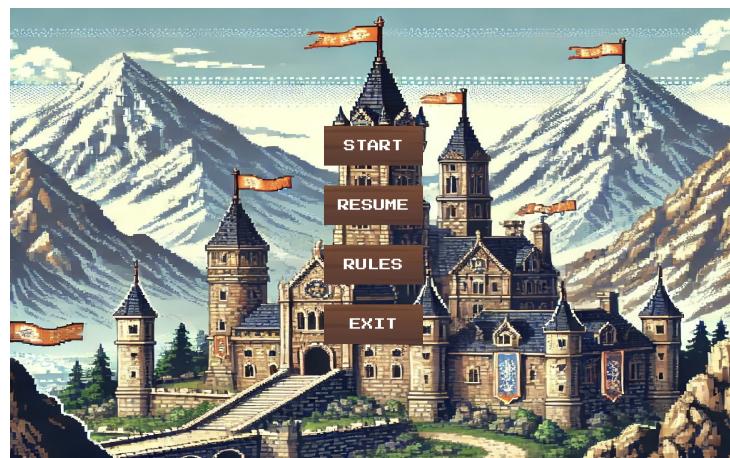


Figura A.1: Schermata del menù iniziale

Regole del gioco: Premendo il pulsante **Rules**, verrà mostrata una schermata contenente una spiegazione dettagliata delle regole di gioco.

Uscita dall'applicazione: Premendo il pulsante **Exit**, ti verrà data la possibilità di chiudere l'applicazione.

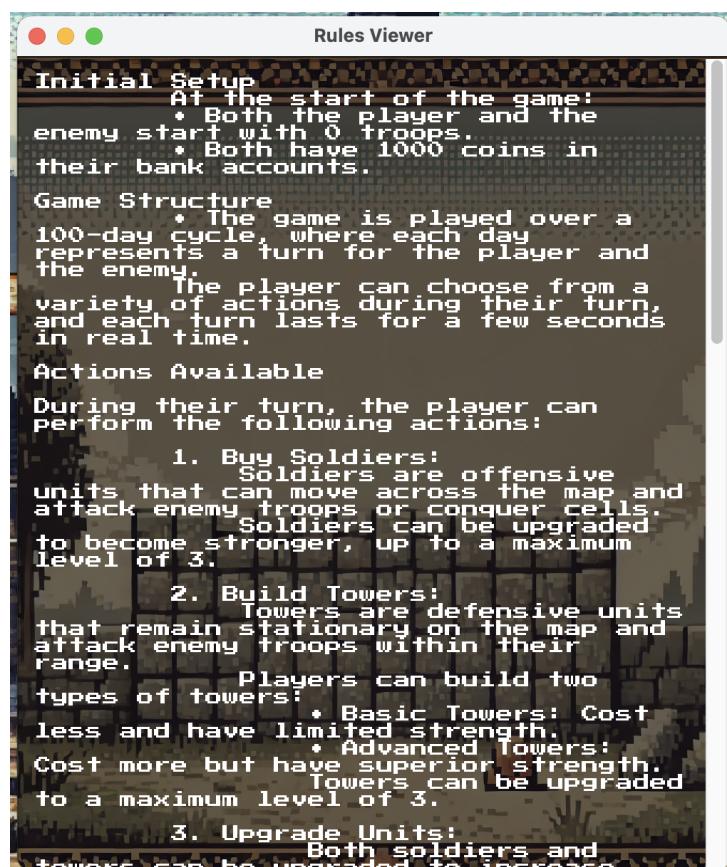


Figura A.2: Regole del gioco

Avvio di una Nuova Partita

Se premi il pulsante **Start**, ti verrà chiesto di inserire un nome per il tuo profilo di gioco. Il nome deve contenere al massimo 8 caratteri. Una volta inserito un nome valido, verrà generata una nuova partita con una mappa casuale e spawn iniziali posizionati in modo casuale. A questo punto si aprirà la schermata di gioco.



Figura A.3: Menù per inserire i Nomi

Schermata di Gioco

La schermata di gioco è suddivisa in tre sezioni principali:

1. **Mappa di Gioco (a sinistra)**: Qui si svolgeranno le principali azioni di gioco.
2. **Contatore e Statistiche (in alto a destra)**: Mostra i giorni trascorsi e le tue statistiche attuali.
3. **Pulsanti di Controllo (in basso a destra)**: Contengono opzioni di gioco, come la creazione di unità e l'acquisto di torri.

