

Relazione

“Tank23”

Federico Diotallevi

Gioele Santi

Giovanni Pisoni

Javid Ameri

10 giugno 2023

Indice

Capitolo 1	3
1.1 Requisiti.....	3
1.2 Analisi e modello del dominio	4
Capitolo 2	6
2.1 Architettura	6
2.2 Design dettagliato	8
Capitolo 3	22
3.1 Testing automatizzato.....	22
3.2 Metodologia di lavoro.....	22
3.3 Note di sviluppo	24
Capitolo 4	27
4.1 Autovalutazione e lavori futuri	28
Guida utente.....	30

Capitolo 1

Analisi

1.1 Requisiti

Il gruppo si pone l'obiettivo di realizzare un videogioco di nome **Tank23** di tipo sparatutto, ispirato al celebre gioco "Battle City". Il carro armato comandato dal giocatore si trova inizialmente alla base della mappa, accanto ad una statua che dovrà proteggere dagli attacchi nemici. L'obiettivo è sopravvivere distruggendo tutti i nemici delle varie ondate prima che questi riescano a colpire la propria torre.

Requisiti funzionali

Requisiti funzionali obbligatori

- Tank23 si occuperà di offrire al giocatore una mappa, costituita da vari ostacoli ed una torre da difendere, che il giocatore potrà percorrere liberamente.
 - Gli ostacoli dovranno essere di almeno due tipi: muri distruttibili e non.
- Il giocatore dovrà affrontare diverse ondate, a difficoltà incrementale, di nemici di vario tipo, alcuni si muoveranno in modo casuale, altri seguiranno il giocatore e gli ultimi andranno diretti a distruggere la torre da difendere.
 - Il movimento dei nemici sarà gestito da semplici AI (intelligenze artificiali). Per AI si intende una parte del software capace di analizzare la mappa e individuare il percorso tra due punti.
 - Il gioco dovrà occuparsi di presentare al giocatore le ondate di nemici, che compariranno in tre diversi punti collocati nel lato superiore della scena di gioco (angoli alti sinistro e destro e centro del lato).
 - La difficoltà incrementale è data dall'aumentare del numero di entità totali o di entità "più intelligenti", cioè quelle che seguono precisi obiettivi.
- Il giocatore dispone di tre vite, ogni vita può essere persa durante il duello con un nemico tramite spari o collisioni tra i due carri armati. Il gioco dovrà essere in grado di mostrare le vite rimanenti durante tutto il corso della partita.

Requisiti funzionali opzionali

- Il gioco offrirà al giocatore bonus o power-up per completare le ondate in modo più veloce e facile.
- Verrà sviluppata una modalità di multiplayer locale. Per multiplayer locale si intende una modalità di gioco in cui più giocatori possono partecipare contemporaneamente ad una partita nella stessa macchina.
- Saranno aggiunti nuovi ostacoli alla mappa di gioco aumentandone la diversificazione del tipo e degli eventi che questi generano (per esempio pozze d'acqua attraversabili dai proiettili, ma non dai carri armati, oppure cespugli che nascondono i giocatori alla visione delle AI).

Requisiti non funzionali

- L'applicazione deve consentire un'esperienza di gioco fluida e priva di errori, in modo da rendere ogni partita piacevole da giocare.
- Tank23 si presenterà con uno stile grafico che ricordi un classico videogioco degli anni '90, pur rimanendo al passo coi tempi.
- L'applicazione dovrà cercare di adattarsi al meglio ad ogni display per consentire al giocatore di avere una corretta visione dell'intera mappa e delle informazioni di gioco.

1.2 Analisi e modello del dominio

Tank23 è il classico gioco arcade in cui il giocatore comanda un carro armato col quale si immerge in un mondo in cui dovrà scontrarsi con altri carri armati. Il giocatore potrà muovere il proprio carro armato in ognuna delle quattro direzioni principali (nord, sud, est, ovest) e sparare proiettili per neutralizzare i nemici o distruggere i muri della mappa. Il carro armato del giocatore verrà generato alla base della mappa, accanto ad una torre che dovrà difendere al fine di continuare la partita. Tank23 è responsabile della creazione e dell'inserimento nel mondo di vari nemici. Questi ultimi saranno comandati da AI di diverso tipo, alcune che muovono il nemico in modo casuale nella mappa e altre che seguiranno precisi obiettivi, come ad esempio il giocatore o la torre da difendere. Nel mondo di Tank23 sono inoltre presenti ostacoli di diverso genere, che bloccheranno la libera navigazione della mappa da parte del giocatore e coi quali potrà interagire distruggendoli, se possibile. Il gioco si svolge per tutta la durata della partita sulla stessa mappa, nella quale hanno luogo diversi round, per ogni round è predisposto un preciso numero di nemici che possono generarsi in tre diversi punti (sinistra, centro, destra) sulla parte alta dello schermo. Una volta che tutti i nemici di un round sono stati neutralizzati ne avrà inizio un'altro più difficile, con un numero di nemici sempre superiore al precedente. Una volta che il giocatore ha perso tutte le vite o la torre è stata distrutta la partita termina, mostrando una schermata che offrirà la possibilità di uscire dall'applicazione o iniziare una nuova partita.

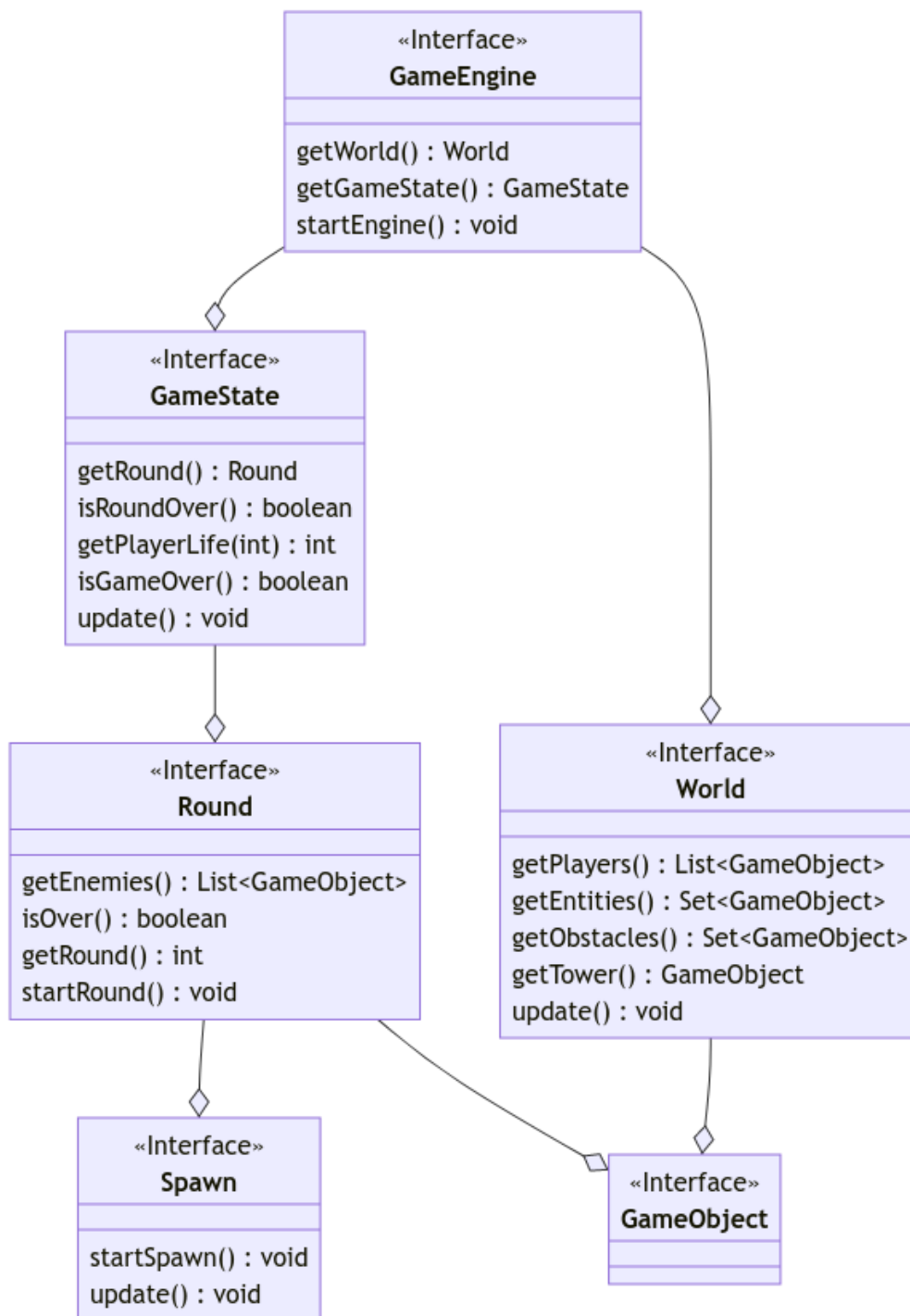


Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti tra loro.

Capitolo 2

Design

2.1 Architettura

Tank23 è stato sviluppato seguendo la struttura del pattern MVC e del pattern Component System. Il pattern Component offre multipli vantaggi, per esempio evitare la creazione di gerarchie di entità piuttosto complesse, favorendo piuttosto la loro composizione e ridurre al minimo la ripetizione di codice. Utilizzando i component ogni entità funge essenzialmente da “contenitore” di componenti, che gli permettono di estendersi su diversi domini senza doverli obbligatoriamente racchiudere in un’unica classe. Combinando i due pattern otterremo che ogni GameObject (parte del Model) sarà corredato di diversi componenti, appartenenti a domini differenti e specifici (come ad esempio la Fisica, le Collisioni, l’Input ecc.) e responsabili del loro aggiornamento. Per comunicare tra loro i componenti possono utilizzare il contenitore comune (cioè il GameObject) e comunicare tramite un sistema di comunicazione indiretta e diretto. Il sistema di comunicazione è indiretto se i componenti utilizzano parti del GameObject che sono già state modificate da altri componenti (per esempio il componente della Fisica aggiornerà la posizione basandosi sulla direzione modificata dal componente dell’Input senza comunicare direttamente con questo). Per far comunicare in modo diretto i componenti è stato invece previsto un semplice sistema di messaggistica che utilizza il GameObject come intermediario.

