Relazione
“ClashClass”

Alessandro Ricci
Lucas Antonio Leonte
Francesco Volpini
Lorenzo Bulfoni

5 giugno 2025

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il progetto si pone come obiettivo quello di realizzare un clone semplificato del famoso videogioco per mobile "Clash of Clans", dove il giocatore ha a disposizione un villaggio che può espandere attraverso la costruzione di nuovi edifici, rubando le risorse necessarie nel corso degli attacchi effettuati, con le proprie truppe, nei confronti dei villaggi appartenenti ad altri giocatori nel mondo.

Requisiti funzionali obbligatori

- L'utente avrà la possibilità di visualizzare graficamente il proprio villaggio, organizzato a griglia, e composto da diversi edifici, tra i quali:
  - Town Hall: il municipio del villaggio, un edificio unico che rappresenta la parte fondamentale del villaggio stesso
  - Wall: le mura del villaggio, con cui è possibile proteggere gli edifici interni
  - Cannon: uno degli edifici di difesa del villaggio. Durante le battaglie, è in grado di danneggiare le truppe che vengono schierate dal giocatore attaccante
- Il gioco dovrà essere in grado di avviarsi caricando i dati del villaggio dell'utente, dapprima creandone uno di default e, successivamente, utilizzando un sistema di salvataggio per rendere persistenti le modifiche che avvengono all'interno dell'applicazione.
- L'utente potrà accedere ad un negozio, dentro al quale è possibile acquistare nuovi edifici per ampliare il proprio villaggio, ciascuno dei quali avrà un prezzo e un tipo di risorsa necessaria all'acquisto (oro o elisir, da verdersi come diverse "valute" monetarie)
- L'utente potrà entrare in modalità battaglia, potendo attaccare un altro villaggio preimpostato con le truppe che possiede.
- Le truppe dovranno essere in grado di muoversi in autonomia verso gli edifici del villaggio per distruggerli, saccheggiando le risorse dai depositi di oro ed elisir.
- la battaglia dovrà essere in grado di terminare quando si verificano una tra le seguenti condizioni:

  - Tutto il villaggio viene distrutto (ad eccezione delle mura, che non contribuiscono al conteggio della percentuale di distruzione perché sono viste come semplici ostacoli per le truppe)
  - Tutte le truppe schierate vengono sconfitte, e non si hanno più truppe disponibili
  - Il timer che indica il tempo massimo di battaglia giunge al termine
  - L'utente sceglie di terminare la battaglia manualmente

- Dovrà essere mostrato un report dettagliato di quello che è avvenuto durante la battaglia, come il numero di stelle ottenute (1 stella per il 50% di distruzione, 1 per la distruzione del municipio e 1 per il 100% di distruzione) e la quantità di risorse rubate

- L'utente dovrà essere in grado di tornare al proprio villaggio come all'apertura dell'applicazione, visualizzando le modifiche derivanti dall'esito dell'avvenuta battaglia

Requisiti non funzionali

- L'applicazione dovrà essere in grado di gestire la grande quantità di oggetti in gioco in maniera fluida, senza intaccare gravemente sulle prestazioni generali, soprattutto quando, in modalità battaglia, ogni singola truppa schierata dovrà utilizzare un sistema di movimento intelligente assestante, che potrebbe gravare pesantemente sul processore

- Per attenersi al videogioco originale, l'applicazione dovrà disegnare tutti gli oggetti in scena utilizzando una prospettiva detta "isometrica"

1.2 Modello del dominio

L'applicazione si suddivide logicamente in due principali stati di gioco in cui si può trovare in un determinato momento: la modalità "villaggio utente" e la modalità "battaglia".
Nella prima modalità, l'utente può visualizzare la disposizione a griglia degli edifici proprio villaggio e i dettagli riguardanti le risorse possedute; inoltre può accedere al negozio per acquistare nuovi edifici.
Nella seconda modalità, l'utente visualizza il villaggio da attaccare, e può schierare le proprie truppe nelle posizioni valide della griglia (ovvero, nelle celle che non sono già occupate da edifici e al tempo stesso non sono al di fuori della griglia). Ogni truppa ha un comportamento autonomo: attraverso una "pseudo" intelligenza artificiale, compie ciclicamente delle azioni specifiche, come la scelta del prossimo edificio da attaccare, la costruzione del percorso più breve per raggiungere tale edificio, e il danneggiamento dell'edificio stesso (sottraendo il valore della propria statistica di danno dalla statistica di vita posseduta dall'edificio). Allo stesso tempo, gli edifici di difesa del villaggio (come il cannone e la torre dell'arciere) hanno un comportamento simile, colpendo e danneggiando le truppe che si avvicinano a loro, attraverso il medesimo sistema di statistiche. In aggiunta, le difese del villaggio possiedono anche un raggio di azione, espresso in quantità di celle della griglia entro il quale si "attivano" per danneggiare le truppe più vicine. Al termine della battaglia, che avviene quando si verifica una delle condizioni già elencate nell'analisi

dei requisiti, viene mostrato all'utente una schermata di riepilogo della battaglia appena avvenuta.
In generale, quindi, un entità generica di tipo "villaggio" possiede sia le informazioni relative al giocatore (fisico o virtuale) associato a quel villaggio (come le risorse possedute), sia l'elenco di edifici posseduti, rappresentati come generici GameObject. La modalità "villaggio utente" gestisce il negozio, e la modalità "battaglia" gestisce lo schieramento truppe e il riepilogo finale.

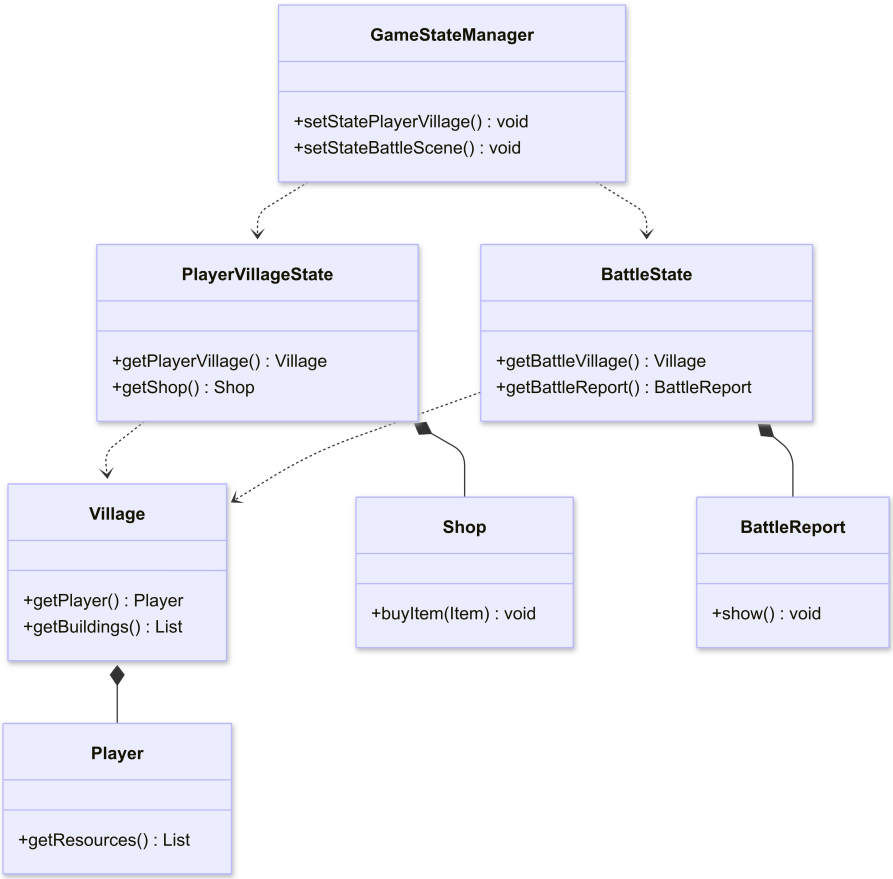

Figura 1.1: Schema UML dell'analisi del modello, con le entità principali e i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Il progetto è stato realizzato favorendo l'uso del pattern architetturale MVC, per separare correttamente le parte di logica da quella di grafica.
Alla radice del progetto, viene utilizzato uno State pattern per dividere le scene disponibili in diverse stati, di cui solo uno può essere attivo in un determinato momento.
Inoltre, è stato utilizzato il pattern "ECS" (Entity-Component System) per rappresentare e gestire intelligentemente la complessa natura delle diverse entità in gioco (chiamate "GameObject"). Questo pattern, infatti, è divenuto ormai uno standard globale per tutte le applicazioni di motori grafici (come i famosi "Unity" e "Unreal Engine"), siano essi utilizzati per lo sviluppo di videogiochi o per applicazioni grafiche general-purpose. Il pattern ECS favorisce la composizione rispetto all'ereditarietà, creando svariati micro-componenti assestanti organizzati per piccole responsabilità (come un componente che rappresenta la vita di un giocatore, o un componente che si occupa solo di calcolare il danno effettuato, o ancora un componente che indica che una certa entità è "danneggiabile"). Questi componenti possono essere poi combinati come fossero "mattoncini" per comporre ogni singola entità presente in gioco. I moduli dell'MVC sono rappresentati da:

- **Model**: la logica del model è affidata ai singoli "state" che può assumere il gioco, ovvero "player village" e "battle", che contengono rispettivamente anche le logiche del negozio e del report di battaglia.

- **Controller**: il controller è rappresentato alla radice dal GameStateManager, che contiene tutti i singoli controller degli "state" che il gioco può assumere il un determinato momento. Al GameStateManager è anche affidato il compito di avviare il GameEngine, che a sua volta avvia il GameLoop, che ha il compito di gestire l'aggiornamento dei GameObject in gioco e la corretta sequenza dei frame.

- **View** la view è gestita dall'interfaccia "Graphic", che fornisce metodi di disegno grafico a prescindere dall'implementazione della libreria grafica che sarà poi scelta.

Essendo l'architettura MVC organizzata in questo modo, la parte di View può essere teoricamente implementata con qualsiasi libreria grafica, come JavaFX e Swing, e il passaggio tra queste non riguarda e non intacca la parte logica del sistema.

