Aren's Adventures

Alessia Lombardi, Giulia Montini, Lorenzo Signoretti

Giugno 2022

Indice

Capitolo 1

Analisi

Il software realizzato per il corso di Programmazione ad Oggetti è ***Aren's Adventures***, un videogioco ispirato al primo capitolo della saga di The Legend of Zelda. L'obiettivo del gioco è quello di trovare i quattro oggetti nascosti sparsi all'interno della mappa stando attenti, però, a trovarli nel giusto ordine.

La difficoltà nel gioco sta nel riuscirsi ad orientare all'interno della mappa capendo dove potrebbero essere gli oggetti nascosti.

1.1 Requisiti

Requisiti funzionali :

- Vari livelli con ambientazioni differenti
- Possibilità di cambiare le impostazioni di gioco
- Gestione degli eventi scatenati dall'interazione del giocatore con gli oggetti
- Gestione dell'inventario del giocatore
- Gestione degli eventi scatenati solo in determinate condizioni (*es.* un determinato oggetto presente nell'inventario)
- Gestione delle collisioni del giocatore con i blocchi
- Gestione del cambiamento delle mappe di gioco

Requisiti non funzionali :

- Grafica accattivante e intuitiva
- Movimenti fluidi del giocatore

1.2 Analisi e modello del dominio

Il gioco è composto da un mondo a sua volta composto da più mappe, tra cui il giocatore può muoversi liberamente. Il giocatore deve raccogliere diversi oggetti di gioco sparsi per le mappe. La maggior parte di essi può essere raccolto solo se un altro oggetto specificato è già stato raccolto.

Gli oggetti raccolti vengono poi inseriti all'interno dell'inventario del giocatore.

Inoltre, all'interno delle mappe, il giocatore troverà alcuni oggetti che lo bloccheranno e che non gli consentiranno il passaggio, pertanto si avrà bisogno di sviluppare un adeguato sistema di fisica che consenta lo sviluppo di collisioni tra il giocatore e questi oggetti.

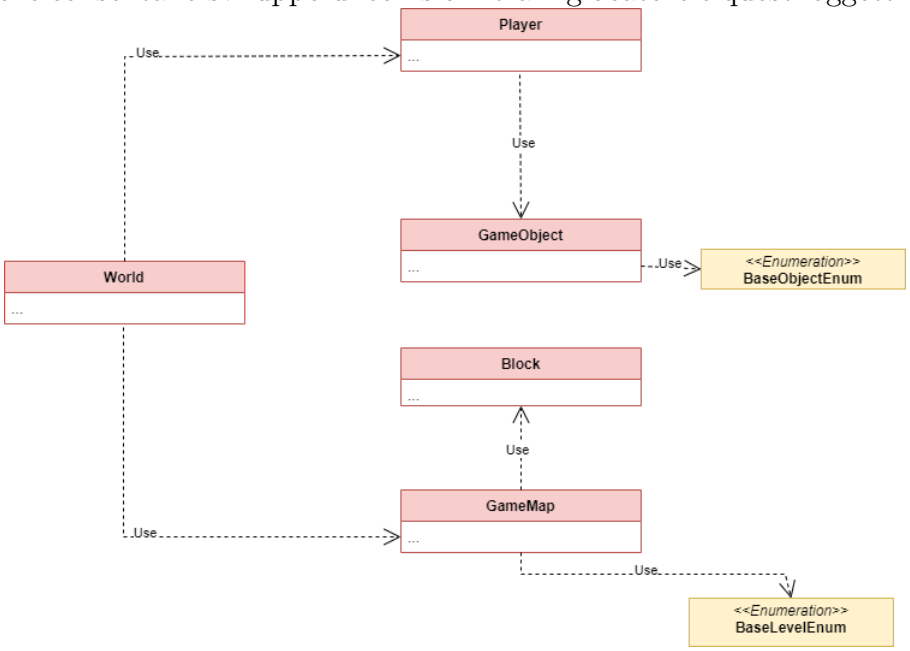

Figura 1.1: Rappresentazione UML dell'architettura base delle entità.

Capitolo 2

Design

2.1 Architettura

L'architettura di Aren's Adventure segue il modello architetturale MVC. In questo modo si rende indipendente l'interfaccia grafica dagli altri componenti.