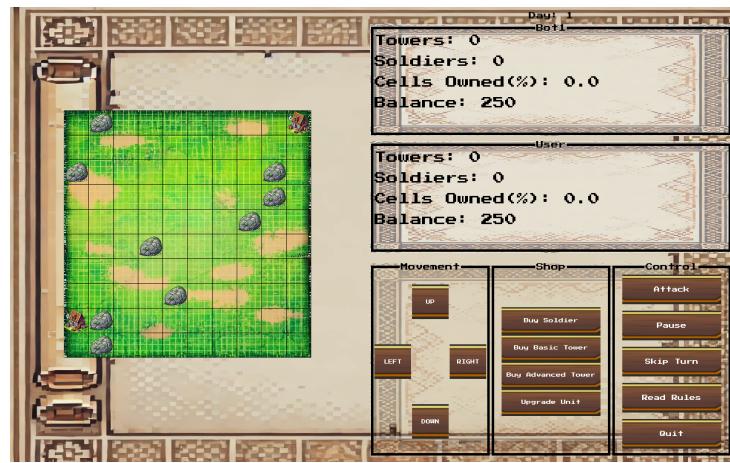


Figura A.4: Questa è tutta la schermata di gioco

La Mappa

Sulla mappa noterai due casette:

- **A destra:** Rappresenta il tuo spawn, dove puoi generare soldati.
- **A sinistra:** Rappresenta lo spawn dell'avversario, il tuo obiettivo da conquistare.



Figura A.5: Questa è la mappa di gioco

Generazione di Soldati

Per generare un soldato:

1. Clicca sul tuo spawn (la casella sottostante diventerà più scura).
2. Premendo il tasto appropriato, apparirà un soldato di primo livello.

Una volta generato, il soldato potrà essere selezionato e mosso sulla mappa.

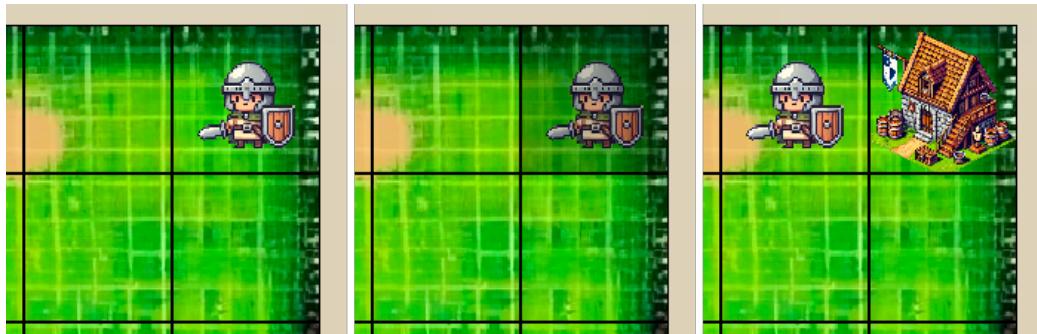


Figura A.6: Fase di spawn, selezione e movimento

Azioni Disponibili

- **Costruzione di Torri:**

- Per costruire una torre, seleziona una casella libera e premi il pulsante **Buy Tower**.
- Puoi scegliere tra due tipi di torre: **Basic Tower** e **Advanced Tower**.

- **Acquisto e upgrade delle Unità:**

- Per migliorare un'unità, selezionala e premi il pulsante **Upgrade Unit**.
- Ogni unità può essere migliorata fino al livello massimo di 3.



Figura A.7: Joystic shop e torri piazzate

- **Conquista del Territorio:**

- Muovendo i tuoi soldati su caselle vuote, queste verranno aggiunte al tuo territorio, migliorando le statistiche.

- **Interazione con le Torri Nemiche:**

- Se un soldato entra nel raggio di una torre nemica (due caselle), subirà attacchi automatici dalla torre.