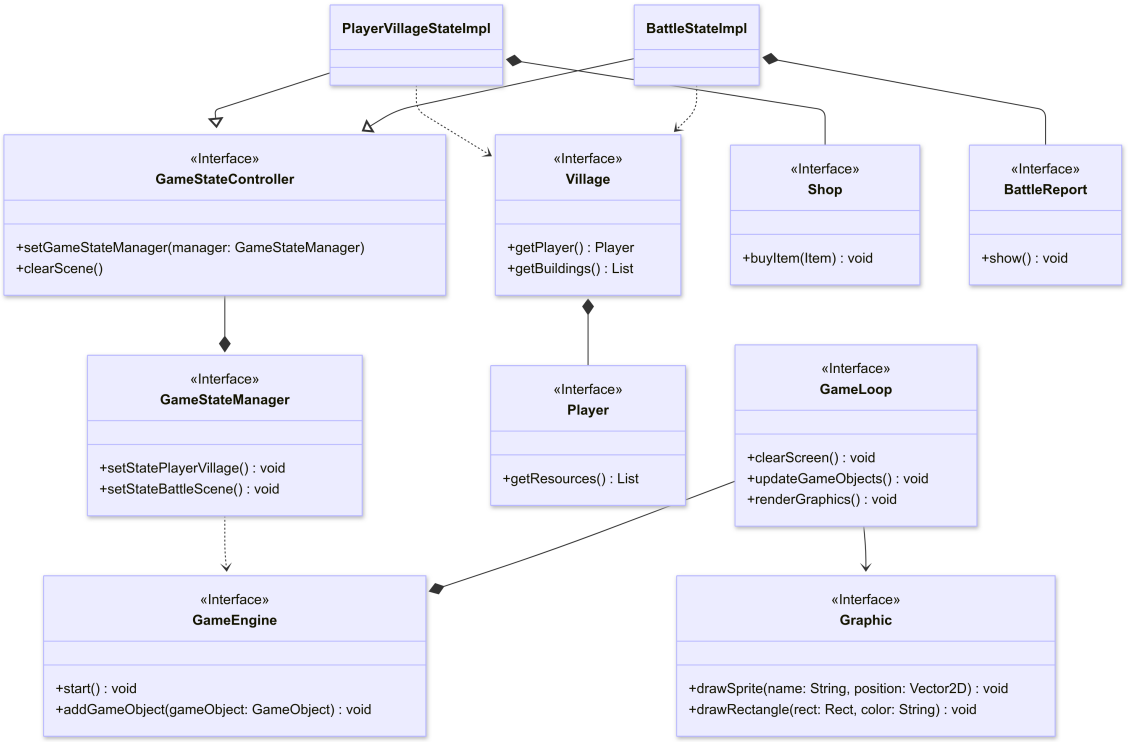

Figura 2.1: Schema UML dell'architettura

2.2 Design dettagliato

Alessandro Ricci

Creazione dell'Entity-Component System (ECS)

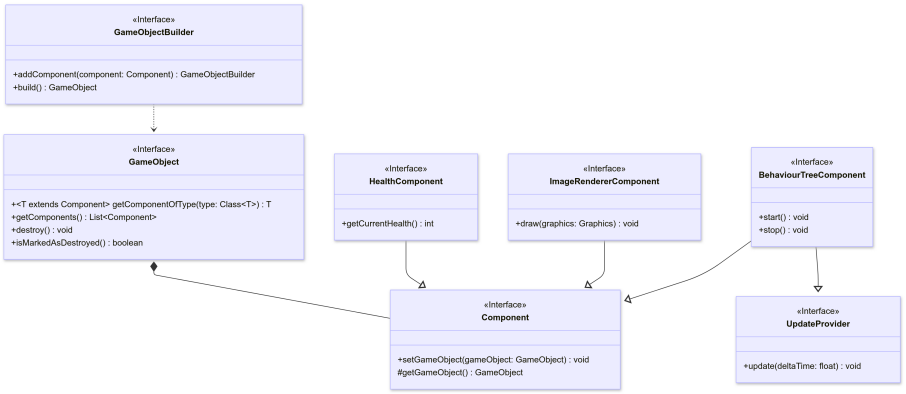

Figura 2.2: Rappresentazione UML del pattern ECS per la gestione delle entità (GameObject) in gioco

Problema Rapida perdita di controllo della complessità delle gerarchie di entità in caso di utilizzo della classica ereditarietà tra classi, a causa di comportamenti molto differenti e variegati che devono assumere le diverse entità.

Soluzione Utilizzo del pattern **Entity-Component System** per utilizzare la composizione al posto dell'ereditarietà.

In questo contesto, l'entità (rappresentata dal `GameObject`) nasce come un oggetto vuoto, e assume significato solo quando vengono aggiunti dei `Component` alla sua lista di componenti interna. I `Component` possono rappresentare qualsiasi micro-comportamento, concetto o informazione. I `Component` devono poi poter dialogare tra loro, per permettere la creazione del comportamento "complesso" desiderato per l'entità, inteso come combinazione dei comportamenti "semplici" dei vari componenti. Per fare questo, ad ogni `Component` viene passato un riferimento al `GameObject` a cui appartiene, attraverso il metodo `setGameObject()`. Dopodiché, dal `GameObject`, è possibile chiamare il metodo `getComponentOfType()` per restituire il primo componente della lista che sia del tipo desiderato.
Siccome non tutti i `Component` hanno bisogno di essere aggiornati ad ogni frame, si è creata un'interfaccia funzionale `UpdateProvider` da cui un `Component` può ereditare in caso abbia bisogno di essere aggiornato ad ogni frame (ad esempio, nell'UML fornito, `BehaviourTreeComponent`, usato per l'AI di edifici e truppe, eredita sia da `Component` che da `UpdateProvider`). Il GameLoop si occupa poi di aggiornare solo i `Component` che ereditano da `UpdateProvider`, ottimizzando così le prestazioni generali.

Gestione delle factory per quelle entità che hanno componenti diversi a seconda della scena

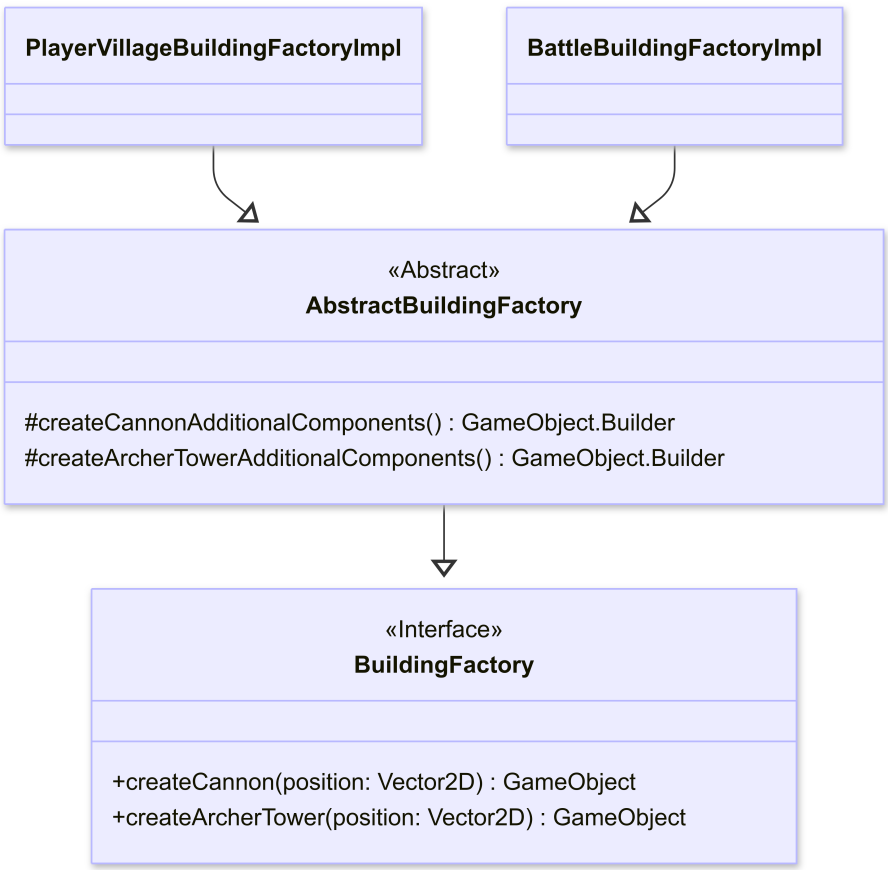

Figura 2.3: Rappresentazione UML delle Factory Method usate per gli edifici dei villaggi

Problema Alcune entità, come gli edifici del villaggio, pur essendo concettualmente lo stesso oggetto in entrambe le scene del gioco, hanno bisogno di essere create con diversi componenti a seconda che vengano istanziati nella scena "player village" o nella scena "battle". Questo perché, ad esempio, un edificio in battaglia deve possedere i componenti atti a farlo "combattere" contro le truppe attaccanti, come il `BehaviourTree` e i componenti che contengono i valori delle statistiche di battaglia (danno, raggio d'azione), ma NON deve avere questi stessi componenti in modalità "player village", poiché non sarebbero utilizzati.

Soluzione Utilizzo di gerarchia di factory, intesa come **Factory Method**. L'interfaccia `BuildingFactory` permette di creare gli edifici, come il cannone e la torre dell'arciere. È poi presenta una classe astratta `AbstractBuildingFactory` che implementa l'interfaccia precedente. Ogni di creazione viene implementato nel seguente modo: prima vengono aggiunti alcuni `Component` che l'edificio deve avere a prescindere dal contesto (quindi le proprietà in comune tra le scene, come ad esempio il componente `ImageRenderer` per disegnare la sprite dell'edificio a schermo), poi viene chiamato un metodo aggiuntivo, relativo allo stesso edificio, che consente di aggiungere altri componenti oltre a quelli già inseriti di default. Le implementazioni figlie di questa classe astratta hanno il compito di specializzare questi metodi "protected" con l'elenco di componenti da aggiungere rispetto al contesto (scena) corrente. Ad esempio, `BattleBuildingFactoryImpl` specializza il metodo `createCannonAdditionalComponents()` aggiungendo un `BehaviourTreeComponent` poiché necessario in modalità battaglia, ma `PlayerVillageBuildingFactoryImpl` specializza lo stesso metodo in un modo diverso, adatto alla propria scena.

Gestione del sistema di ricerca del percorso minimo di una truppa verso un edificio del villaggio

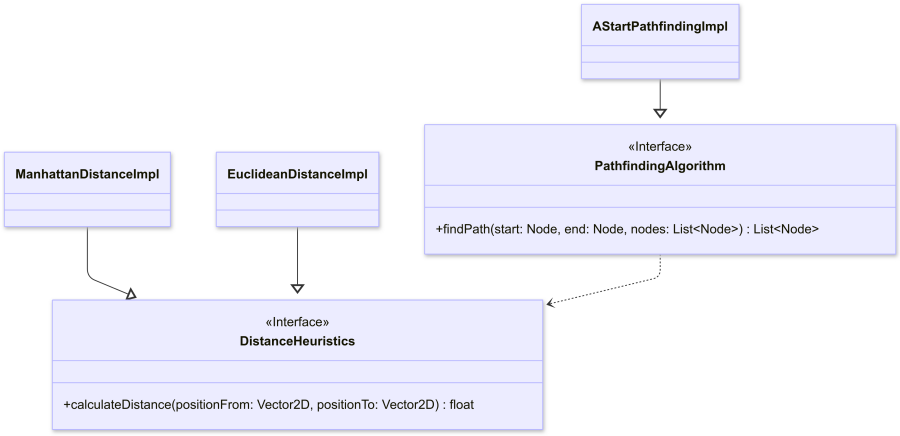

Figura 2.4: Rappresentazione UML del pattern Strategy usato per la ricerca del percorso minimo

Problema A fronte dello schieramento di una truppa in modalità battaglia, questa dovrebbe essere in grado di muoversi autonomamente verso l'edificio più vicino, trovando il percorso con costo minore. Questa necessità è resa ancora più complessa dalla presenza delle mura, che devono essere viste come ostacoli, e non ulteriori edifici ad distruggere. La loro distruzione, infatti, non contribuisce alla distruzione

del villaggio. Tuttavia, se sono disposte in maniera tale da circondare completamente l'edificio scelto come bersaglio, le truppe devono essere in grado di capire che non c'è alcun percorso valido per l'edificio, e quindi cambiare bersaglio sul muro più vicino, con l'obiettivo di distruggerlo e aprire così una "breccia" nel villaggio.