Il *GameState* svolge il ruolo di Model, pertanto gestisce le varie entità e le relative interazioni. Controller e View interagiscono attraverso il *GameEngine*, che al suo interno contiene il metodo *loop*. Quest'ultimo costituisce il motore del gioco e ciclicamente processa gli input, aggiorna lo stato ed esegue il render dell'interfaccia grafica.

Grazie all'architettura MVC è possibile modificare la *View* senza impattare il Controller o il Model; ad esempio sostituendo la libreria grafica da *Java Swing* a *JavaFX* non si avrà necessità di modificare il model.

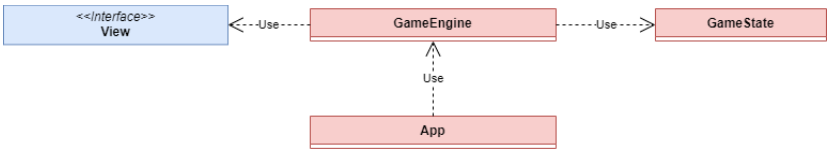

Figura 2.1: Rappresentazione UML dell'architettura base.

2.2 Design dettagliato

Alessia Lombardi :

Generazione degli elementi di gioco :

Problema: Il gioco presenta la necessità di caricare e posizionare i diversi componenti all'interno della schermata di gioco.

Soluzione: Per fare ciò sono partita utilizzando il pattern creazionale ***Static Factory***. La classe *GameFactory* si occupa di creare le istanze delle diverse entità, mentre la classe *GameState* le inizializza e gestire i diversi stati del gioco.

Per creare i blocchi di confine delle varie *GameMap* ho realizzato la classe *MapsLoader* che carica il file contenente le posizioni dei blocchi, in modo da rendere il codice più comprensibile e snello (come mostrato nella Figura 2.3).

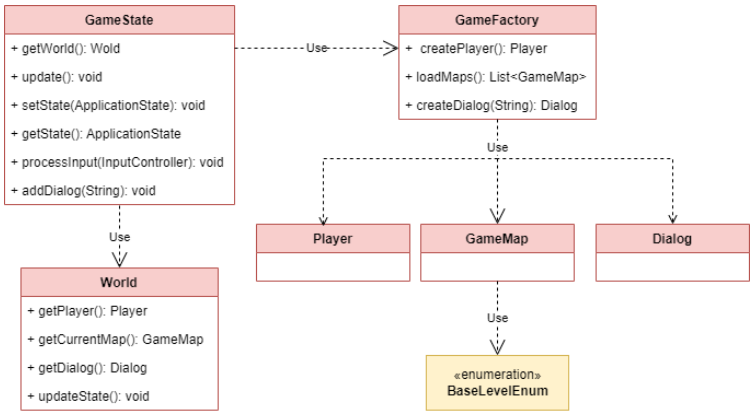

Figura 2.2: Rappresentazione UML della generazione degli elementi di gioco.

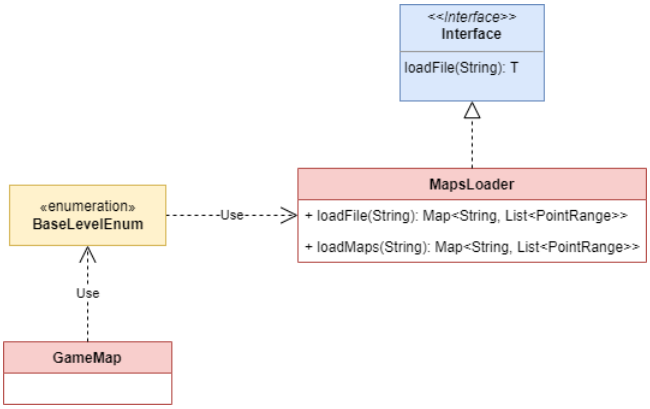

Figura 2.3: Rappresentazione UML della gestione del caricamento delle game map.

Gestione degli input :

Problema: Il sistema deve ricevere input da tastiera per permettere il movimento del player e l'interazione con i blocchi.

Soluzione: Per la gestione degli input è stato utilizzato il pattern ***Command***; infatti *InputComponent* funge da interfaccia generica per l'esecuzione del comando, che sarà poi implementato nello specifico dalle classi che realizzano tale interfaccia.