Per quanto riguarda il pattern MVC:

- **Model:** Il ruolo di Model di Tank23 è assunto da un insieme di classi che comprendono GameState, World, Round, GameObject, Spawn e Component con le loro implementazioni ed estensioni. E’ importante notare che se si volesse sviluppare un nuovo videogioco Tank23 sarebbe una buona base di partenza, infatti sarebbe sufficiente fare qualche modifica alle suddette classi per cambiare totalmente la logica, lasciando di fatto il gioco funzionante.
- **View:** Il ruolo di View è totalmente gestito dalle classi GameView e RenderingEngine. GameView funge da finestra di gioco, capace di settare diversi scenari (title menu, game over, game scene). Il RenderingEngine è invece il responsabile dell’effettiva renderizzazione del pannello di gioco, contiene dentro di sé tutte le informazioni sui componenti grafici di ogni entità.
- **Controller:** Il ruolo di controller è affidato alla classe GameEngine che si occupa di gestire l’aggiornamento delle classi del Model e della conseguente renderizzazione della parte di View.

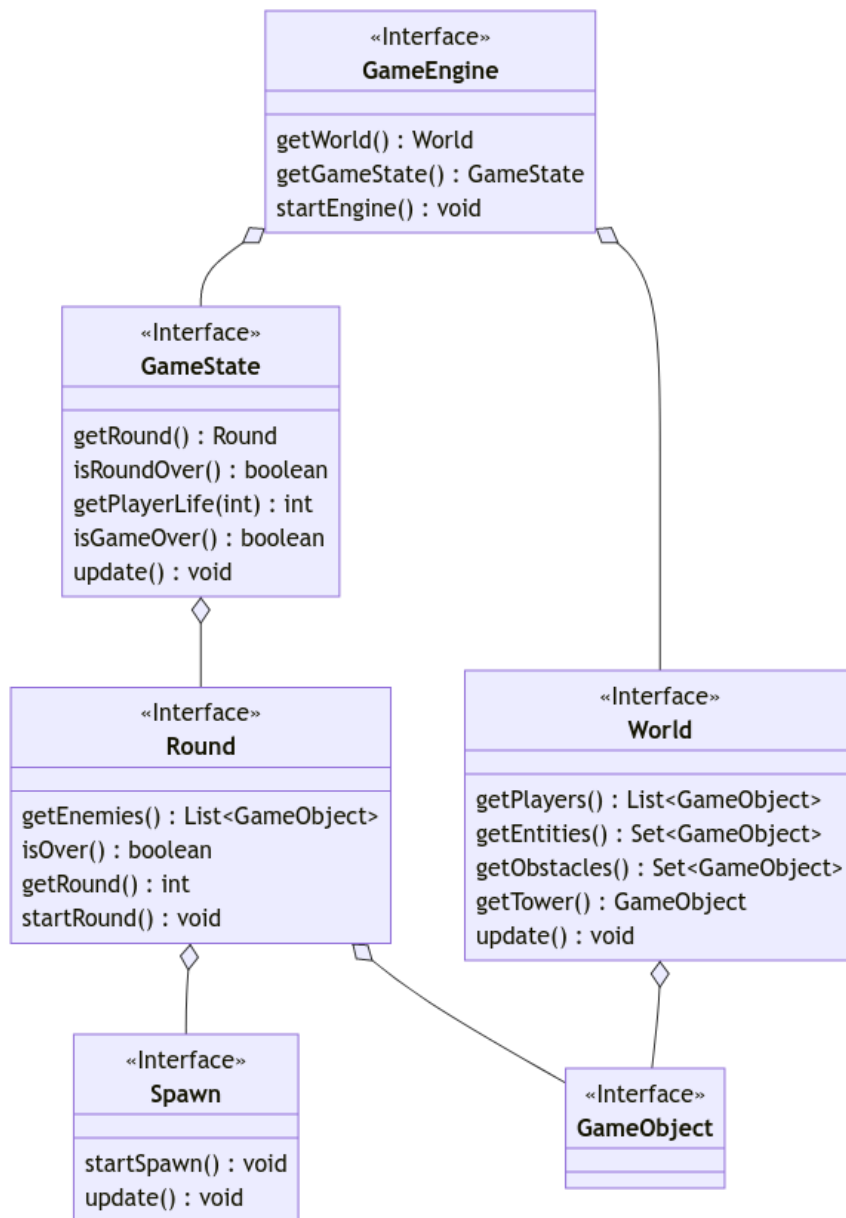


Figura 2.1 design architetturale.

Dallo schema UML fornito in figura 2.1 è possibile notare in particolare la presenza di un componente grafico (GraphicComponent) collegato al model. Tale componente non crea problemi nella struttura del MVC, in quanto tale componente contiene solo il nome dello sprite scelto per la sua renderizzazione. Perciò qualora si decidesse in futuro di passare, per esempio, da JavaFx a Swing, sarebbe sufficiente cambiare le implementazioni delle interfacce GameView e RenderingEngine.

2.2 Design dettagliato

Federico Diotallevi

Separazione di GameEngine e GameLoop

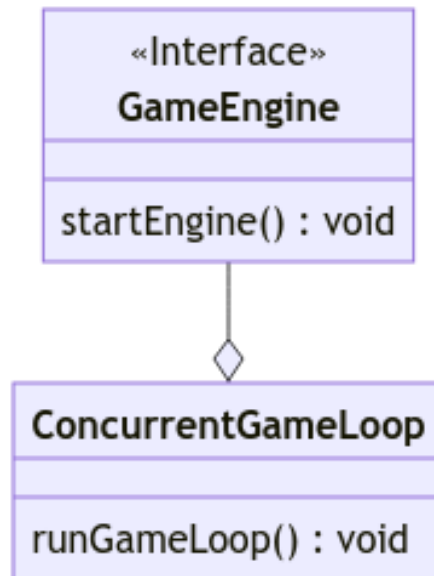


Figura 2.2 rappresentazione UML del pattern *Strategy* per il GameEngine

Problema:

Disaccoppiare la logica del *GameEngine* da quella del *GameLoop*, così da garantire il *Single Responsibility Principle* e rendere il codice della classe *GameEngine* più semplice. Separare i due domini permetterebbe il corretto funzionamento e facile estensione del *GameLoop* che rappresenta uno degli aspetti più importanti di un videogioco.

Soluzione:

Il sistema del *GameEngine* utilizza il pattern *Strategy* per delegare la gestione del game loop alla classe *ConcurrentGameLoop* progettata per quell'unico scopo.

Decorazione del GameLoop

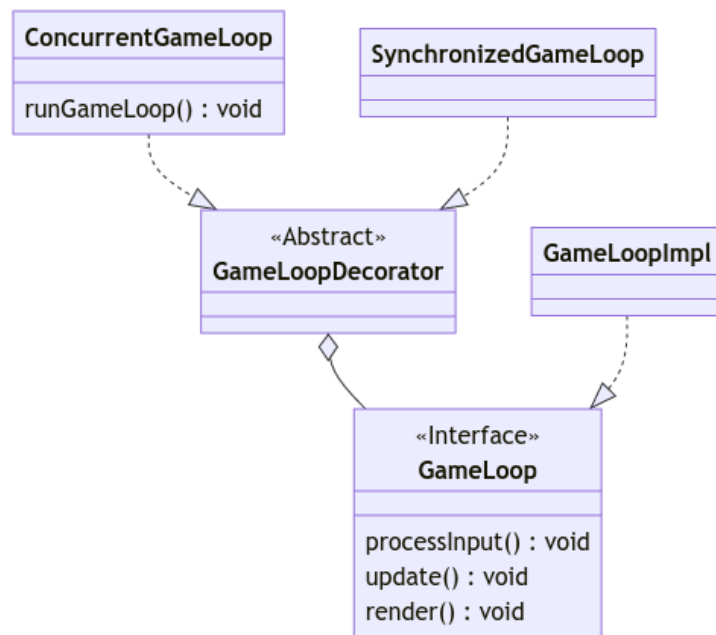


Figura 2.3 rappresentazione del pattern *Decorator* per il *GameLoop*

Problema: La gestione del *GameLoop* risulta essere un aspetto piuttosto complesso in un videogioco. In fase di sviluppo ci si è accorti che l'implementazione del *GameLoop*, pur svolgendo le medesime operazioni di base, poteva essere implementata in svariati modi, a seconda delle esigenze, che potevano risultare piuttosto complessi e che in futuro si sarebbe potenzialmente potuto decidere di cambiare. Per l'aggiunta di nuove feature cambiare l'implementazione di base del *GameLoop* sarebbe potuto risultare difficile e incline ad errori.

Soluzione: Il sistema del *GameLoop* utilizza il pattern *Decorator* per ridurre al minimo la ripetizione di codice e semplificare l'estensione del *GameLoop*. Tramite il pattern *Decorator* risulta molto più semplice andare ad aggiungere nuove feature al game loop, potendo utilizzare le implementazioni di base fornite dal game loop di base. Per la futura estensione di un *GameLoop* si potrà semplicemente aggiungere un nuovo decoratore e comporli in base alle esigenze del gioco.