Soluzione Utilizzo del pattern **Strategy** per la creazione di interfacce funzionali che astraggono il concetto di algoritmi utili per questa ricerca del percorso minimo. In particolare, viene utilizzata un'interfaccia `PathfindingAlgorithm`, che rappresenta un generico algoritmo di ricerca del percorso minimo. Una sua implementazione è il noto algoritmo `A*` (letto "A-Star"), che è di gran lunga il più utilizzato nei videogiochi per la sua particolare velocità di esecuzione, tale da poter essere eseguito ad ogni frame e anche da più truppe nello stesso frame.
L'algoritmo di pathfinding necessita di un'euristica (ovvero di una stima) della distanza tra due nodi, che utilizza costantemente al suo intero per confrontare le coppie di nodi. Anche questa è implementata attraverso uno `Strategy`, fornendo due implementazioni classiche:

- **EuclideanDistance**: la distanza più accurata (quella effettiva, realistica) tra due punti nello spazio, che utilizza il teorema di Pitagora con il calcolo della radice quadrata.
- **ManhattanDistance**: detta anche "distanza quadrata", è una stima della distanza tra due punti che, al contrario della "EuclideanDistance" di prima, non utilizza la radice quadrata, ed è quindi più veloce. Il termine origina dal fatto che, nella città di Manhattan, essendo organizzata prevalentemente "a blocchi" con strade totalmente dritte (orizzontali o verticali), un taxi non può calcolare la distanza tra due punti "in linea d'aria", poiché non ci sarebbe alcuna strada percorribile che tagli in diagonale la città, ma deve essere vincolato dalla struttura "a scacchiera" delle strade, rivalutando quindi il concetto stesso di distanza.

È di importante rilevanza far notare che, come tutti gli algoritmi di ricerca del percorso minimo, è necessario fornire una rappresentazione "discreta" del terreno navigabile, approssimandolo con una serie di punti. Questa operazione è facilitata dal fatto che il villaggio del gioco è già organizzato a griglia, con un numero variabile di celle in orizzontale e in verticale.

Gestione dell'AI di truppe ed edifici con Behaviour Tree

Problema Data la diversa natura dell'intelligenza artificiale che deve essere impostata per truppe ed edifici, sarebbe necessaria l'implementazione di un sistema che sia in grado di minimizzare le ripetizioni di codice e rendere possibile modificare ed estendere facilmente i comportamenti assegnati a ciascuna entità in gioco.

Soluzione Utilizzo di una nota struttura dati denominata **Behaviour Tree**, una struttura ad albero organizzata a nodi, tale che ogni nodo possa avere il suo micro-comportamento specifico, ed essere riutilizzato e combinato con altri nodi per creare una gerarchia complessa, modulare e scalabile, che dia origine al macro-comportamento finale desiderato per ciascuna truppa ed edificio di difesa. In effetti, si tratta di un approccio molto simile al pattern **Composite**, già utilizzato nell'`Entity-Component System` descritto in precedenza.

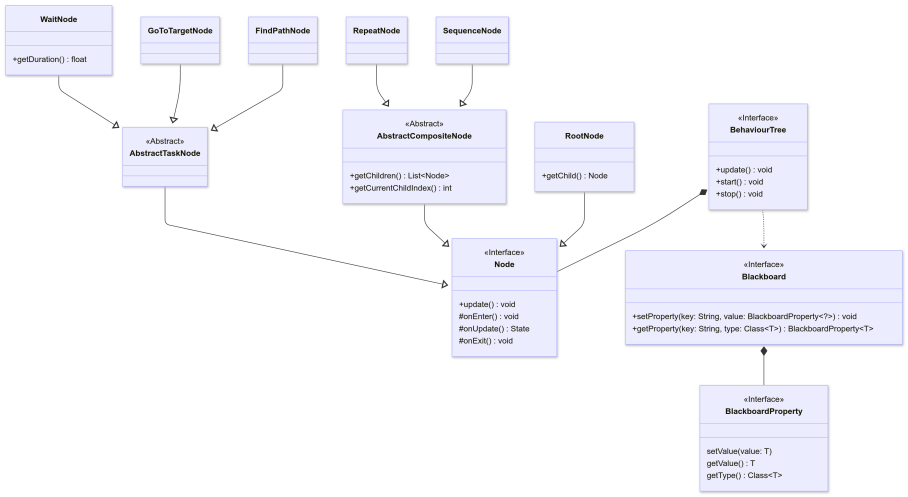

Figura 2.5: Rappresentazione UML della struttura del Behaviour Tree implementato nel progetto

In questa implementazione del `BehaviourTree` sono presenti due nodi astratti principali:

- `AbstractCompositeNode`: un nodo che può avere una lista di nodi figli.
- `AbstractTaskNode`: un nodo funzionale, che effettua una singola azione specifica e non può avere figli.

Combinando questi nodi, e implementandoli per singoli comportamenti specifici, ricavati dall'attenta suddivisione in parti del comportamento finale atteso, e possibile costruire una struttura ad albero modulare che può essere adattata alle esigenze di ogni diversa entità in gioco.
Il `BehaviourTree` utilizza internamente un pattern **Template Method**, poiché sono presenti tre metodi "protected" nella classe astratta del nodo che possono essere ridefiniti, del tutto o in parte, dalle varie classi figlie. In particolare:

- **onEnter()**: chiamato la prima volta che si entra nel nodo; può essere usato per inizializzare
- **onUpdate()**: chiamato ad ogni frame. Restituisce uno "State" che può essere [SUCCESS, FAILURE, RUNNING]. L'azione effettuata da un nodo può essere eseguita in sola chiamata (ad esempio, il nodo che trova il percorso minimo per raggiungere il bersaglio) oppure può durare più tempo (ad esempio, il `WaitNode`, che attende un tot di secondi prima di terminare e far proseguire l'esecuzione dei prossimi nodi). L'esecuzione dei nodi dell'albero è coordinata in base al valore di questo "State" restituito da questa funzione di ogni nodo
- **onExit()**: chiamato quando il nodo termina, prima di passare al prossimo nodo; può essere usato per liberare le risorse

Il `BehaviourTree` utilizza anche un sistema detto "`Blackboard`" per dare la possibilità ad ogni nodo interno di leggere e scrivere variabili arbitrarie condivise con tutto l'albero.

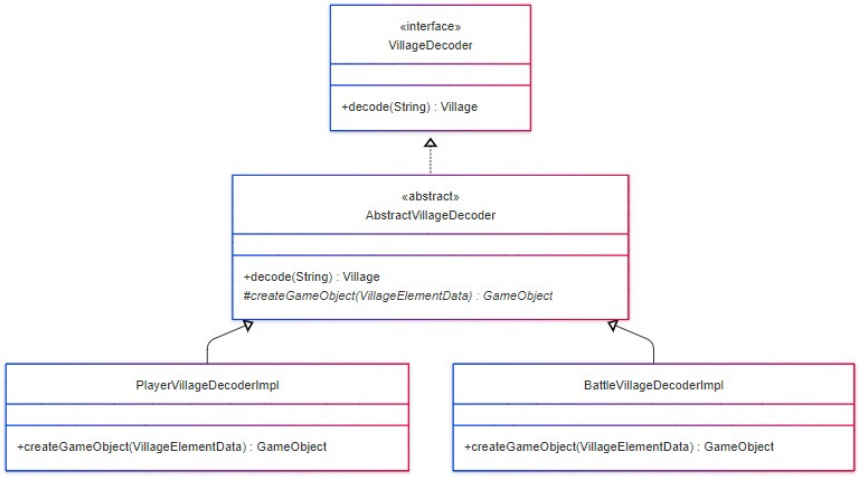

Figura 2.6: Rappresentazione UML del Template Method per usare diverse Factory in contesti diversi

Lucas Antonio Leonte

Sistema di caricamento da file CSV

Problema: Effettuare il parsing da file CSV e ricostruire gli oggetti del villaggio con le opportune istanze dei GameObjects.

Soluzione: Il sistema di decodifica utilizza il `Template Method Pattern`, implementato in `AbstractVillageDecoder`. Il metodo template **decode()** definisce l'intero algoritmo di decodifica del villaggio:

1. Analizza la sezione delle risorse
2. Analizza la sezione delle truppe
3. Analizza la sezione degli edifici
4. Crea oggetti di gioco tramite il metodo astratto `createGameObject()`

Il metodo astratto **createGameObject()** rappresenta il "hook" che le sottoclassi devono implementare.

Questo consente alla classe base di controllare il flusso generale della decodifica, lasciando alle sottoclassi la possibilità di personalizzare la creazione degli edifici in base al contesto (villaggi di battaglia vs villaggi del giocatore).

Modello del Villaggio

Problema: Necessità di una rappresentazione spaziale del villaggio di gioco che possa contenere gli edifici e le relative collissioni.

Soluzione: la classe `Village` funge da aggregate root che incapsula le griglie spaziali, la gestione degli edifici e l'associazione al giocatore.

Sistema di creazione edifici

Problema: creare diversi tipi di istanze di GameObject in base al contesto (gestione del giocatore vs scenari di battaglia), mantenendo al contempo un'interfaccia coerente.

Soluzione: Il sistema delle factory per gli edifici implementa il Design Pattern Abstract Factory tramite la classe `AbstractBuildingFactory`.