La classe *KeyListenerImpl* permette di intercettare l'input da tastiera e di chiamare il corrispettivo controller per attivare il comando.

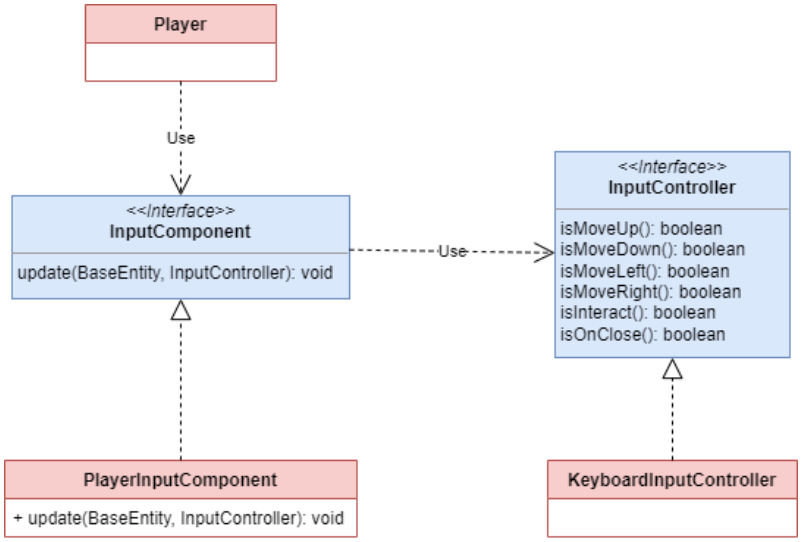

Figura 2.4: Rappresentazione UML della gestione degli input.

HUD di gioco :

Problema: Il gioco necessita di mostrare all'utente alcune informazioni in seguito al verificarsi di determinati eventi.

Soluzione: Per fare ciò si è pensato di generare una nuova entità *Dialog* contenente le informazioni testuali da mostrare a video. Come le altre classi che estendono *BaseEntity* possiede un *GraphicComponent* che permetterà di disegnarla sullo schermo in seguito alla chiamata.

Inoltre, man mano che gli oggetti saranno raccolti, verranno inseriti nel backpack del player e mostrati nel box nell'angolo in alto a destra.

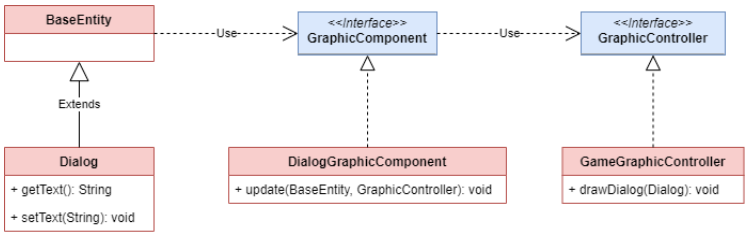

Figura 2.5: Rappresentazione UML della gestione dell'HUD di gioco.

Transport Event :

Problema: Il sistema deve supportare il passaggio da una mappa di gioco all'altra.

Soluzione: Per la gestione del cambio della mappa e del conseguente posizionamento del player, è stato creato un nuovo evento (*TransportEvent*) usato dai blocchi di transito da una mappa all'altra. Quando il *Player* collide con questi blocchi, si scaturisce l'evento che permette di aggiornare la mappa da visualizzare e colloca il player nella posizione corretta.