Costruzione di un grafo per lo sviluppo dell'ai

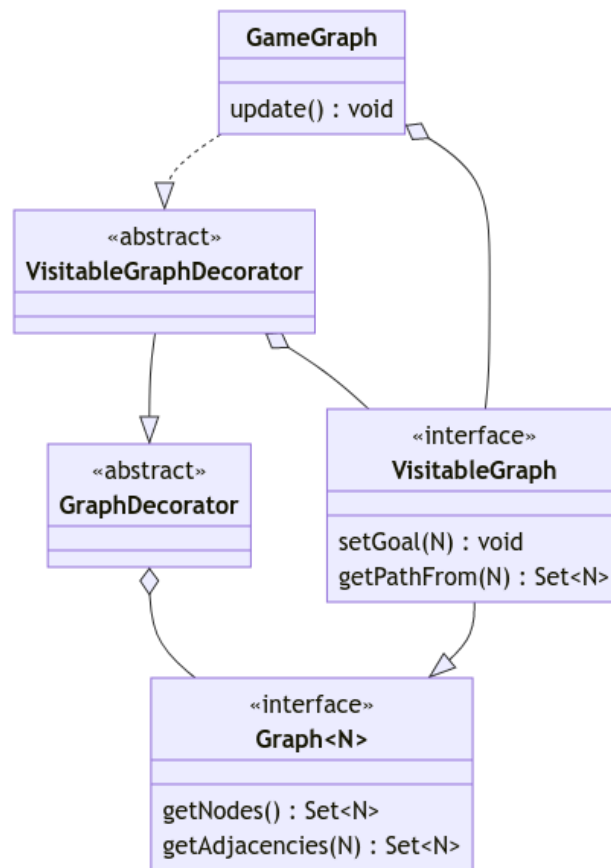


Figura 2.4 rappresentazione del pattern *Decorator* per *Graph*

Problema: La costruzione di un grafo che rappresenti una griglia di gioco può risultare un aspetto per niente banale nella costruzione di un videogioco. In particolare il grafo avrebbe dovuto rappresentare una griglia ed essere visitabile, oltre a contenere funzioni per la collocazione degli ostacoli del mondo di gioco. Una struttura di questo tipo risulta subito essere, già da una prima descrizione, piuttosto confusa e tendente alla generazione di errori molto difficili da individuare.

Soluzione: La soluzione più naturale è stata quella di separare i concetti di *Graph*, *VisitableGraph* e *GameGraph*. Per farlo è stato utilizzato, come nel caso del game loop, il pattern *Decorator*. Utilizzare questo pattern rende molto più facile e comprensibile la struttura di un grafo con le caratteristiche elencate, oltre ad ottimizzare il riutilizzo di codice. In particolare il *GameGraph* è un decoratore di un *VisitableGraph* che aggiunge alcune funzioni utili per la gestione degli ostacoli nel gioco. Tale struttura risulta molto utile nella fase di costruzione del grafo, poiché partendo da un'implementazione di base si possono evitare errori nascosti testando le diverse implementazioni.

Costruzione di View indipendente dal Model

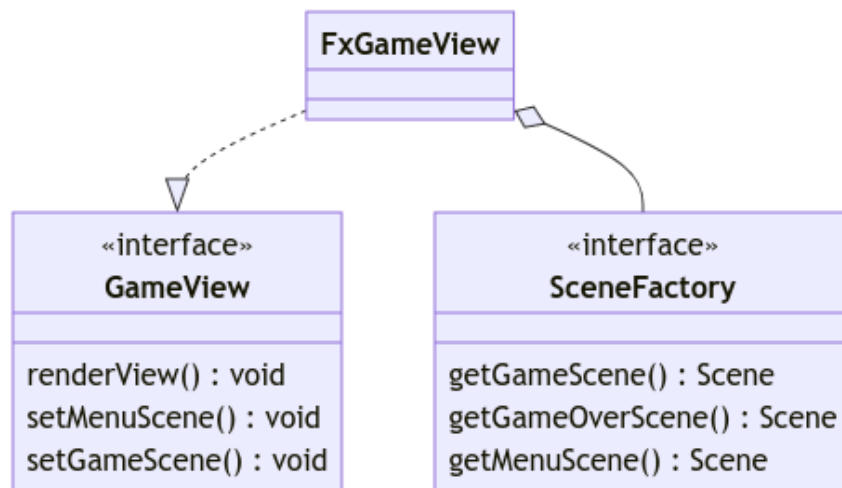


Figura 2.5 rappresentazione del pattern *Bridge* per *GameView*

Problema: Nell'utilizzo di un pattern architetturale *MVC* risulta essenziale rendere la parte di *View* completamente isolata dalla parte di *Model*. In un gioco come *Tank23* però la struttura richiede che *View* e *Model* siano in comunicazione tra di loro, poiché al variare dello stato di gioco è necessario che cambino anche le scene.

Soluzione: Per ovviare al problema si è scelto di utilizzare il pattern *Bridge*, definendo dei comportamenti di base che dovrebbe avere una parte di *View* di gioco, rendendoli però indipendenti dalle implementazioni. Se un domani si volesse cambiare tecnologia e passare a *Swing* sarebbe sufficiente sviluppare una nuova implementazione di *GameView* senza dover affatto toccare la parte di *Model*. Per la costruzione delle diverse scene si è deciso di utilizzare il pattern *Factory Method*. È importante notare però che l'interfaccia di *SceneFactory*, dovendo creare scene compatibili con la tecnologia scelta, è strettamente legata alla classe *FxGameView*.

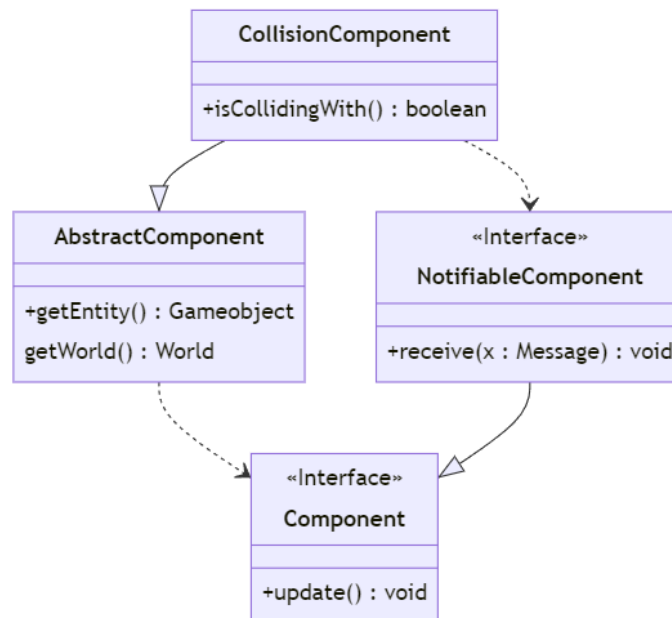


Figura 2.6 rappresentazione UML per gestione del *CollisionComponent*

Problema: Gestire l'avvenimento della collisione e notificarlo ai diversi componenti

Soluzione: La classe *CollisionComponent* è una componente di gioco che gestisce il rilevamento e la risoluzione delle collisioni tra le varie entità. La classe estende *AbstractComponent*, che fornisce un'implementazione di base per i componenti di gioco. Tramite il metodo di *update*, cioè di aggiornamento periodico, viene rilevato l'effettivo avvenimento della collisione tra due entità, l'azione che viene svolta dipende poi dal tipo di essa. Successivamente vengono notificati *EntitiesHealthComponent*, *BulletHealthComponent* e *PhysicsComponent*, nel caso l'entità possieda questi ultimi. Gestire le collisioni tramite componenti offre diversi vantaggi nell'organizzazione e nella gestione del sistema di collisione in un'applicazione, ad esempio, è possibile separare in modo chiaro le responsabilità tra gli oggetti nel sistema. Ogni oggetto può avere un componente di collisione che si occupa della sua logica di collisione, senza influenzare gli altri aspetti dell'oggetto.

Utilizzo di un FxRenderingEngine

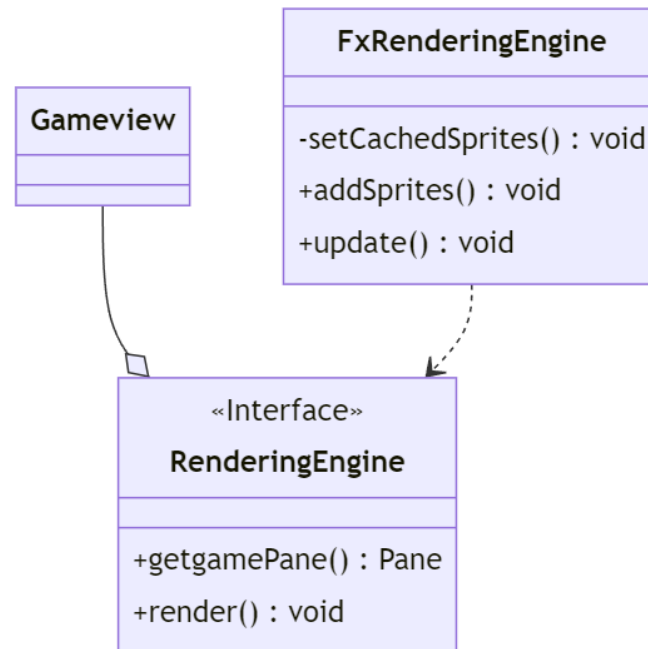


Figura 2.7: rappresentazione UML della gestione del Render di gioco

Problema: Separare il rendering del gioco dalla logica del gioco stesso e aumentare la velocità di rendering.

Soluzione: La creazione della classe *FxRenderingEngine* fornisce una classe esterna alla logica di gioco specializzata nel rendering grafico del gioco stesso, questa scelta fornisce diversi vantaggi:

- **Utilizzo di cache per le immagini:** La classe mantiene una cache di immagini per evitare il continuo caricamento delle stesse immagini da disco. Questo porta a prestazioni migliori in termini di velocità di rendering e utilizzo delle risorse.
- **Gestione delle entità:** La classe tiene traccia delle entità presenti nel mondo di gioco e dei relativi sprite. Aggiorna dinamicamente la posizione e la rotazione degli sprite in base allo stato attuale delle entità nel mondo di gioco.
- **Facilità di utilizzo:** La classe fornisce un metodo *render()* che si occupa di aggiornare e renderizzare gli sprite nel pannello di gioco. Questo semplifica notevolmente il processo di rendering e riduce la complessità del codice necessario per gestire l'aspetto visuale del gioco.
- **Scalabilità:** La classe è progettata per essere facilmente estendibile e personalizzabile. Puoi aggiungere nuovi tipi di sprite e immagini alla cache, definire nuovi comportamenti di rendering e personalizzare l'aspetto grafico del gioco in base alle proprie esigenze.