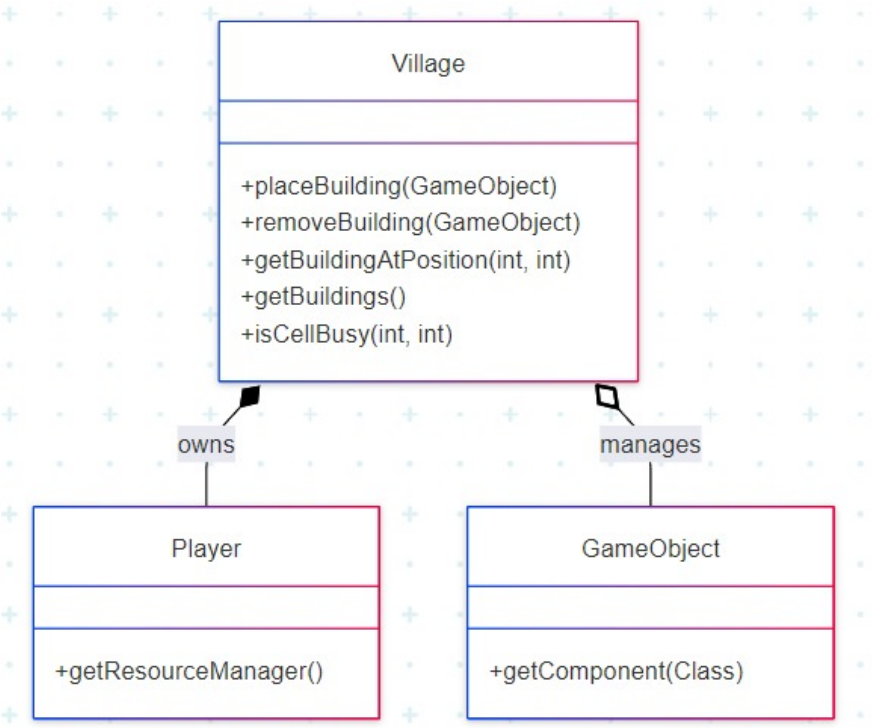

Figura 2.7: Rappresentazione UML del dominio del Villaggio

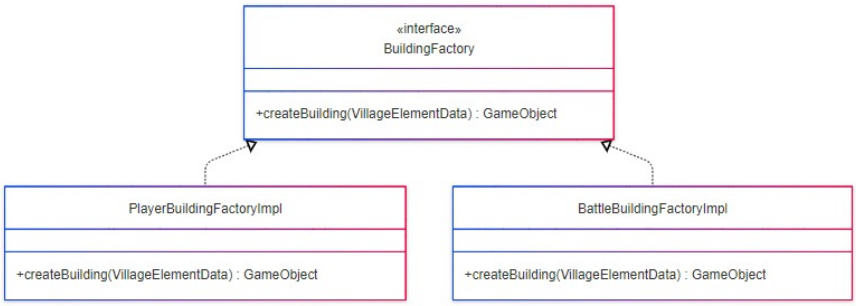

Figura 2.8: Rappresentazione UML del pattern Abstract Factory

- **Interfaccia Abstract Factory**: `BuildingFactory` definisce i metodi per creare tutti i tipi di edifici.
- **Classe Abstract Factory**: `AbstractBuildingFactory` fornisce la logica comune per la creazione degli edifici e i metodi astratti per la specializzazione.
- **Factory Concrete**:
  - `BattleBuildingFactoryImpl`: crea edifici con componenti da battaglia (vita, IA, logica di danno).
  - `PlayerBuildingFactoryImpl`: crea edifici semplici con solo componenti base.

Integrazione delle Building Factory

Problema: i decoder dei villaggi hanno bisogno di cpaire quale Factory utilizzare in base al contesto del villaggio in cui ci si trova.

Soluzione: è stata implementata una classe `BuildingFactoryMapper`, che incapsula le factory concrete, che tramite il metodo **BuildingFactoryMapper.getFactoryFor()**

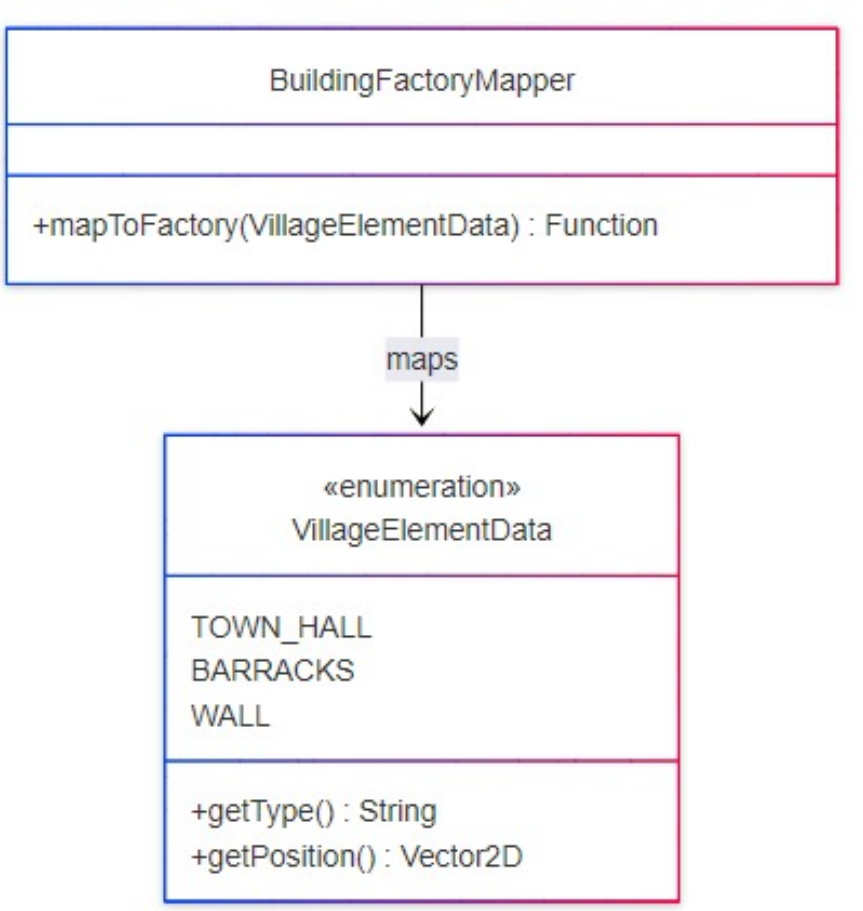

Figura 2.9: Rappresentazione UML del sistema di mappatura alle Factory

restituisce il metodo di creazione appropriato.

Sistema Manager del Villaggio

Problema: Coordinare l'intero processo di salvataggio e caricamento, gestire i diversi contesti e fornire un'interfaccia unificata per la persistenza dei villaggi.

Soluzione: Pattern Facade che orchestra encoder, decoder e operazioni sui file, astraendo la complessità di gestione del villaggio. Quando viene chiamato, **saveVillage()** raccoglie tutte le operazioni di codifica e scrittura su file in un'unica chiamata.

- **Codifica**: passa gli oggetti `GameObject` al `VillageEncoder`, che li converte in una stringa CSV.
- **Scrittura su File**: successivamente, tramite il `FileWriter`, scrive la stringa codificata su un file CSV.
- **Astrazione della complessità**: il client (cioè chi invoca il metodo saveVillage) non deve conoscere questi dettagli, poiché tutte le complessità sono gestite dal manager.

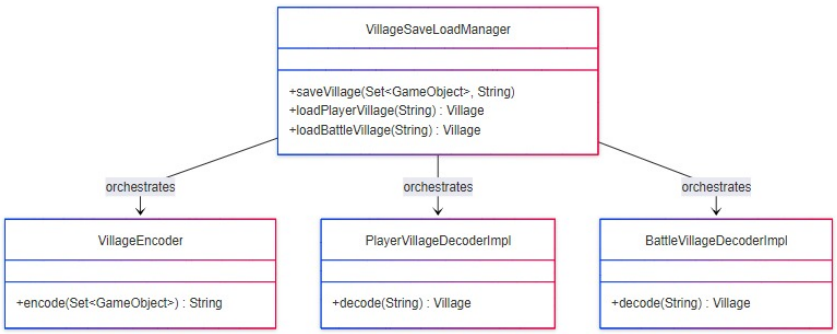

Figura 2.10: Rappresentazione UML del Facade Pattern utilizzato per gestire le principali operazioni del villaggio

Lorenzo Bulfoni

Problema La gestione del timer richiede una soluzione precisa per tracciare la durata del gioco in tempo reale e notificare eventi alla scadenza del tempo limite, garantendo la sincronizzazione con le dinamiche del gioco. I problemi principali individuati inizialmente sono stati

- **Assenza di modularità**: Sistemi dove la gestione del tempo era integrata direttamente con altre logiche, rendendola poco riutilizzabile e difficilmente manutenibile.
- **Problema thread-safe**: In alcuni contesti, il controllo manuale di thread o di intervalli di tempo poteva causare inconsistenze.

Soluzione La soluzione adottata è stata l'implementazione di un **Timer modulare** e dedicato alle dinamiche della partita, rappresentato da un'interfaccia (`Timer`) e dalla relativa classe concreta (`TimerImpl`). La classe `TimerImpl` è completamente separata da qualsiasi altra logica di gioco, rispettando i principi di **Single Responsibility** ,questo permette la gestione del tempo fuori dal game loop . È possibile utilizzarla in diversi scenari senza modifiche ulteriori. **Thread dedicato al timer**: Il timer opera in un thread separato, garantendo che i conteggi temporali non interferiscano con altre elaborazioni del gioco e riducendo al minimo i rischi di blocchi o sovraccarico del main thread. **Funzionalità basiche**:

- **start()**: Avvia il timer in un thread separato.
- **stop()**: Interrompe il timer in esecuzione e chiude il thread.
- **getElapsedTime()**: Ritorna il tempo trascorso

Gestione degli eventi alla scadenza: La classe offre il metodo `onFinished()` che può essere facilmente sovrascritto o integrato con logiche specifiche, come terminare una partita, aggiornare lo stato di gioco o notificare l'utente.

Problema: Gestione degli eventi di distruzione La necessità di gestire situazioni in cui diversi osservatori (come manager delle truppe o condizioni di fine battaglia) devono essere notificati quando un edificio, un villaggio o un altro oggetto viene distrutto. L'implementazione deve: Fornire un meccanismo per notificare eventi di

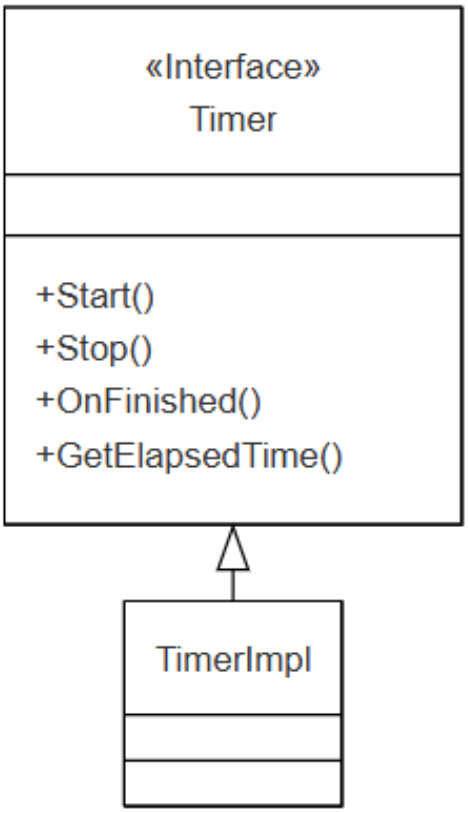

Figura 2.11: Rappresentazione UML

distruzione senza accoppiare direttamente componenti del sistema e supportare la rimozione o l'aggiunta di osservatori, garantendo che ogni oggetto osservabile (ad esempio un edificio) possa notificare una serie di osservatori registrati. Si necessita di gestire una specifica condizione di fine battaglia basata sul tempo, verificando costantemente lo stato di un timer, gestire un'altra condizione di fine battaglia basata sul livello di distruzione totale di un villaggio ossia il completamento del 100% della distruzione degli edifici.Un altro problema e' aggiornare correttamente il comportamento delle truppe in gioco in risposta alla distruzione di edifici/mura(modifiche nella strategia del movimento)

Soluzione: Il pattern Observer e' il fulcro del design. Il pattern separa gli eventi di distruzione dall'applicazione dei loro effetti, garantendo che gli osservatori possano essere aggiunti o rimossi senza modificare il codice del soggetto osservabile (`DestructionObservableImpl`). e i componenti che reagiscono agli eventi di distruzione (ad es. la fine della battaglia o il comportamento delle truppe) sono indipendenti dalle entità specifiche che li generano.