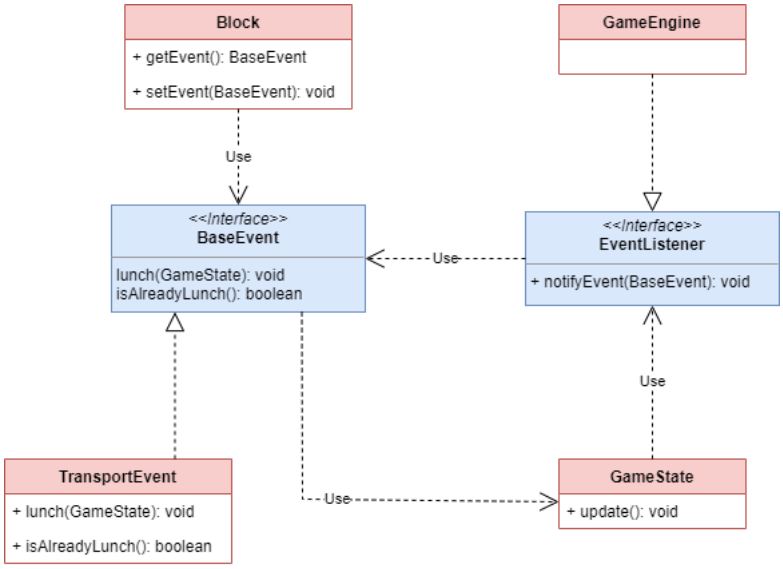

Figura 2.6: Rappresentazione UML della gestione dell'evento di trasporto.

Giulia Montini :

La mia parte di progetto consisteva nella gestione della fisica dei movimenti e delle collisioni, nel salvataggio e caricamento di file e nella creazione del menù di avvio e del menù delle impostazioni.

Gestione della fisica del gioco :

Problema: Il gioco presenta la necessità di mostrare un sistema di fisica dei movimenti effettuati dal giocatore e delle relative collisioni di quest'ultimo con i diversi componenti di gioco.

Soluzione: Il controllo delle collisioni viene effettuato all'interno di World. Se non sono presenti collisioni si procede con l'aggiornamento dei modelli di gioco.

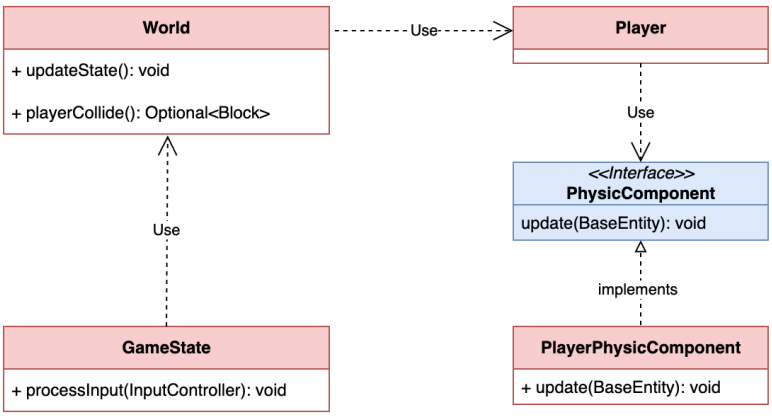

Figura 2.7: Rappresentazione UML della gestione della fisica.

Gestione dei file :

Problema: Il sistema presenta la necessità di scrivere e salvare file.

Soluzione: Per quanto riguarda la gestione dei file si avrà che grazie all'interfaccia *FileSaver* si ha il salvataggio in un file, mentre grazie all'interfaccia *FileLoader* si ha il caricamento da file.

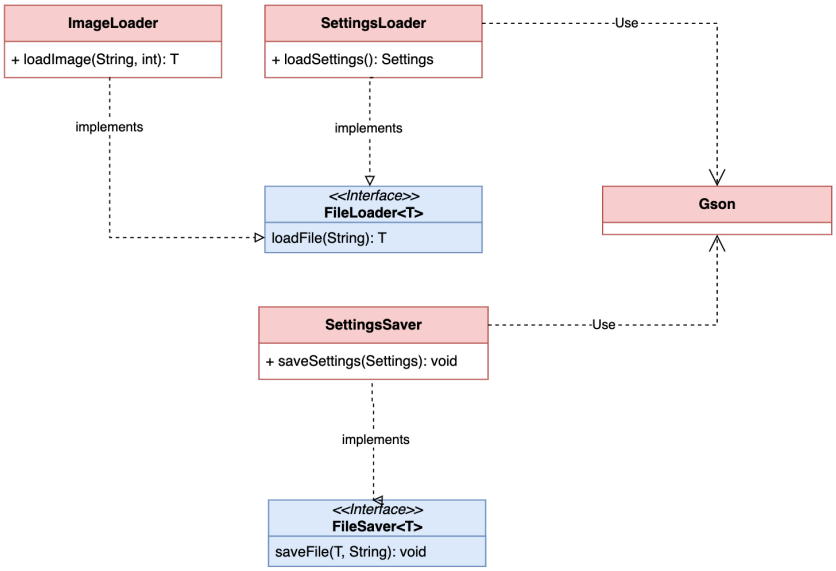

Figura 2.8: Rappresentazione UML della gestione dei file.