Creazione della mappa tramite file

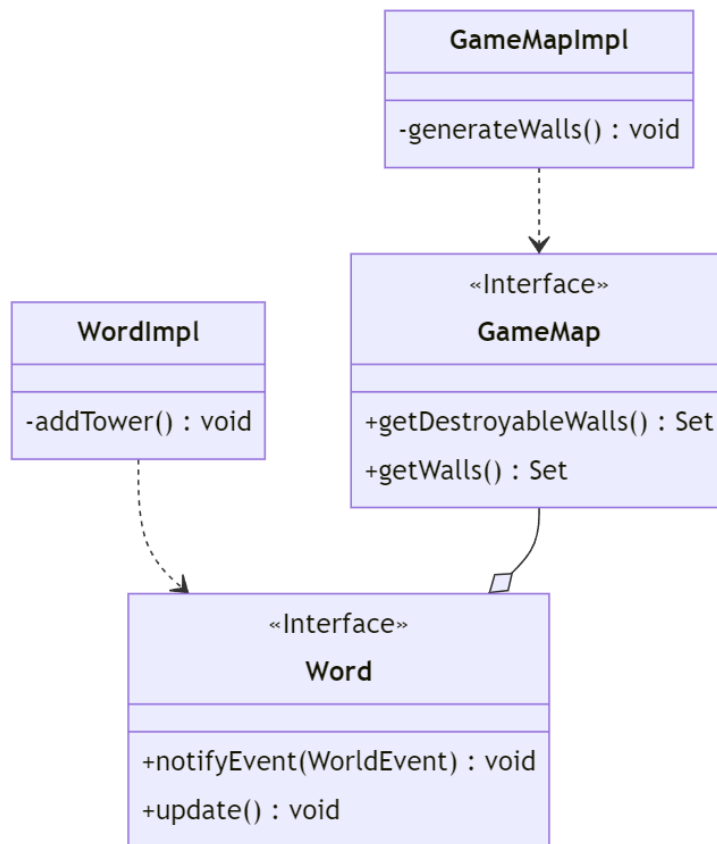


Figura 2.8: rappresentazione UML della creazione della mappa di gioco

Problema: Creazione di una mappa facilmente modificabile senza dover intaccare il codice sorgente.

Soluzione: La creazione della mappa di gioco attraverso un file di testo consente di avere diversi vantaggi:

- Separazione dei dati: L'utilizzo di un file di testo per definire la mappa consente di separare i dati della mappa dalla logica di generazione dei muri. Ciò significa che puoi modificare o sostituire facilmente la mappa senza dover modificare il codice sorgente.
- Flessibilità: Utilizzando un file di testo, puoi facilmente creare mappe personalizzate o modificare la disposizione dei muri senza richiedere modifiche nel codice. Questo rende il sistema molto più flessibile e adattabile.
- Facilità di modifica: Modificare la mappa richiede solo la modifica del file di testo, senza la necessità di ricompilare o riconfigurare l'applicazione. Ciò semplifica il processo di sviluppo e testing, consentendo di apportare rapidamente modifiche alla mappa durante lo sviluppo del gioco.
- Riutilizzabilità: Il metodo *generateWalls* può essere riutilizzato per caricare diverse mappe da file di testo. Ciò consente di creare facilmente diverse configurazioni di mappa e generare le corrispondenti strutture dei muri senza dover scrivere codice specifico per ciascuna mappa.

Giovanni Pisoni

Creazione dei proiettili

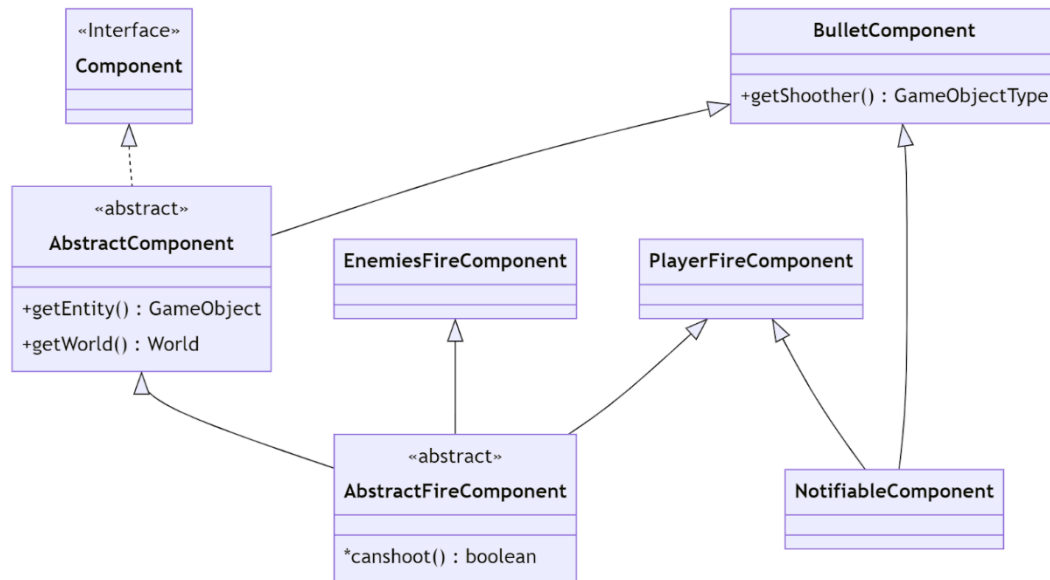


Figura 2.9 rappresentazione UML del pattern per la creazione del proiettile

Problema: La creazione dei proiettili deve avvenire in maniera tale da garantire un'estendibilità futura del codice e una comprensibile organizzazione di quest'ultimo.

Soluzione: Utilizzo di una classe Abstract in aggiunta al Component Pattern.

Ho deciso di organizzare le classi che riguardano la gestione del comportamento del *FireComponent* in una gerarchia di classi astratte, che hanno in comune il metodo protetto *canShoot* e il metodo *update*, per fornire funzionalità comuni e per facilitare l'estensione e la personalizzazione. Come è possibile notare dall'UML di sopra riportato, la classe *AbstractFireComponent* costituisce la base per tutti i componenti di "fuoco". La classe *EnemiesFireComponent* rappresenta il comportamento di "fuoco" per i nemici, basato su un timer che tiene il conto del numero di frame passati tra uno sparo e l'altro. *PlayerFireComponent* rappresenta il comportamento di "fuoco" per i giocatori, che, esattamente come i nemici, hanno un timer. L'aspetto che contraddistingue le due classi è che quella del player è notificabile, ovvero riceve una notifica dall'*InputComponent* quando avviene il click sulla tastiera del tasto relativo allo sparo, mentre *EnemiesFireComponent* non necessita di ricevere feedback. Infine, abbiamo la classe *BulletComponent*, la quale rappresenta un componente per un proiettile nel gioco, e dentro alla quale si mantiene traccia del tipo di entità che l'ha sparato.

Concludendo, i vantaggi nell'utilizzare classi astratte in questo contesto sono:

- Il riuso del codice, grazie al fatto che le funzionalità comuni tra i diversi componenti sono implementate solo nella classe *AbstractFireComponent*, evitando quindi anche la duplicazione del codice.
- Il polimorfismo, siccome le classi astratte consentono di utilizzarlo per trattare oggetti di diverse classi derivate come istanze della classe astratta di base.
- L'estendibilità, poiché la gerarchia di classi astratte consente di estendere il comportamento di "fuoco" aggiungendo nuove classi derivate, le quali possono personalizzare il comportamento sovrascrivendo i metodi astratti.

Gestione dei Round e degli Spawn

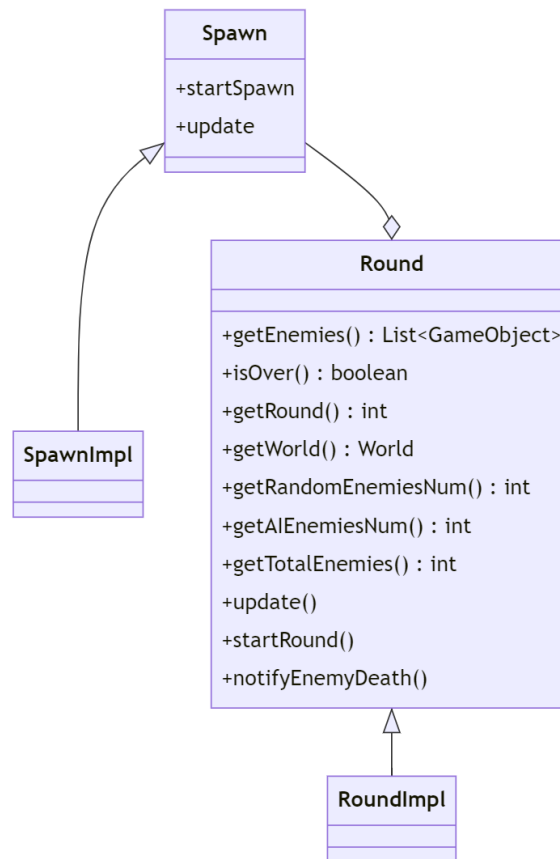


Figura 2.10 rappresentazione UML della gestione dei Round e degli spawn

Problema: Gestione di ondate di nemici a numero crescente con il proseguire dei round, gestione della generazione temporale dei nemici e gestione di errori di concorrenza.

Soluzione: Utilizzo di un “algoritmo” che calcola il numero di nemici per round, della classe *Timer* e di liste sincronizzate.

Le classi *RoundImpl* e *SpawnImpl* costituiscono l’implementazione dei round di gioco, la logica di generazione dei nemici e la gestione di essi. Per la gestione dei nemici ho pensato ad un algoritmo presente dentro il metodo *fillEnemiesList* all’interno di *RoundImpl*, che, considerando il numero del round in cui ci si trova, calcola un numero di nemici. Questo numero è direttamente proporzionale al numero del round; si calcola con $round * 3$ per i nemici randomici, mentre per i nemici che hanno un AI è pari a $round / 2$. Così facendo, avremo dei round di difficoltà crescente e con nemici sempre più numerosi e insidiosi.