La classe `DestructionObservableImpl` rappresenta un soggetto osservabile con un elenco di osservatori registrati. L'interfaccia `DestructionObserver` definisce un contratto per i componenti che vogliono essere notificati (come `EndBattleAllVillageDestroyed`). Quando un edificio o un oggetto monitorato viene distrutto, il metodo `update()` di `DestructionObservableImpl` notifica tutti gli osservatori. La notifica include l'oggetto distrutto (`GameObject`), permettendo agli osservatori di reagire in modo specifico. L'interfaccia `EndBattleTimerIsOver` estende `DestructionObserver`, permettendo all'implementazione di reagire a eventi di distruzione, come la scadenza

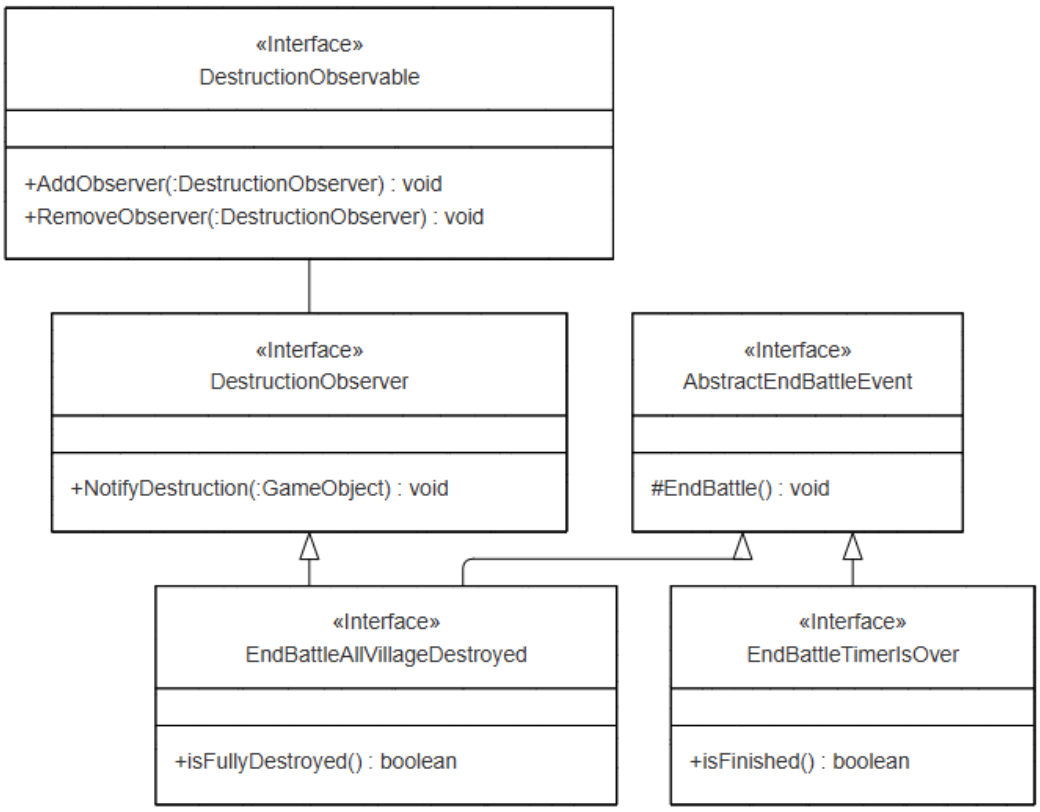

Figura 2.12: Rappresentazione UML

di un timer. Quando la condizione è verificata, l'evento di fine battaglia viene generato attraverso `AbstractBattleEvent`, tramite il metodo `EndBattle()` che contiene la logica di terminazione della battaglia

L'interfaccia `EndBattleAllVillageDestroyed` estende `DestructionObserver`, monitorando eventi di distruzione relativi a edifici o componenti del villaggio. La classe implementativa `EndBattleAllVillageDestroyedImpl` controlla se il villaggio è completamente distrutto utilizzando il metodo `isFullyDestroyed()`

L'interfaccia `BattleTroopsBehaviorManager` estende `DestructionObserver`, permettendo di ricevere notifiche relative agli eventi di distruzione. La classe implementativa `BattleTroopsBehaviorManagerImpl` mantiene un elenco di truppe (`GameObject`) e aggiorna il loro comportamento dopo la distruzione di un edificio, chiamando meccanismi di intelligenza artificiale (ad es. tramite `AiNodesBuilder`). width=0.5.png

L'uso di interfacce (`DestructionObserver`, `DestructionObservable`) consente di aggiungere nuovi tipi di osservatori o osservabili con modifiche minime. Ad esempio: È possibile aggiungere un nuovo osservatore per monitorare altri eventi di distruzione (es. distruzione di risorse)per garantire la massima estensibilità

Problema

Il sistema di battaglia deve gestire dinamicamente la fine della battaglia quando tutte le truppe vengono distrutte. Le principali sfide sono:

1. Monitorare in tempo reale la morte delle truppe.
2. Verificare se tutte le truppe sono state eliminate.
3. Terminare immediatamente la battaglia una volta soddisfatta questa condizione.
4. Garantire indipendenza tra la logica di monitoraggio, verifica e fine battaglia.

Soluzione

La soluzione utilizza il **Observer Pattern** per gestire gli eventi di morte delle truppe e determinare la fine della battaglia, la logica di monitoraggio è separata dalla gestione della fine battaglia, garantendo codice chiaro ed espandibile:

1. `TroopDeathObservable|:` Ogni truppa funge da osservabile e consente agli osservatori di registrarsi ai suoi eventi. Quando la truppa viene distrutta, notifica tutti gli osservatori.
2. `TroopDeathObserver|:` Gli osservatori implementano la logica per reagire agli eventi di morte delle truppe. `EndBattleAllTroopsDead` usa questo sistema per tracciare le truppe distrutte.
3. `EndBattleAllTroopsDead|:`
    - Monitora le truppe distrutte e traccia il numero totale attraverso il metodo `setTroopCount()`.
    - Verifica dinamicamente la condizione di fine battaglia con `isAllTroopsDead()`.
    - Quando tutte le truppe sono distrutte, termina automaticamente la battaglia attivando un evento dedicato.

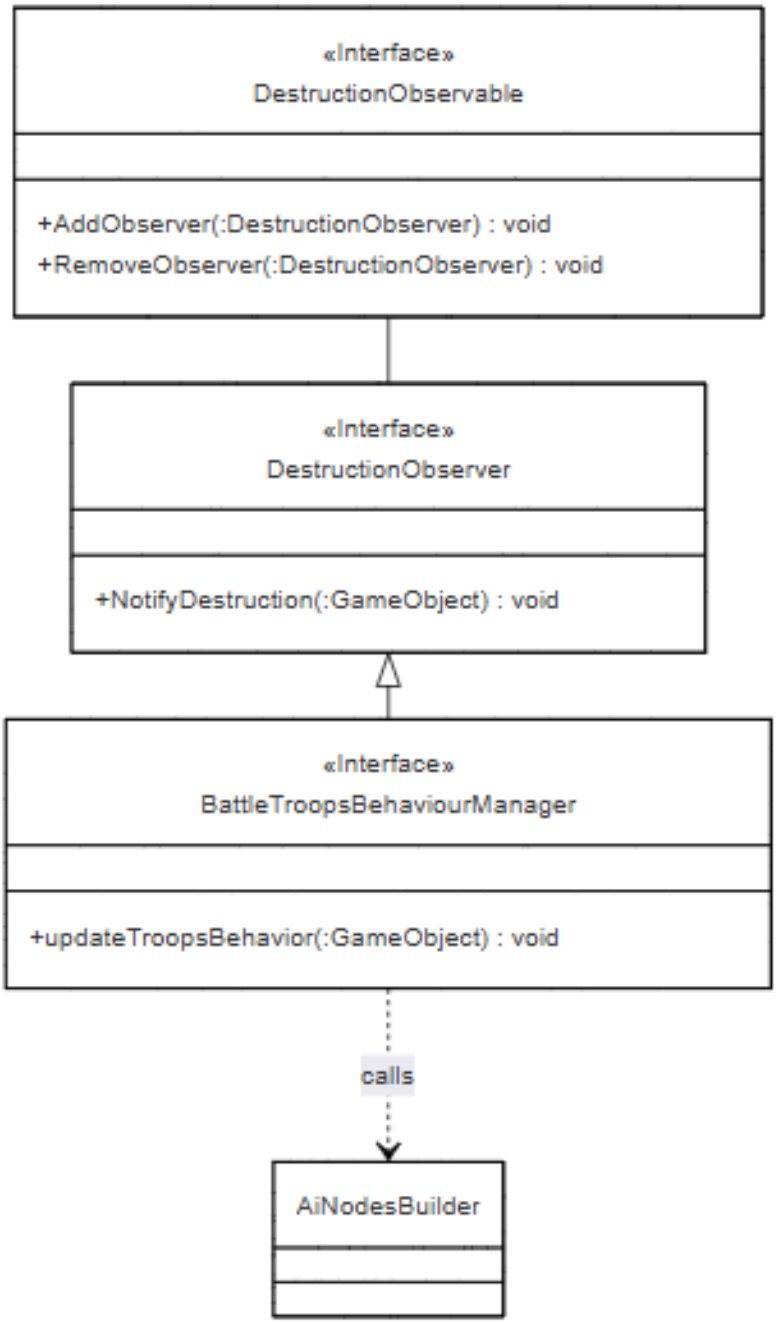

Figura 2.13: Rappresentazione UML

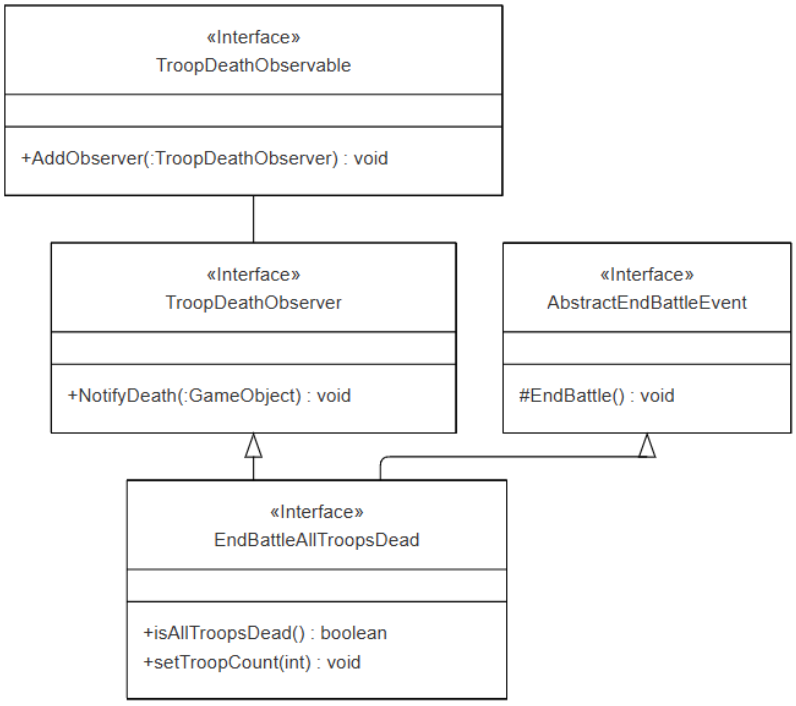

Figura 2.14: Rappresentazione UML

Problema

Il sistema deve gestire dinamicamente gli eventi di distruzione durante una battaglia, includendo:

- Monitorare la distruzione degli edifici di un villaggio aggiornandone lo stato, calcolando la percentuale di distruzione e il numero di stelle guadagnate.
- Registrare e rendere disponibili informazioni chiave sul rapporto di battaglia, come risorse rubate, truppe usate, progresso di distruzione, e risultati (vittoria/sconfitta).
- Aggiornare in tempo reale l'interfaccia utente per garantire che i cambiamenti nel modello siano prontamente riflessi nella visualizzazione.
- Mantenere un design indipendente che consenta facilmente l'estensione delle funzionalità (es. nuovi parametri da monitorare o logiche di battaglia) senza modificare o compromettere gli altri componenti del sistema.

  La soluzione doveva garantire che questi elementi collaborassero senza creare forti dipendenze tra i componenti, migliorando sia la manutenibilità che l'efficienza del sistema complessivo.

Soluzione

La soluzione prevede l'implementazione di un design basato su **Model-View-Controller (MVC)** combinato con l' **Observer Pattern**, che consente di separare e coordinare efficacemente i vari componenti:

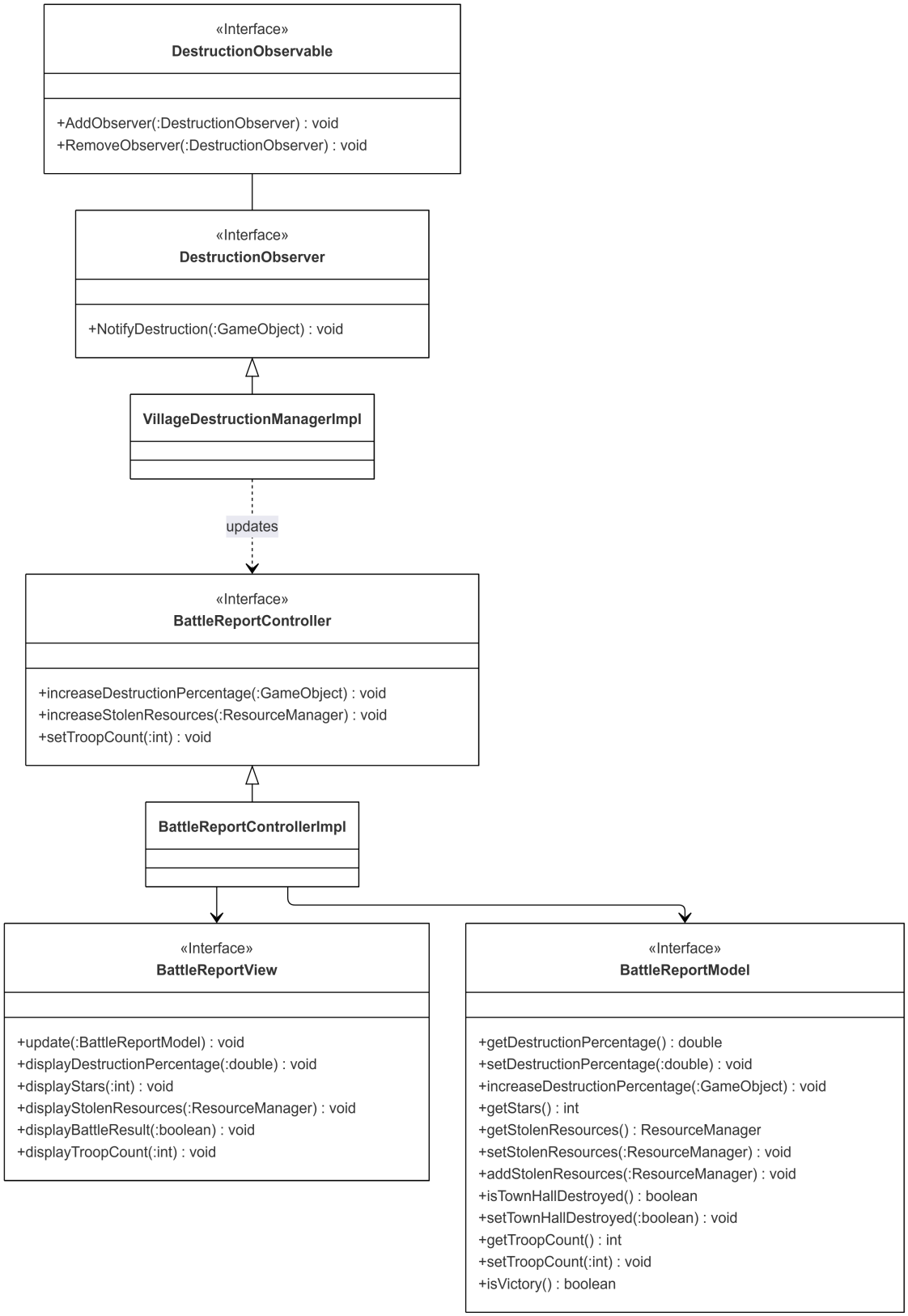

Figura 2.15: Rappresentazione UML

- **Model:** rappresentato dalla classe `BattleReportModelImpl`. Gestisce lo stato del rapporto di battaglia (percentuale di distruzione, stelle guadagnate, risorse rubate, ecc.). È centralizzato e garantisce coerenza nei dati.
- **View:** implementata da `BattleReportViewImpl`, si occupa esclusivamente della presentazione dei dati all'utente. Mostra dettagli come percentuale di distruzione, stelle guadagnate e risorse rubate in tempo reale, senza accedere direttamente alla logica del modello.
- **Controller:** la classe `BattleReportControllerImpl` funge da intermediario tra modello e vista. Coordina gli aggiornamenti, ricevendo i dati dal sistema (ad esempio, notifiche di distruzione dal `VillageDestructionManager`) e aggiornando i dati del modello e la vista di conseguenza.
- **Observer Pattern:** implementato dal `VillageDestructionManagerImpl`, tiene traccia degli eventi di distruzione e notifica il controller, che aggiorna i dati e la visualizzazione.

Grazie a questo design, viene garantito un forte disaccoppiamento tra i componenti, permettendo una facile manutenzione, estensibilità e un aggiornamento in tempo reale che rende il sistema efficiente e scalabile.

Francesco Volpini

Gestione degli item nello shop

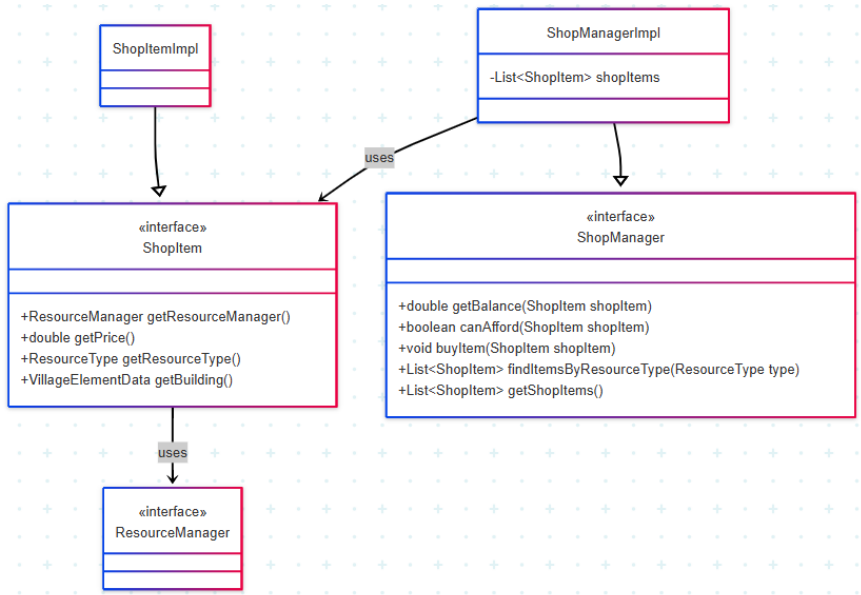

Figura 2.16: Rappresentazione UML del sistema di gestione degli ShopItem

Problema: Lo shop di gioco deve fornire informazioni su tuttui gli oggetti in vendita che differiscono per tipologia, prezzo, tipo di risorsa usata per l'acquisto.

Soluzione: La classe ShopManagerImpl implementa e opera su una lista di ShopItems che contengono in essi tutte le informazioni necessarie.

Gestione di risorse multiple

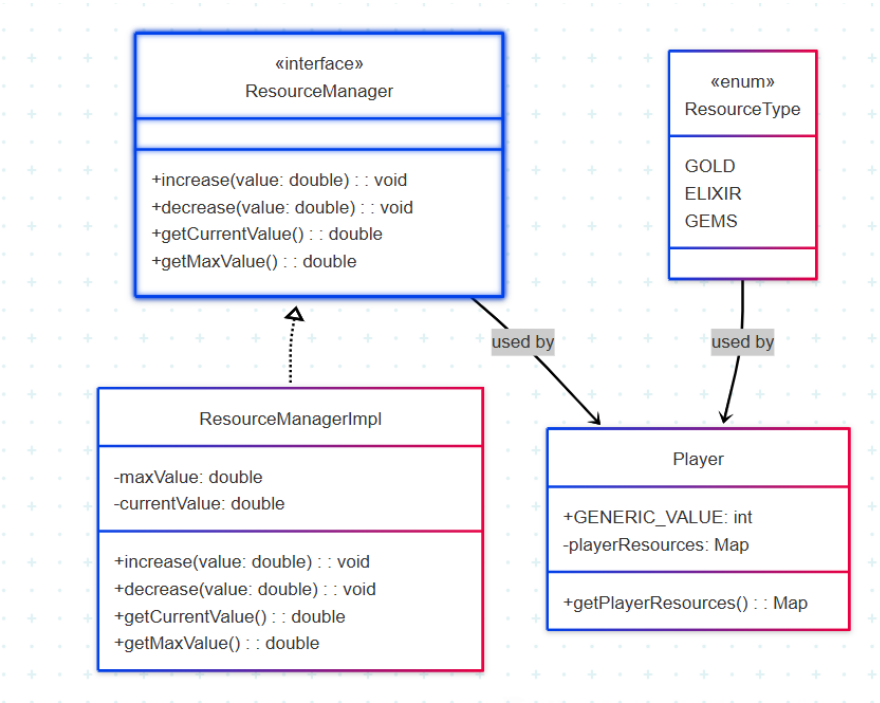

Figura 2.17: Rappresentazione UML del sistema di gestione delle Risorse

Problema: Il gioco dovrà gestire 3 tipi di risorse indipendenti tra loro, su cui eseguire le medesime operazioni.