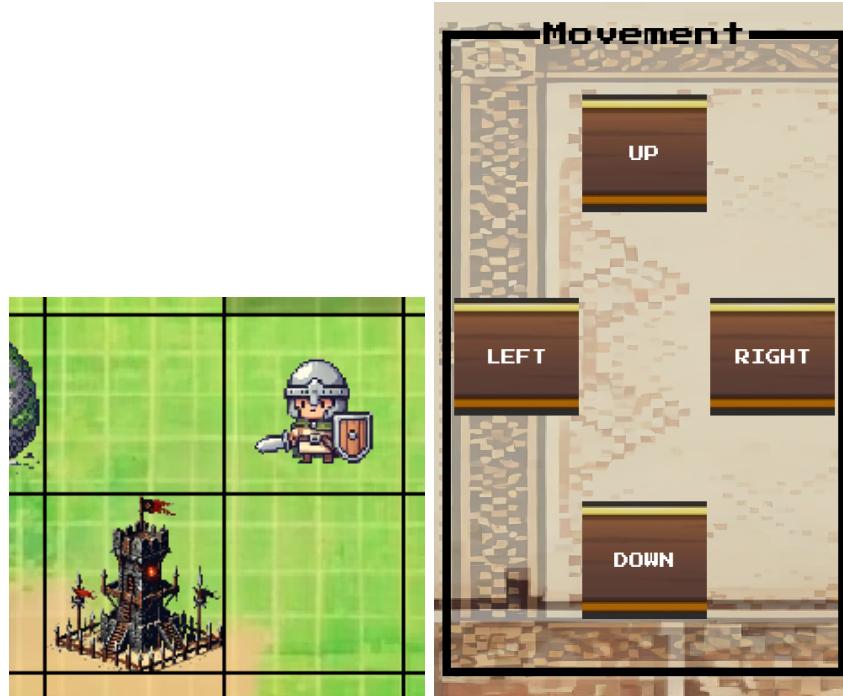


Figura A.8: Interazioni con le torri e joystic movimento

Condizioni di Fine Partita

La partita può terminare in due modi:

1. **Conquista dello Spawn Nemico:** Quando uno dei giocatori arriva con un soldato sulla casella dello spawn avversario.
2. **Fine dei 100 Giorni:** Se nessuno ha conquistato lo spawn nemico entro 100 giorni, il vincitore sarà determinato dal numero di celle conquistate.

Alla fine della partita, potrai chiudere l'applicazione o tornare al menu iniziale.



Figura A.9: Schermata di Game Over

Pulsanti di Controllo

Durante il gioco, puoi utilizzare i seguenti pulsanti nella sezione di controllo:

- **Attacco ai Nemici:**

- Per attaccare un soldato nemico, avvicina un tuo soldato alla casella adiacente, selezionalo e premi il pulsante **Attack**.
- Un'animazione del dado deciderà il vincitore, mentre il perdente scomparirà dalla mappa.

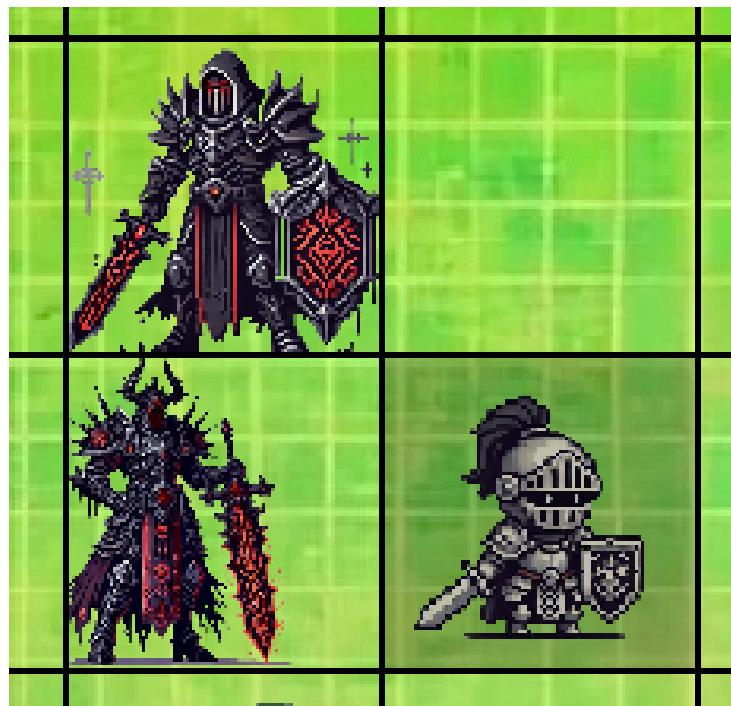


Figura A.10: combattimento

- **Skip:** Salta il turno.
- **Read Rules:** Consulta nuovamente le regole.
- **Quit:** Esci dalla partita senza salvare.
- **Pause:** Apre la schermata di pausa.