Per la gestione della generazione temporale, all’interno della classe *SpawnImpl*, utilizzo la classe *Timer* di *java.util*. Il *Timer* viene programmato affinché ricominci ogniqualvolta *RoundImpl* richiama il metodo *startSpawn* di *SpawnImpl* alla fine di ogni round. Questo metodo interagisce con la funzione interna *getSpawnPos*, che prevede l’attribuzione di una posizione (scelta casualmente tra 3 opzioni possibili) nella mappa di gioco, assicurandosi però che sia possibile lo spawn. Una volta che questo elemento viene stabilito, la classe *SpawnImpl* chiama uno *SpawnEvent* in modo tale da aggiungere l’entità al mondo.

Utilizzando la classe *Timer*, e quindi anche *TimerTask*, durante lo sviluppo del gioco si è verificato un problema di concorrenza tra il thread del *Timer* e il thread di gioco. Questo problema era dovuto al fatto che le liste di appoggio che venivano utilizzate all’interno del *TimerTask* venivano modificate contemporaneamente da entrambi i thread, e così facendo si generava una “*ConcurrentModificationException*”. Per ovviare a questo problema, ho reputato consono utilizzare delle

liste thread safe, ovvero delle `synchronizedList`, che, nonostante garantiscano la coerenza dei dati nei thread, possono comportare un rallentamento delle prestazioni a causa del blocco dei thread in attesa. Avendo comunque pochi thread che lavorano in concorrenza, ho deciso che sarebbe stato comunque efficace utilizzarle. Queste tipologie di liste mi ha permesso, infatti, di gestire in modo sicuro l'accesso e la modifica concorrente ai nemici durante il processo di generazione e aggiornamento dei round.

Creazione e gestione dei menù laterali della scena di gioco

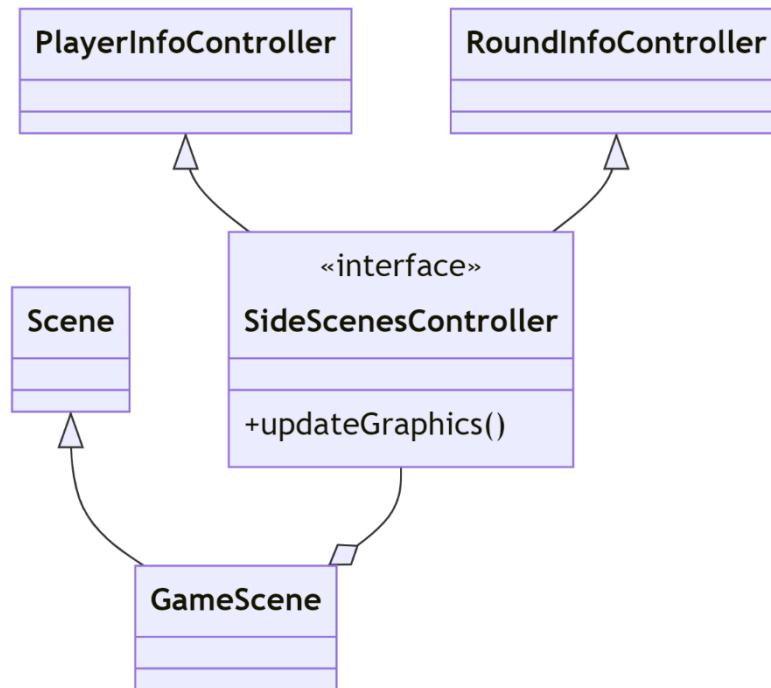


Figura 2.11 rappresentazione UML della gestione delle scene laterali.

Problema: La gestione delle scene laterali deve essere indipendente dalla generazione e dalla gestione della scena di gioco.

Soluzione: Utilizzo di una classe *GameScene* che rappresenta una scena personalizzata.

Per ovviare al problema sopra indicato, ho deciso di creare la classe *GameScene*, che rappresenta una scena personalizzata dell'interfaccia di gioco e estende la classe *Scene* di JavaFX, fornendo funzionalità aggiuntive per visualizzare le informazioni del giocatore e del round in due scene laterali indipendenti dalla scena di gioco. Utilizzare *GameScene* ha i seguenti vantaggi di avere:

- un layout flessibile, dato che utilizza come root della scena un *BorderPane*, consentendo così di organizzare facilmente gli elementi dell'interfaccia di gioco in aree specifiche.
- un caricamento dinamico degli elementi dell'interfaccia utente tramite l'utilizzo di *FXMLLoader*.
- un adattabilità dello schermo, dovuta al fatto che la classe tiene conto delle sue dimensioni.

Seguendo il pattern MVC, *GameScene* prende in input due controller, *PlayerInfoController* e *RoundInfoController*. Questi sono componenti fondamentali nell'architettura di un'applicazione grafica, dal momento che fungono da intermediari tra la vista (UI) e il modello di dati. I due controller gestiscono l'interazione dell'utente (aggiornando i contatori quando cambia il round o quando muoiono i nemici). Grazie ad essi è possibile:

- riutilizzare il codice, dato che possono essere riapplicati in diverse parti del gioco;
- una testabilità del codice, facilitando il testing della logica e dell'interfaccia utente,

- una scalabilità, dovuta al fatto che i controller consentono di aggiungere nuove funzionalità o modificare la logica di presentazione senza dover apportare modifiche significative alla vista stessa
- una migliore organizzazione del codice.

Javid Ameri

Gestione di HealthComponent

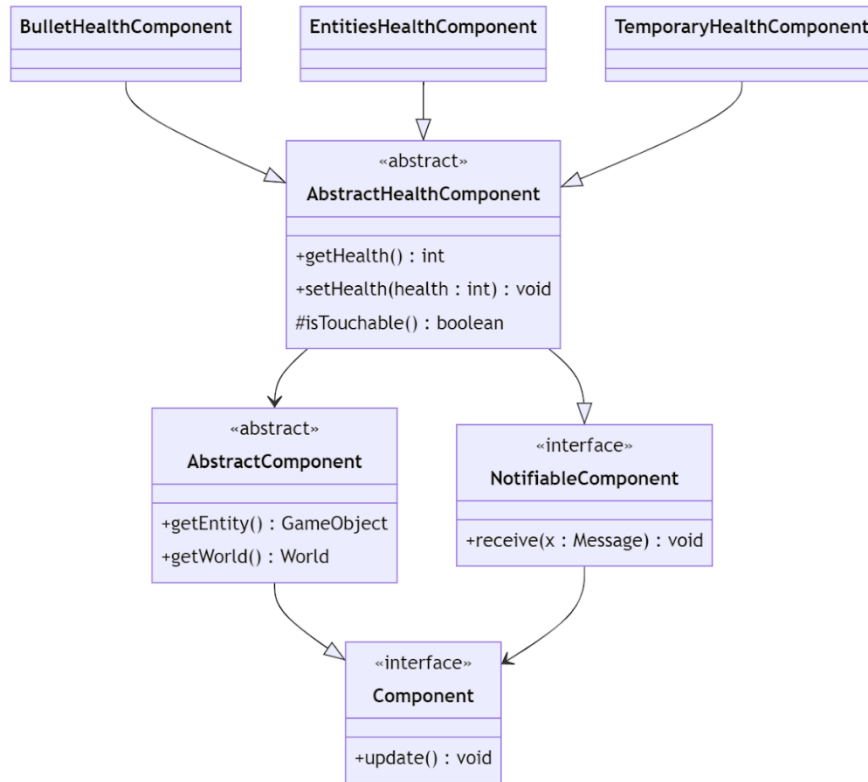


Figura 2.12 rappresentazione UML del pattern Component per l'HealthComponent

Problema:

Nel descrivere questo problema mi servirò di *due parole* fondamentali nella terminologia dei videogiochi, ma che non intendo comunque dare per scontato e per tanto provvederò a fornirvi già da ora una breve spiegazione del loro ruolo in questo contesto:

SPAWN = Il nome che prende l'azione di spawnare, ossia di "nascere" o di "rinascere" (dopo essere morti), compiuta da un'entità all'interno del gioco.

COOL DOWN = Il periodo di "cooldown", nel nostro specifico caso, descrive quel breve lasso di tempo in cui un'entità è immune a qualunque tipo di danno da fuoco. Un lasso di tempo che io ho collocato negli attimi negli attimi subito dopo essere stati colpiti o subito dopo essere spawnati a inizio gioco.

Questa immunità di cool-down (sopra citata), che inizialmente avevo progettato di fare, avrebbe riguardato esclusivamente i carri armati (quindi i GameObject di tipo Player o Enemy). Anche l'ammontare dei punti-salute avrebbe richiesto una differenziazione in base al GameObjectType, tuttavia, per ognuno di essi, la salute avrebbe sicuramente funzionato in maniera prevalentemente simile. L'obiettivo era dunque di rendere la salute dei GameObject differenziata per Type ma che fosse allo stesso tempo versatile per implementare dei metodi comuni.

Soluzione:

La soluzione è stata ovvia: ho creato la classe astratta `AbstractHealthComponent` in cui ho dichiarato e implementato i metodi che poi sarebbero serviti ai diversi `GameObjectType`. Dopodiché ho creato due classi che estendessero quest'ultima: `EntitiesHealthComponent` e `BulletHealthComponent`.

La prima costituisce il componente della salute per i seguenti tipi: `Player`, `Enemy` e `Wall`; la seconda costituisce quello per il `Bullet` type. In quest'ultima ho dovuto re-implementare solo il metodo `isTouchable`, il quale restituirà sempre "true" poiché i proiettili non sono soggetti alla funzione di cool-down (non sarebbe logico per ovvi motivi). In `EntitiesHealthComponent` invece ogni metodo della classe astratta è stato re-implementato a seconda delle esigenze dei carri armati, ad esempio implementando correttamente il cool-down.