Soluzione: Il codice nella classe Player usa **enum** per distinguere le risorse e rimane aperto a varianti implementative del ResourceManager grazie al **Pattern Strategy**. È stato notato solo in un secondo momento come la classe Player fosse poco modulabile e rendesse difficile l'aggiunta di potenziali risorse future, ciò è risolvibile con una **Factory**, una soluzione che è stata valutata ma mai implementata a causa delle tempistiche e del numero finito e costante delle risorse necessarie al gioco.

Creazione delle Scene

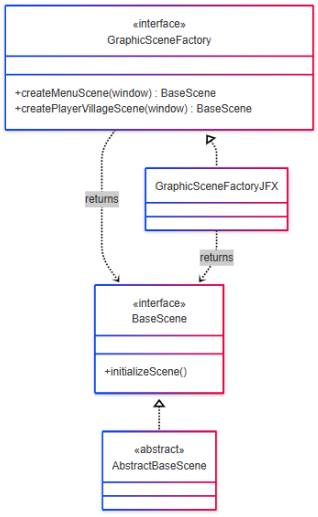

Figura 2.18: Rappresentazione UML del pattern Factory in relazione alla gestione delle Scene

Problema: Il sistema di Scene deve funzionare senza legarsi a una specifica libreria grafica.

Soluzione: La risoluzione del problema avviene attraverso l'impiego di una BaseScene che definisce uno scheletro comune, mentre AbstractBaseScene implementa la logica e le proprietà condivise da tutte le Scene, mentre GraphicSceneFactory astrae la creazione delle Scene che viene implementata tramite una **Factory**.

Creazione delle Window

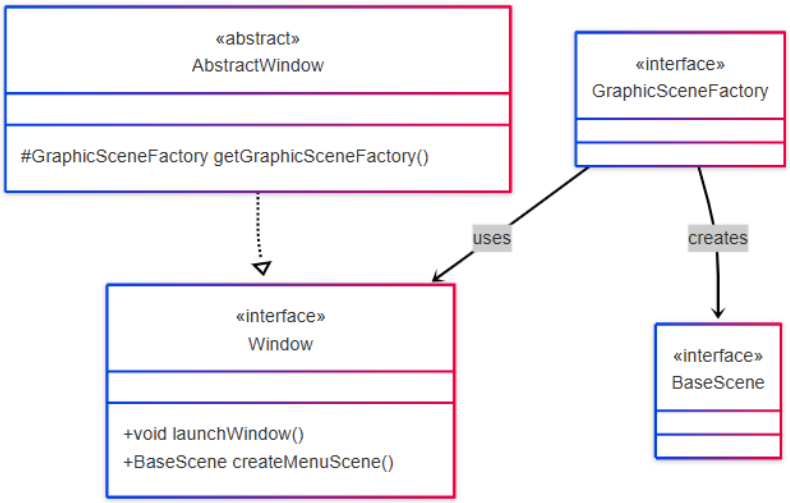

Figura 2.19: Rappresentazione UML del pattern Template Mathod in relazione alla gestione delle Window

Problema: Il sistema di Window deve funzionare senza legarsi a una specifica libreria grafica.

Soluzione: Come per il problema precedente è stato impiegata una classe base Window e una sua astrazione AbstractWindow in accordo col **Pattern Template Method**, lasciando l'implementazione delle eventuali librerie grafiche a classi che estendono quest'ultima.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per lo sviluppo di test abbiamo utilizzato JUnit 5.

È necessario evidenziare che, per problemi relativi al tempo, non è stato possibile correggere i problemi derivanti dai plugin impostati nella build di gradle riguardo ai **soli file contenenti i test**. Nello specifico, sono stati disabilitati:

- Checkstyle Test
- PMD Test
- Spotbugs Test

Sono invece rimaste attive tutte le loro versioni "Main".

I test che abbiamo effettuato sono:

- **GameObjectTest**, controlla il corretto funzionamento dell'Entity-Component System:
  - Testare che un Componente aggiunto al GameObject sia poi ottenibile in un secondo momento
  - Testare che solo il tipo di Componente richiesto venga restituito dalla lista di componenti del GameObject, e, in caso di presenza di due Componenti dello stesso tipo, che solo il primo in ordine di inserimento venga restituito
  - Testare che sia possibile ottenere un Componente attraverso il tipo di una sua interfaccia
  - Testare che il **GameObjectBuilder** crei il GameObject con la lista di componenti corretta e ordinata
- **GameEngineTest**, controlla la corretta gestione del thread secondario utilizzato dal GameLoop
- **TroopFactoryTest**, controlla il corretto funzionamento della struttura a gerarchia di Factory per la creazione delle truppe:
  - Testare che la creazione del GameObject della truppa abbia il comportamento atteso

  - Testare che la truppa creata con un'implementazione specifica della Factory abbia i relativi componenti aggiuntivi

- **ChooseTargetLogicTest**, controlla il corretto funzionamento dell'algoritmo di selezione del prossimo edificio da attaccare per le truppe, in particolare verifica che l'edificio scelto sia quello più vicino alla truppa tra il set di dati fornito

- **PathfindingAlgorithmTest**, controlla il corretto funzionamento dell'algoritmo di ricerca del percorso minimo, fornendo un set di celle con un layout specifico e confrontando il risultato restituito dall'algoritmo con il percorso ottimale precalcolato, rappresentato come una lista ordinata di celle (dove la prima cella è quella dove si trova la truppa, e l'ultima quella dove è posizionato l'edificio)

- **BlackboardTest** controlla che la gestione delle varibili della Blackboard, chiamate "BlackboardProperty", funzioni correttamente, sia in scrittura che in lettura

- **VillageSaveLoadTest**, controlla che i villaggi possano essere caricati da file CSV con le corrette mappature degli elementi.

- **TimerTest**, questa classe verifica il funzionamento del Timer, con attenzione particolare ai seguenti scenari:

  - Testare se il timer parte correttamente.
  - Verificare che il timer si fermi correttamente.
  - Accertarsi che il tempo trascorso aumenti correttamente.
  - Controllare che il timer si comporti correttamente al raggiungimento del limite di tempo impostato.
  - Testare il metodo `onFinished` e verificare che il timer si fermi dopo il completamento.

- **DestructionAndBattleManagementTest**, questa classe contiene diversi test inerenti alla gestione della distruzione e al comportamento in battaglia. In particolare:

  - La corretta aggiunta e rimozione delle truppe nel **BattleTroopsBehaviorManagerImpl**.
  - Testare che un osservatore di tipo **DestructionObserver** riceva notifiche appropriate.
  - Verificare che un'istanza di **EndBattleAllVillageDestroyedImpl** notifichi correttamente lo stato di distruzione completa del villaggio.
  - Assicurarsi che il **timer** associato alla battaglia funzioni correttamente in **EndBattleTimerIsOverImpl**.

- **EndBattleTest**, questa classe verifica la logica di fine battaglia:

  - Controllare che il villaggio non risulti distrutto all'inizio.
  - Verificare che il timer non scada immediatamente all'avvio.

  - Nota: i test più approfonditi sul timer si trovano nella classe **TimerTest**.

- **TroopDeathTest**, questa classe testa il comportamento legato alla morte delle truppe nel manager:
  - Assicurarsi che le truppe vengano aggiunte al **BattleTroopsBehaviorManagerImpl** senza errori.
  - Verificare che una truppa con vita pari a zero venga trattata come morta.
  - Testare che una truppa morta venga rimossa dal manager.
- **BattleReportTest**, questa classe verifica il funzionamento dei report di battaglia:
  - Controllare che l'incremento della percentuale di distruzione funzioni correttamente.
  - Testare l'aumento delle risorse rubate durante la battaglia.
  - Assicurarsi che il conteggio delle truppe utilizzate in battaglia venga registrato correttamente.
  - Verificare che il TownHall distrutto notifichi lo stato corretto.
  - Testare il calcolo delle stelle ottenute in base alla percentuale di distruzione e allo stato del TownHall.
  - Controllare se viene determinato correttamente lo stato di vittoria o sconfitta.

3.2 Note di sviluppo

Alessandro Ricci

In autonomia mi sono occupato di:

- **Creazione del Game Engine**, con conseguente creazione del GameLoop su di un altro thread e gestione controllata delle operazioni non-thread-safe
- **Creazione dell'Entity-Component System**, rendendo possibile la creazione delle entità in gioco mediamente composizione
- **Creazione del Game State Manager**, con l'implementazione dei due "stati", equivalenti alle due scene presenti nel gioco, "player village" e "battle"
- **Creazione delle Factory per truppe ed edifici**, secondo la struttura a gerarchia descritta in precedenza
- **Creazione del sistema di ricerca del percorso minimo**, con implementazione del noto algoritmo "A*" e degli algoritmi di scelta del prossimo bersaglio e calcola della stima della distanza tra due nodi
- **Creazione di un'implementazione personalizzata del noto "BehaviourTree"**, per la gestione dell'intelligenza artificiale delle entità in gioco, con inclusione del sistema di "Blackboard" per le variabili modificabili dai nodi interni

- **Creazione del sistema di statistiche di truppe ed edifici**
- **Creazione dei Componenti principali per le entità in gioco**, come l'HealthComponent, il GridTileData2D, i componenti per le statistiche, ecc...
- **Creazione della parte grafica dell'applicazione**, con implementazione in JavaFX

Sviluppi avanzati:

- **Uso massivo di Stream**: ove possibile, sono state utilizzate le Stream per effettuare query complesse e favorire un approccio funzionale. Esempio: https://github.com/AleRicci26/OOP24-clashclass/blob/e72b5ad4787b167cf1c279eb27
src/main/java/clashclass/ai/pathfinding/PathNodeGridImpl.java#L68
- **Creazione di interfacce funzionali**: per meglio rappresentare, ad esempio, l'astrazione dei concetti di algoritmi particolari, come quello di calcolo della distanza tra due punti nello spazio: https://github.com/AleRicci26/OOP24-clashclass/blob/e72b5ad4787b167cf1c279eb272ef59ea484f4b0/src/main/java/clashclass/ai/pathfinding/DistanceHeuristic.java#L9
- **Utilizzo di Lambda e Method References**: per passare funzioni temporanee come parametro (lambda) e per rendere più compatte alcune query effettuate con le Stream (method references). Esempio: https://github.com/AleRicci26/OOP24-clashclass/blob/431c4b3a85b55c1e982d7d38f8843416a9602094/src/main/java/clashclass/view/graphic/GraphicJavaFXImpl.java#L96
- **Utilizzo dei Generics**: usati, ad esempio, per permette ad una `BlackboardProperty` di contenere variabili di tipo generico: https://github.com/AleRicci26/OOP24-clashclass/blob/e72b5ad4787b167cf1c279eb272ef59ea484f4b0/src/main/java/clashclass/ai/behaviourtree/blackboard/BlackboardProperty.java#L8

Lorenzo Bulfoni

In autonomia mi sono occupato di:

- **Gestione degli eventi di battaglia** (`clashclass.battle`): Implementazione delle differenti condizioni che determinano la fine della battaglia, come la distruzione totale del villaggio, la morte di tutte le truppe e la scadenza del tempo.
- **Timer** (`clashclass.battle.timer`): Creazione di un sistema asincrono che gestisce la durata della battaglia, con funzionalità per monitorare e notificare la scadenza del tempo.
- **Gestione delle truppe** (`clashclass.battle.troopdeath`): Monitoraggio del comportamento delle truppe in battaglia e gestione della loro morte, con un sistema di notifiche e aggiornamenti.
- **Osservatori e notifiche** (`clashclass.battle.destruction`): Implementazione di un sistema per il monitoraggio e la notifica degli eventi di distruzione legati agli edifici e altre strutture.