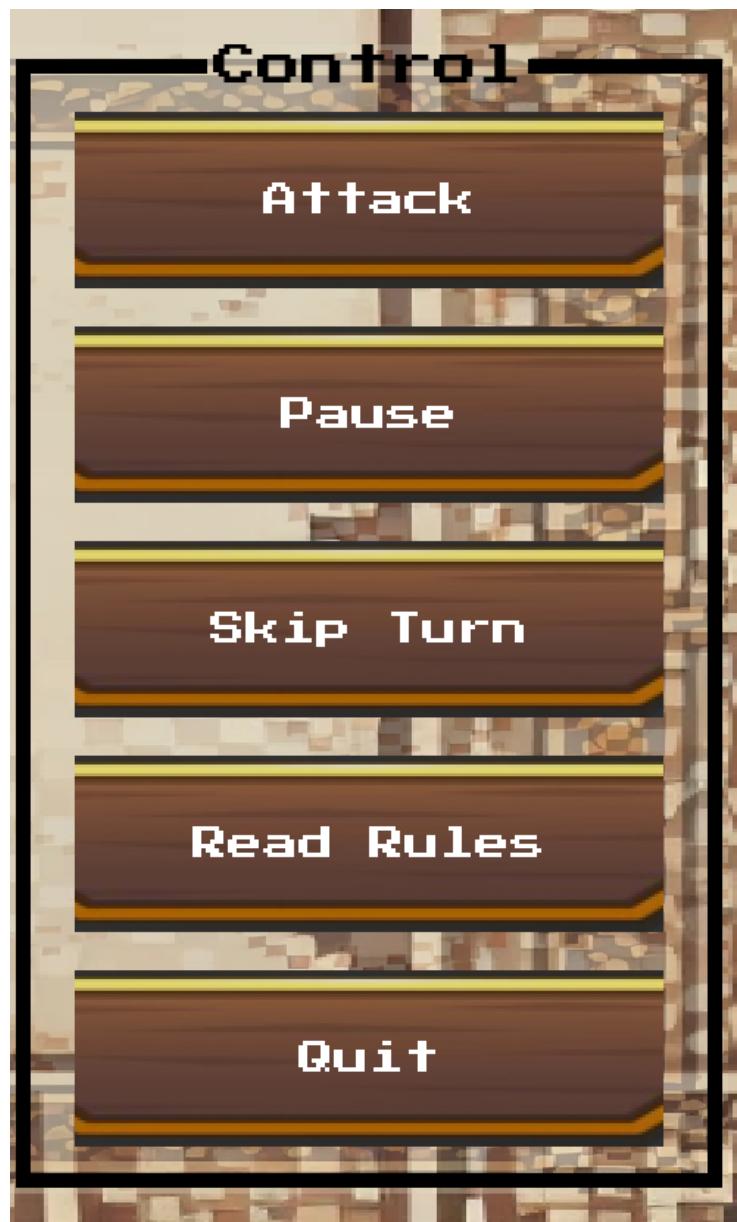


Figura A.11: Joystic control

Schermata di Pausa

La schermata di pausa offre le seguenti opzioni:

- **Resume:** Riprende la partita.
- **Menu:** Torna al menu iniziale.

- **Exit:** Esce dal gioco.



Figura A.12: Schermata di pausa

Se premi **Menu** o **Exit**, ti verrà chiesto se vuoi salvare la partita in corso. In caso positivo, il salvataggio potrà essere ripreso successivamente premendo **Resume** nel menu iniziale.



Figura A.13: Finestra di messaggio che appare quando decidi di uscire

Conclusione

Questo tutorial ti guida attraverso tutte le funzionalità principali di *The100DaysWar*, assicurandoti un'esperienza di gioco fluida e piacevole. Esplora il gioco, utilizza le strategie a tua disposizione e conquista il campo di battaglia!