Gestione dei comandi di Input

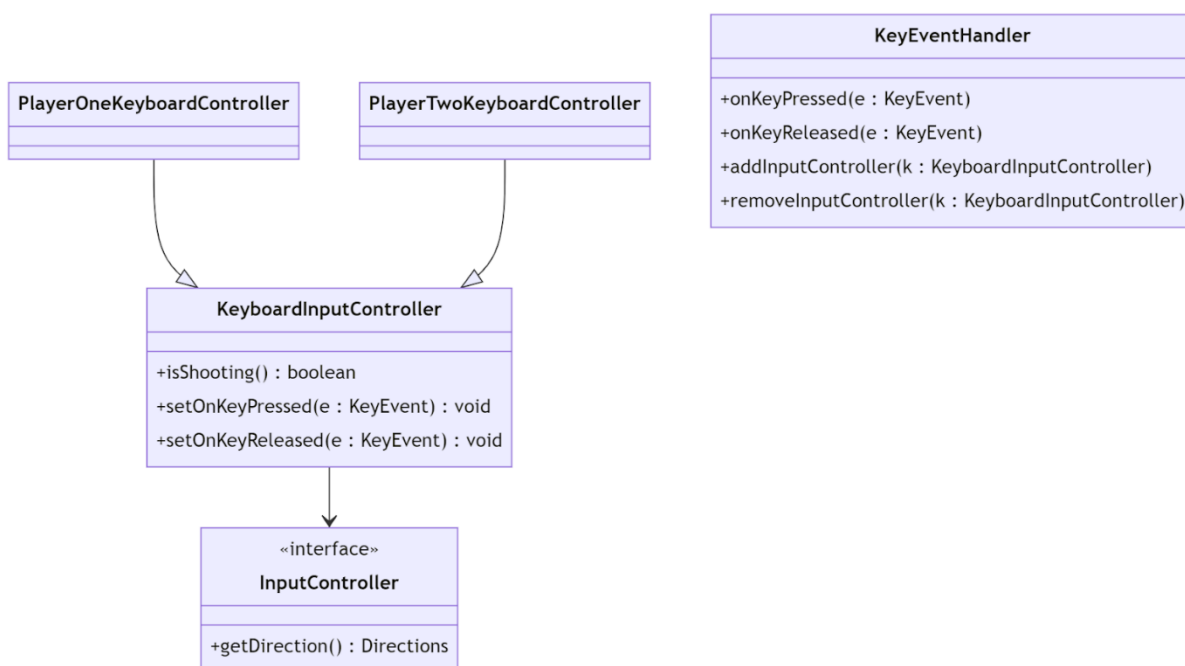


Figura 2.13 rappresentazione UML delle classi di InputController

Problema:

Dovevo progettare il controller (ossia l'insieme dei comandi di input) per muovere il giocatore, che al momento era soltanto uno. La futura inclusione di un secondo giocatore era incerta ma non per questo escludibile, dovevo dunque pianificare un'eventuale estendibilità che avrebbe visto l'aggiunta di più controller per la modalità multiplayer.

Soluzione:

Dopo aver creato l'interfaccia Input Controller, l'ho fatta estendere da un'altra interfaccia KeyboardInputController (rappresentate il generale modello di un controller), la quale definisce i metodi necessari per gestire l'input dalla tastiera, inclusa la rilevazione dell'azione di sparo e la gestione degli eventi di pressione e rilascio dei tasti.

A quel punto ho creato il primo effettivo controller da tastiera destinato al primo giocatore: PlayerOneKeyboardController. Ed ecco arrivare allora la classe per gestire il problema sopra descritto: KeyEventHandler. Quest'ultima crea a priori una lista che andrà poi a contenere tutti i controller che si desidera aggiungere (ed eventualmente rimuovere). Tramite i metodi onKeyPressed e OnKeyReleased, la classe è in grado di gestire autonomamente le azioni di ogni controller presente nella sua lista. Difatti questi due metodi prendono in input l'evento di un tasto (KeyEvent) che è stato premuto oppure rilasciato, e notificano tale evento ad ogni controller in lista. Il controller notificato chiamerà allora il proprio metodo setOnKeyPressed o setOnKeyReleased (passando l'evento come parametro).

Avendo così garantito la funzionalità di più controllers simultaneamente (multiplayer), è stato poi creato un secondo controller PlayerTwoKeyboardController per implementare finalmente il secondo giocatore.

Gestione dei menù principali

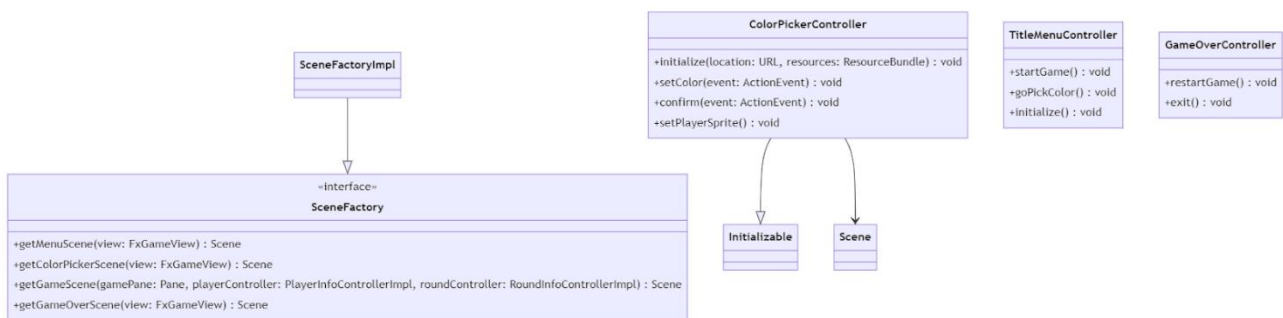


Figura 2.14 rappresentazione UML delle classi dei menù principali

Problema:

Volevo che fin dalla schermata del titolo (controllata da TitleMenuController), al giocatore fosse offerta la possibilità di personalizzare il proprio carro armato con uno dei tanti colori proposti. Così ho creato una scena, accessibile tramite un tasto dalla schermata del titolo, che permettesse tale funzionalità. Queste due scene comunicanti però erano una potenziale fonte di potenziali dubbi e problemi. Descriverò qui sotto come ogni piccolo passo fatto nella classe ColorPickerController (ossia il controller della schermata di personalizzazione) fosse poi seguito da un “ah però poi dovrei fare questo”. Ovviamente tralascerò i problemi di design e grafica (come, ad esempio, l'impostazione di uno sfondo) poiché non costituiscono un elemento di valutazione.

Soluzione:

La prima cosa di ColorPickerController che ho realizzato è stata implementare la scelta del colore per il giocatore tramite una stringa che assumerà il valore di quello selezionato nella choice box.

Ma se il giocatore, una volta aperta la scena, non dovesse selezionare alcun valore? Allora da subito ho inizializzato la stringa con un valore: “Pink”, così che di default il carro armato del giocatore sarebbe stato rosa. Comunque, alla stringa del colore sarebbe stato aggiunto un prefisso e suffisso in modo che diventasse il percorso esatto dell'immagine da selezionare come sprite (immagine o animazione che viene visualizzata

sullo schermo per rappresentare un'entità di gioco) del giocatore. Una volta cliccato il pulsante di conferma, si sarebbe tornati alla schermata del titolo per cliccare il pulsante "START GAME" per iniziare il gioco. A quel click, sarebbe stato creato il mondo di gioco che si sarebbe servito della stringa del percorso nel momento della creazione del giocatore.

Tutto bene finché non decisi di aggiungere un altro elemento alla nostra scena: un interruttore che, una volta cliccato, avrebbe aggiunto un secondo giocatore rivelando anche una seconda choice box per la scelta del suo colore. Nessun problema con l'interruttore, ma come potevo avvisare il mondo di aggiungere un secondo giocatore se questo veniva creato dopo aver premuto il pulsante di start situato nella scena del titolo?? Ho preso la scelta meno contorta e più lineare: creare il mondo in `ColorPickerController` così da gestire a piacimento i due giocatori.

Ma se l'utente, trovandosi sulla schermata del titolo, volesse di cliccare direttamente il pulsante "STAR GAME"? Senza passare per la scena di personalizzazione non verrebbe creato alcun mondo! Questo problema è stato risolto creando nel costruttore della view (classe `FxGameView`) un mondo (di default, con un solo giocatore) a priori. Se invece si passa per la schermata di personalizzazione allora, una volta cliccato il pulsante di conferma, il mondo di default in `FxGameView` sarà rimpiazzato dal mondo con le preferenze (riguardanti i giocatori) scelte nella nostra scena.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per sviluppare i test automatizzati è stato utilizzato JUnit 5.

- **DirectionsTest:** Testa le funzioni riguardanti i vettori delle direzioni, in particolare `testFlipped` racchiude dentro di sé anche il test del metodo `fromVector`, poiché utilizzato dal metodo `flipped`.
- **Point2DTest:** Si occupa di testare il corretto funzionamento di tutti i metodi della classe `Point2D`.
- **Vector2DTest:** Si occupa di testare il corretto funzionamento di tutti i metodi della classe `Vector2D`.
- **PairTest:** Si occupa di testare le funzioni `hashCode` e `equals` di `Pair`.
- **GridGraphNodeTest:** Viene testato il corretto funzionamento della ricerca degli indici adiacenti e dei metodi `hashCode` e `equals`.
- **VisitableGridGraphTest:** Viene testato il corretto funzionamento della visita in ampiezza e varie situazioni di aggiunta e rimozione di nodi al grafo.
- **GameObjectTypeFactoryTest:** Viene testato la corretta lettura del file `json` contenente le informazioni sui diversi tipi di `GameObject`.
- **MessageTest:** Testa il funzionamento dei messaggi, cioè che la classe `Message` restituisca correttamente il messaggio ricevuto.
- **NotifiableComponentTest:** Testa il funzionamento della classe `Notifiable Component`, controllando se il messaggio viene effettivamente ricevuto.
- **Rect2DTest:** testa il corretto funzionamento della classe `Rect2D`, controllando la giusta restituzione in caso di intersezione.