- **Report di battaglia** (`clashclass.battle.battlereport`): Gestione dei dati relativi alla battaglia, tra cui la percentuale di distruzione, le stelle guadagnate e le risorse rubate, con aggiornamento dinamico delle informazioni.
- **Distruzione del villaggio** (`clashclass.battle.battlereport`): Sistema per monitorare e aggiornare la percentuale di distruzione degli edifici durante le battaglie.

Francesco Volpini

Sviluppi avanzati:

- **Uso di Stream**: https://github.com/AleRicci26/OOP24-clashclass/blob/1abb028dfba0404ca6e08e003db8547c07497baa/src/main/java/clashclass/shop/ShopManagerImpl.java#L54)
- **Definizione e utilizzo di interfacce funzionali**:https://github.com/AleRicci26/OOP24-clashclass/blob/1abb028dfba0404ca6e08e003db8547c07497baa/src/main/java/clashclass/shop/ShopManagerImpl.java#L54)

Lucas Antonio Leonte

Utilizo di Bounded Types, un esempio di progettazione con generici e è presente nella classe `BuildingFactoryMapper` https://github.com/AleRicci26/OOP24-clashclass/blob/4c59a9d61cb1821cb680b06b3c9b7601850b3622/src/main/java/clashclass/elements/buildings/BuildingFactoryMapper.java#L17)

Utilizzo di Method Reference, sempre nela classe `BuildingFactoryMapper` ho ampiamente sfruttato la potenzialità di non dover scrivere lambda expressions per ogni tipo di elemento, usando semplicemente dei referene methods https://github.com/AleRicci26/OOP24-clashclass/blob/4c59a9d61cb1821cb680b06b3c9b7601850b3622/src/main/java/clashclass/elements/buildings/BuildingFactoryMapper.java#L34C8-L43C87

Optional + Lambda expression L'uso della seguente tecnica è stata apllicata anche in altre classi. Il seguente è solo un singolo esempio per mostrare l'eleganza di aver adottato `Optional` per gestire i `null` in modo esplicito e più sicuro, con l'ausilio di una `espressione lambda`, permettendo di lanciare un'eccezione accettando una lambda per generare dinamicamente l'eccezione.https://github.com/AleRicci26/OOP24-clashclass/blob/4c59a9d61cb1821cb680b06b3c9b7601850b3622/src/main/java/clashclass/elements/buildings/BuildingFactoryMapper.java#L53C5-L56C60

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Alessandro Ricci

Mi ritengo, in generale, piuttosto soddisfatto del lavoro svolto e del risultato ottenuto, nonostante i vari ostacoli incontrati durante il percorso.
Mi sono occupato principalmente del core-engine del gioco, del sistema Entity-Component, delle factory e dell'AI di truppe ed edifici, dell'algoritmo di ricerca del percorso minimo, e della creazione della parte visuale (maggiori dettagli sono stati già forniti nel capitolo "Note di sviluppo" precedente). A fronte dei miei sviluppi, è stato comunque necessario che io continuassi a procedere con ulteriori sviluppi al fine di finalizzare il progetto, principalmente per ragioni di tempo e dimensione del progetto stesso, attraverso la creazione di un branch di "emergenza" denominato "alericci/feat/project-finalization".
Col senno di poi, ritengo che, tra i miei sviluppi concordati in origine, la parte riguardante la gestione dell'AI attraverso l'implementazione del `BehaviourTree` sia stata sicuramente soddisfacente, ma esagerata per lo "scope" del progetto.
Ritengo infine che, nonostante tutto, il progetto abbia raggiunto un livello di qualità piuttosto notevole (inteso a livello funzionale) se contestualizzato al tempo e alle condizioni in cui è stato sviluppato.

Lucas Antonio Leonte

Mi ritengo soddisfatto del lavoro svolto, è stato un progetto altamente coinvoltente e stimolante da sviluppare. Mi sono occupato principalmente dello sviluppo del modulo di salvataggio, e grazie anche al contributo proattivo di Alessandro Ricci sono riuscito a capire dettagliamente molti aspetti implementativi del Game Development che fino ad ora mi erano ignoti. Mi sarebbe piacuto raggiungere tutti i requisiti che ci eravamo proposti in modo da poter presentare l'applicazione così come si era immaginata. Con le consocenze attuali ritengo di poter affrontare meglio in futuro un progetto di questa portata ed importanza. Sicuramente ritornerò a lavorare sul progetto per continuare ad affinare e mie conoscenze e capacità di sviluppo in Java.

Lorenzo Bulfoni

Sono abbastanza soddisfatto del lavoro svolto sul pacchetto battle in questo progetto. All'inizio ho avuto bisogno di un po' di tempo per comprendere le logiche

del sistema e il contesto generale, ma una volta chiariti i concetti principali, sono riuscito a lavorare in modo efficace e a raggiungere gli obiettivi che mi ero prefissato. Lavorare su aspetti come la gestione della battaglia, il comportamento delle truppe e il sistema di report mi ha permesso di approfondire concetti importanti della programmazione orientata agli oggetti, come la modularità e la riusabilità del codice. Nonostante alcune difficoltà iniziali, sono riuscito a implementare soluzioni funzionali. Ritengo che il progetto sia stato un'ottima opportunità per migliorare le mie capacità tecniche e organizzative. In futuro, mi piacerebbe lavorare su progetti simili per consolidare le competenze acquisite e migliorare ulteriormente nell'uso di strumenti collaborativi come Git. Nonostante le sfide iniziali, sono contento di quanto ho realizzato e considero questo progetto un ottimo punto di partenza per lavori futuri.

Francesco Volpini

Mi considero soddisfatto del risultato finale del progetto, anche in relazione alle tempistiche a disposizione. Sono grato di aver lavorato all'interno di questo gruppo, il continuo confronto con i miei colleghi di progetto è stata un'esperienza preziosa ed arricchente. Non ostante dubito che questo progetto possa avere altro valore se non quello didattico lascio aperta la possibilità di tornare a lavorarci in futuro per poterlo ampliare e limare alcune ruvidità scaturite dalle fasi più iniziali dello sviluppo come esercizio personale.

4.2 Difficoltà incontrate e commenti per i docenti

Lucas Antonio Leonte

Soggettivamente ritengo di aver avuto alcune difficoltà a gestire efficacemente il coordinamento e avanzamento degli sviluppi del progetto con gli altri componenti del gruppo principalmente per mia mancanza di esperienza e conoscenza di come lavorare in gruppo, sopratutto per progetti di rilevante importanza come il medesimo. Farò tesoro delle difficoltà affrontate per non ripetere gli stessi errori e garantire anche a futuri colleghi un coordinamento più fluido. Consiglierei ai docenti di proporre durante il corso lo sviluppo di una mini-applicazione come una esercitazione di laboratorio per permettere agli studenti di simulare il coordinamento di gruppo.

Francesco Volpini

Personalmente la sfida più grande è stato il primo impatto col carico di lavoro che un progetto del genere porta con se, anche in relazione alla mia poca dimestichezza. È dovuto passare del tempo prima che ottenessi un ritmo propedeutico a uno sviluppo efficace dell'applicazione, e osservando tutto in retrospettiva credo che i primi segmenti che ho programmato siano quelli che ne abbiano risentito maggiormente. Oltre a ciò per motivi di tempo, sono stato a malincuore costretto ad abbandonare alcune parti per concentrarmi sul reparto grafico, operando così delle rinunce.

Capitolo 5

Guida utente

All'avvio del programma, dopo il caricamento del villaggio utente, sarà possibile interagire con due pulsanti:

- **Shop**: aprirà l'interfaccia utente dedicata al negozio, in cui è possibile acquistare nuovi edifici
- **Battle**: farà entrare il gioco in modalità "battaglia", da cui seguono i prossimi comandi

In modalità "battaglia", verranno visualizzate, in basso, le truppe possedute.
L'unico input possibile sarà il click del mouse, prima per selezionare un tipo di truppa (barbari o arcieri), poi sul villaggio per posizionare il tipo di truppa selezionata (ad ogni click corrisponde la creazione di un'istanza di truppa). Sarà possibile posizionare le truppe solo in celle appartenenti al villaggio (ad eccezione quindi della parte esterna più scura) e solo in celle che non siano già occupate da edifici.

Finita la battaglia, si aprirà un report con le statistiche relative alla battaglia stessa, e, tramite la pressione del pulsante "Go Back to Village", sarà possibile tornare al proprio villaggio, ricominciando dal punto di partenza.

Nota: per problemi relativi al tempo, non è stato possibile implementare, nello shop, il posizionamento dell'edificio appena comprato nel villaggio. Tuttavia,la logica di acquisto funziona comunque, così come il salvataggio delle risorse rimaste su file.

Importante: il gioco tenterà di leggere i dati del villaggio dell'utente da un file con percorso "/Villages-Data/player-village.csv", relativo alla stessa cartella in cui si trova il fat-jar dell'applicazione.
Se questo file non è presente, il gioco caricherà inizialmente un villaggio di default dalle risorse del progetto, per poi scrivere salvare lo stesso file su disco appena possibile.
Il programma deve quindi avere i permessi di scrittura di file su disco per comportarsi nel modo atteso, altrimenti caricherà sempre il villaggio di default dalle risorse.
In caso le truppe possedute finiscano (perché le si è schierate tutte in battaglia), e non sia quindi più possibile effettuare una battaglia, è necessario modificare manualmente il numero di truppe possedute nel file di salvataggio, impostato i valori di "BARBARIAN" e "ARCHER" (settima e ottava riga del file) ad un numero superiore a zero.