3.2 Metodologia di lavoro

Prima di iniziare lo sviluppo vero e proprio del videogioco, abbiamo reputato doveroso compiere un'analisi iniziale che prevedeva l'impostazione dell'architettura di gioco e la creazione delle interfacce del dominio, in modo tale da avere solide basi. Scegliendo questa strada, e essendoci suddivisi inizialmente il lavoro da svolgere, ci siamo riuniti solo quando risultava necessario accorpate le varie implementazioni e per la discussione di aggiunte. Le uniche problematiche che si sono sviluppate nel corso del lavoro riguardavano piccole modifiche alle interfacce, come aggiunte di metodi non previsti in precedenza. Inoltre, questo *modus operandi* ci ha permesso di lavorare parallelamente in completa autonomia per la maggior parte dello sviluppo dell'applicazione. Abbiamo deciso di utilizzare il DVCS con l'approccio spiegato in aula, ovvero, una volta creato il repository ogni membro del gruppo ne ha fatto un clone in maniera tale da lavorarci in locale e in autonomia condividendo le modifiche e le aggiunte fatte tramite `pull/push` e lavorando su branch opportunamente specializzati quando necessario.

Federico Diotallevi

In autonomia mi sono occupato di:

- Sviluppo di GameLoop e GameEngine (package tnk23.core)
- Sviluppo del grafo per l'ai (package tnk23.game.graph)
- Sviluppo di AiComponent (package tnk23.game.model.components)
- Sviluppo di AiControllerFactory e FollowTargetAi (package tnk23.input)
- Sviluppo di GameView e dell' implementazione FxGameView (package tnk23.view)

In collaborazione mi sono occupato di:

- Con Gioele Santi:
 - Sviluppo di World (package tnk23.game.model)
- Con Giovanni Pisoni:
 - Sviluppo di World (package tnk23.game.model)
 - Sviluppo di GameState ((package tnk23.game.model)
- Con Javid Ameri:
 - Sviluppo di GameObjectType e GameObjectTypeManager (package tnk23.game.model)

Gioele Santi

In autonomia mi sono occupato di:

- Gestione dei componenti di fisica, collisione e grafica (package tnk23.game.components)
- Gestione della creazione e implementazione della mappa (package tnk23.game.model)
- Gestione del rendering grafico del gioco (package tnk23.view)

In collaborazione mi sono occupato di:

- Con Federico Diotallevi:
 - Gestione di World (package tnk23.game.model)
- Con Giovanni Pisoni:
 - Gestione di World (package tnk23.game.model)

Giovanni Pisoni

In autonomia mi sono occupato di:

- Gestione dei round (package tnk23.game.model)
- Gestione degli spawn (package tnk23.game.model)
- Gestione dei proiettili (package tnk23.game.components)
- Creazione e gestione dei menù laterali (package tnk23.view)

In collaborazione mi sono occupato di:

- Con Federico Diotallevi:
 - Gestione di Game State (package tnk23.game.model)
 - gestione di World (package tnk23.game.model)
- Con Gioele Santi:
 - Gestione di World (package tnk23.game.model)
- Con Javid Ameri:
 - Gestione degli eventi (package tnk23.game.events)

Javid Ameri

In autonomia mi sono occupato di:

- Creazione e implementazione delle entità di GameObject (package tnk23.game.model)
- Creazione e implementazione dei menù di gioco principali (TitleMenuController, ColorPickerController e GameOverController) e della SceneFactory (package tnk23.view)
- Gestione completa dell'input da tastiera (package tnk23.input)
- Gestione della vita delle entità di gioco (HealthComponent) (package tnk23.game.components)

In collaborazione mi sono occupato di:

- Con Federico Diotallevi:
 - Sviluppo di GameObjectType e GameObjectTypeManager (package tnk23.game.model)
- Con Giovanni Pisoni:
 - Gestione dei WorldEvents (package tnk23.game.events)

3.3 Note di sviluppo

Federico Diotallevi

Utilizzo di Stream, lambda expressions e method references:

Utilizzate in moltissimi punti, i seguenti sono solo esempi:

Permalink:

- <https://github.com/DiottaNax/OOP22-TNK23/blob/639d6378febbb4c9fa78ff0111df766c68efee4b/src/main/java/it/unibo/tnk23/game/graph/impl/GameGraph.java#L127-L129>
- <https://github.com/DiottaNax/OOP22-TNK23/blob/639d6378febbb4c9fa78ff0111df766c68efee4b/src/main/java/it/unibo/tnk23/game/graph/impl/GameGraph.java#L127-L129>
- <https://github.com/DiottaNax/OOP22-TNK23/blob/639d6378febbb4c9fa78ff0111df766c68efee4b/src/main/java/it/unibo/tnk23/game/graph/impl/VisitableGridGraph.java#L71-L74>

Utilizzo generics

Permalink:

- <https://github.com/DiottaNax/OOP22-TNK23/blob/639d6378febbb4c9fa78ff0111df766c68efee4b/src/main/java/it/unibo/tnk23/game/graph/api/VisitableGraphDecorator.java#L14-L15>
- <https://github.com/DiottaNax/OOP22-TNK23/blob/639d6378febbb4c9fa78ff0111df766c68efee4b/src/main/java/it/unibo/tnk23/game/graph/api/VisitableGraph.java#LL12C1-L12C33>

Utilizzo libreria JSON.simple

Permalink:

<https://github.com/DiottaNax/OOP22-TNK23/blob/639d6378febbb4c9fa78ff0111df766c68efee4b/src/main/java/it/unibo/tnk23/game/model/impl/GameObjectTypeManager.java#L41-L54>

Utilizzo libreria JavaFX

JavaFX è stato utilizzato, in maniera superficiale, nella classe *FxGameView*

Permalink:

<https://github.com/DiottaNax/OOP22-TNK23/blob/be6ea81d90a93c2a27eb7bc6ae42aec03b5b6553/src/main/java/it/unibo/tnk23/view/impl/FxGameView.java#L53-L64>

Algoritmo di visita in ampiezza

Per la visita in ampiezza ho utilizzato lo pseudocodice fornito nel libro *Introduzione agli algoritmi e strutture dati* di Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein.

Gioele Santi

Utilizzo di Stream e lambda expressions

Utilizzato in diverse parti nel codice:

Permalink:

- <https://github.com/DiottaNax/OOP22-TNK23/blob/bc15f22f3b1a471a5abfe5abbe418fccf1c516d6/src/main/java/it/unibo/tnk23/game/components/impl/CollisionComponent.java#L59-L64>
- <https://github.com/DiottaNax/OOP22-TNK23/blob/bc15f22f3b1a471a5abfe5abbe418fccf1c516d6/src/main/java/it/unibo/tnk23/game/model/impl/WorldImpl.java#L187-L192>
- <https://github.com/DiottaNax/OOP22-TNK23/blob/bc15f22f3b1a471a5abfe5abbe418fccf1c516d6/src/main/java/it/unibo/tnk23/view/impl/FxRenderingEngine.java#L59-L63>

Utilizzo di JavaFX

Permalink:

<https://github.com/DiottaNax/OOP22-TNK23/blob/26a756bd48a5733b90b933764056d80b74cfe5d9/src/main/java/it/unibo/tnk23/view/impl/FxRenderingEngine.java#L18>

Giovanni Pisoni

Utilizzo di Stream e lambda

Utilizzato in varie parti del codice, ad esempio:

Permalink:

- <https://github.com/DiottaNax/OOP22-TNK23/blob/bc15f22f3b1a471a5abfe5abbe418fccf1c516d6/src/main/java/it/unibo/tnk23/game/model/impl/SpawnImpl.java#L116-L120>
- <https://github.com/DiottaNax/OOP22-TNK23/blob/bc15f22f3b1a471a5abfe5abbe418fccf1c516d6/src/main/java/it/unibo/tnk23/game/model/impl/RoundImpl.java#L213>

Utilizzo di Optional

Permalink:

<https://github.com/DiottaNax/OOP22-TNK23/blob/bc15f22f3b1a471a5abfe5abbe418fccf1c516d6/src/main/java/it/unibo/tnk23/view/impl/PlayerInfoControllerImpl.java#L47>

Utilizzo di JavaFX

La classe `GameScene` estende la classe di JavaFX `Scene` e prende in input un root di tipo *BorderPane*, anch'esso appartenente a JavaFX.

Permalink:

<https://github.com/DiottaNax/OOP22-TNK23/blob/bc15f22f3b1a471a5abfe5abbe418fccf1c516d6/src/main/java/it/unibo/tnk23/view/impl/GameScene.java#L20>

Javid Ameri

Utilizzo di Stream e Lambda Expression

Utilizzati in più parti, degli esempi sono:

Permalink:

- <https://github.com/DiottaNax/OOP22-TNK23/blob/8ff79cf83f20ee515a95a9532e3564cf562be17f/src/main/java/it/unibo/tnk23/game/model/impl/GameObjectImpl.java#L70-L74>
- <https://github.com/DiottaNax/OOP22-TNK23/blob/8ff79cf83f20ee515a95a9532e3564cf562be17f/src/main/java/it/unibo/tnk23/input/impl/KeyEventHandler.java#L40>
- <https://github.com/DiottaNax/OOP22-TNK23/blob/8ff79cf83f20ee515a95a9532e3564cf562be17f/src/main/java/it/unibo/tnk23/game/model/impl/GameObjectImpl.java#L108-L111>

Utilizzo Generics

Utilizzato spesso, come ad esempio:

Permalink:

- <https://github.com/DiottaNax/OOP22-TNK23/blob/8ff79cf83f20ee515a95a9532e3564cf562be17f/src/main/java/it/unibo/tnk23/game/components/impl/EntitiesHealthComponent.java#L47-L49>
- <https://github.com/DiottaNax/OOP22-TNK23/blob/8ff79cf83f20ee515a95a9532e3564cf562be17f/src/main/java/it/unibo/tnk23/game/model/impl/GameObjectImpl.java#L70-L74>

Utilizzo di JavaFx

Utilizzato davvero molto di frequente, seguono soltanto alcuni esempi:

Permalink:

- <https://github.com/DiottaNax/OOP22-TNK23/blob/7150d6750ca028f6ed84bd988b743a9586720d86/src/main/java/it/unibo/tnk23/input/impl/PlayerOneKeyboardController.java>
- <https://github.com/DiottaNax/OOP22-TNK23/blob/7150d6750ca028f6ed84bd988b743a9586720d86/src/main/java/it/unibo/tnk23/view/impl/ColorPickerController.java>
- <https://github.com/DiottaNax/OOP22-TNK23/blob/7150d6750ca028f6ed84bd988b743a9586720d86/src/main/java/it/unibo/tnk23/view/impl/SceneFactoryImpl.java>

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Federico Diotallevi

Sono molto soddisfatto del risultato finale che abbiamo ottenuto come gruppo. Da parte mia sono felice di aver affrontato il progetto, poiché mi ha aiutato a crescere molto dal punto di vista dello sviluppo. Seppur questo sia stato un progetto con tempo dedicato relativamente breve, è stato effettivamente il primo progetto serio in cui mi sono cimentato e in particolare il mio primo progetto di gruppo. Il fatto di doversi immergere nella progettazione, scelta di design e architettura del software ha richiesto uno sforzo di ricerca che difficilmente avrei fatto senza un progetto assegnato. Dal punto di vista della progettazione mi ritengo abbastanza soddisfatto, se considerato il tempo dedicato, ma ci sono ancora grandi progressi che si possono fare. Tutto sommato mi sono cimentato in una sfida che ritengo abbastanza complicata, la gestione dell'algoritmo di pathfinding, infatti, è stata fatta grazie alle conoscenze ottenute negli altri corsi mettendo in pratica tali nozioni sul campo e vedendo il loro effettivo uso in una semplice applicazione.

Sono felice anche del lavoro che abbiamo svolto come gruppo, che ha lavorato in modo consapevole e produttivo nonostante l'inesperienza e mi ha permesso di confrontarmi con le opinioni di altri collaboratori di cui nutro molta stima e trovare soluzioni a problemi che fino a poco tempo fa non credevo nemmeno di essere in grado di risolvere.

In futuro spero di aver modo di continuare a migliorare il progetto prendendolo come occasione ed esercizio per poter migliorare la mia abilità di programmatore e progettista.

Gioele Santi

Sono molto contento e soddisfatto del risultato finale del progetto, che ritengo sia stato frutto di una ottima collaborazione all'interno del gruppo. Personalmente l'idea di creare un gioco mi ha sempre incuriosito, anche se all'inizio ero un po' intimorito dalla mole del lavoro da effettuare. La stretta collaborazione con gli altri membri del gruppo mi ha subito stimolato e mi ha fatto capire l'importanza di condividere le proprie idee e trovare il metodo migliore per svolgere le cose. Riconosco l'importanza di aver svolto una minuziosa analisi prima di iniziare a sviluppare il gioco, in cui ogni elemento del gruppo ha espresso il proprio parere, suggerendo modifiche e diverse possibilità, così da migliorare l'idea per il progetto. Sono felice di aver approfondito il linguaggio di programmazione utilizzato, di aver imparato ad usare un linguaggio più tecnico e di aver sperimentato tecnologie utili anche nel mondo lavorativo (es. GitHub). In futuro sicuramente rimetterò mano al codice per migliorarlo e aggiungere nuove funzionalità, come l'implementazione di nuove mappe con all'interno oggetti diversi.

Giovanni Pisoni

Sono estremamente soddisfatto dell'esperienza che questo progetto mi ha dato l'occasione di svolgere. Per quanto riguarda l'aspetto organizzativo, sono felice di come siamo riusciti a coordinare il progetto in maniera adeguata ed equilibrata, distribuendo il carico di lavoro in modo equo a ogni membro del gruppo. Siamo riusciti a creare una buona atmosfera e un buon teamwork, aiutandoci nei momenti di dubbio e incertezza e organizzando fin da subito una buona analisi e progettazione del lavoro. Per quanto riguarda la mia parte di contributo, sono soddisfatto di come ho approcciato questa sfida, cercando fin da subito di mettermi in gioco pur non avendo solide basi di game programming. Sotto questo punto di vista, mi ha permesso di migliorare molto le mie competenze nello sviluppo di codici, facendomi scoprire aspetti della programmazione a oggetti che non avevamo avuto modo di affrontare durante il corso, come l'utilizzo e le varie funzioni di JavaFX e le liste thread safe, certo comunque che sia possibile approfondire ancora la conoscenza di questi argomenti. Nonostante non sia il primo lavoro di gruppo a cui ho preso parte, devo ammettere che questo è stato senza dubbio il più stimolante e appagante, considerando sia la quantità di tempo che gli ho dedicato, sia i diversi problemi a livello di programmazione, organizzazione e realizzazione che ho dovuto affrontare.

Una volta effettuata la consegna ritengo che ci siano delle parti da migliorare e funzionalità aggiuntive da migliorare, e probabilmente mi metterò ancora più in gioco, provando a renderlo un gioco il più completo possibile.

Javid Ameri

Grazie a questo progetto ho provato la gioia di essere, nel mio piccolo, Game Developer e Game Designer simultaneamente. Certo, il tempo era poco e forse lo stile arcade limitante per parlare di Game Design, ma questa rimane comunque la mia prima effettiva esperienza dell'obiettivo che ho sempre puntato.

Per tanto, ringrazio i miei compagni Federico, Gioele e Giovanni per avermi lasciato piena libertà in questo e di essersi fidati della mia mano: sapevano quanto fosse importante per me. In questo progetto ci ho messo tanto cervello quanto cuore! La parte grafica varrà anche zero punti per i professori, ma sicuramente non li vale per il futuro a cui punto. E per questo sono felice che, tra un codice e l'altro, io sia riuscito a trovare il tempo per disegnare ogni cosa visibile, dalla prima all'ultima (animazioni comprese), di questo gioco.

Se devo guardare il me reduce di questo lungo percorso, lo vedo sicuramente molto più abile nelle strategie di progettazione del codice e più padroneggiante del linguaggio Java. Ho imparato soprattutto a scomporre problemi e classi complesse, scoprendo così delle strade molto più comode.

Sono felice di aver affrontato un progetto tanto arduo lavorando in gruppo, per di più con delle persone su cui posso contare e con le quali posso confrontarmi. Mi ha reso felice dare una mano quando ho potuto, così come mi ha confortato sapere che c'era sempre qualcuno pronto a darla a me se ne avessi avuto bisogno.

In futuro mi piacerebbe estendere questo prototipo di videogioco che tanto significa per me.

Ad esempio, con dei bonus che permettano trasformazioni, scudi, recuperi, incrementi di statistiche e altro ancora.

Vorrei poi ampliarlo con tracce audio e con nuovi livelli, nuove mappe, nuove meccaniche e soprattutto di qualunque cosa necessiti di una rappresentazione grafica!

Appendice A

Guida utente

Il lancio dell'applicazione si apre con la schermata del titolo. Questa raffigura l'artwork di copertina del videogioco (con tanto di titolo). Possiamo notare due bottoni situati nella zona sud dello schermo. Cliccando quello sopra, con su scritto "PICK YOUR COLOR", si accede alla scena di personalizzazione del carro armato.

Difatti, cliccando sulla casella in alto, sarà possibile scegliere a piacimento il colore del proprio carro armato. Le opzioni previste sono otto: Pink (rosa), Red (rosso), Orange (arancione), Yellow (giallo), Green (verde), Cyan (ciano), Blue (blu) e infine Purple (viola).

Cliccando sull'interruttore scorrevole in cima alla schermata, invece, sarà possibile attivare la presenza del secondo giocatore. Ciò abiliterà anche la sua casella di scelta del colore. Una volta soddisfatti dei colori selezionati, si clicca il bottone in basso "CONFIRM" per confermare la scelta.

A questo punto ci si ritrova nella schermata del titolo, pronti per iniziare il gioco. In caso di ripensamenti sul colore scelto, si è liberi di ri-cliccare il bottone sopra per tornare alla scena di personalizzazione. Per iniziare la partita invece, basta cliccare il bottone sotto: "START GAME!".

Troverete la guida ai comandi alla fine di questo paragrafo.





L'obiettivo del gioco è eliminare quanti più nemici possibili evitando che la torre, situata in fondo allo schermo al centro, venga distrutta ed evitando di esaurire le vite disponibili (tre per ogni giocatore). In questi due ultimi casi descritti, la partita termina e appare una schermata di Game-Over. Qui si trovano due pulsanti (situati nella parte sud dello schermo). Il pulsante più in alto, "RESTART", ci permette di tornare nuovamente alla schermata del titolo iniziale così da iniziare una nuova partita. Il pulsante più in basso, "EXIT", terminerà il gioco chiudendo la finestra, ponendo così fine all'intera applicazione.

GUIDA AI COMANDI DI INPUT PER I GIOCATORI:

Entrambi i giocatori utilizzeranno dei tasti presenti su ogni tastiera.

Per comodità, i tasti del Giocatore 1 si troveranno sulla parte destra della tastiera, quelli del Giocatore 2 sulla parte sinistra.

CARRO ARMATO DEL GIOCATORE 1:

- Premere il tasto “” (freccia direzionale SU) per muoversi verso NORD
- Premere il tasto “” (freccia direzionale GIÙ) per muoversi verso SUD
- Premere il tasto “” (freccia direzionale DESTRA) per muoversi verso EST
- Premere il tasto “” (freccia direzionale SINISTRA) per muoversi verso OVEST
- Premere il tasto “BACKSPACE” (tasto per cancellare, detto anche CANC o DELETE) per SPARARE

CARRO ARMATO DEL GIOCATORE 2:

- Premere il tasto “W” per muoversi verso NORD
- Premere il tasto “S” per muoversi verso SUD
- Premere il tasto “D” per muoversi verso EST
- Premere il tasto “A” per muoversi verso OVEST
- Premere il tasto “Q” per SPARARE

Bibliografia

Nella fase di analisi abbiamo utilizzato il libro consigliato in aula durante lezione intitolato “Game Programming Patterns” di Robert Nystrom, e il repository git mostratoci a lezione dal professor A. Ricci:

<https://github.com/pslab-unibo/oop-game-prog-patterns-2022>. Inoltre, le classi Vector2D, Point2D e Pair2D sono delle rivisitazioni delle classi viste nel repository qui indicato.