Creazione del menù di avvio :

Problema: Il sistema presenta la necessità di mostrare all'utente un menù da cui poter iniziare a giocare.

Soluzione: Con questa soluzione diventa più semplice anche anche gestire l'evento "Fai iniziare il gioco" in quanto si tratta solo di *cambi di stato*.

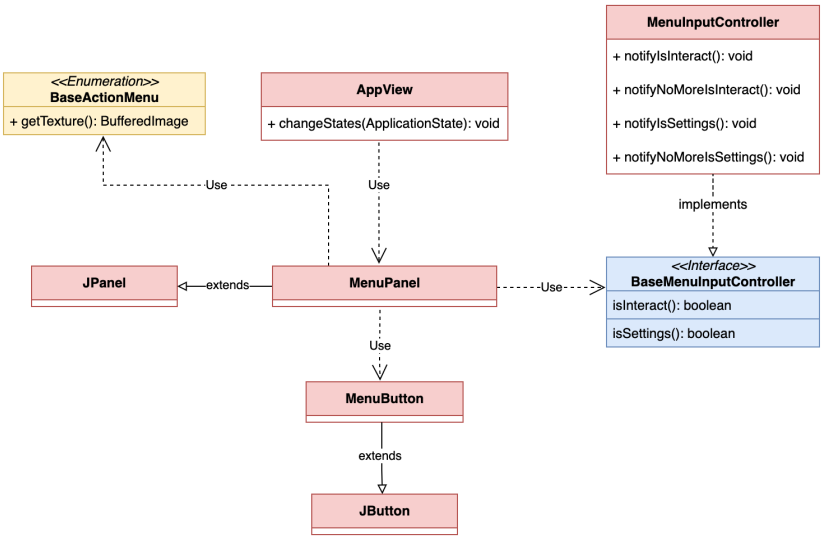

Figura 2.9: Rappresentazione UML del menù principale.

Creazione del menù delle impostazioni :

Problema: Il sistema presenta la necessità di mostrare all'utente un menù in cui poter scegliere le impostazioni di gioco.

Soluzione: Per quanto riguarda la gestione delle impostazioni si è deciso di utilizzare dei file json gestendoli grazie alla libreria *google.gson*.

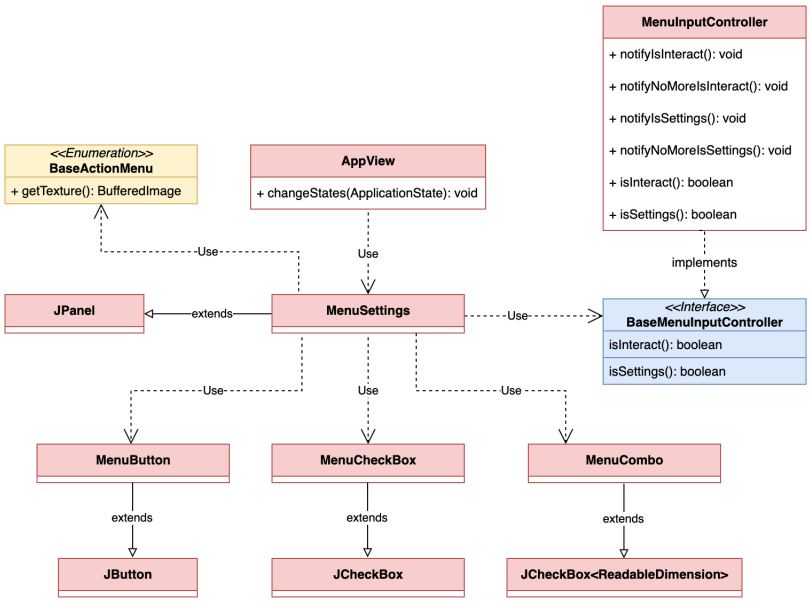

Figura 2.10: Rappresentazione UML del menù delle impostazioni.

Lorenzo Signoretti :

Modelli base :

Problema: Il gioco presenta la necessità di modellare delle entità che rappresentino gli elementi di gioco.

Soluzione: Partendo da questa idea si è creata la classe *BaseEntity*. Questa classe non è istanziabile e contiene gli elementi comuni a tutte le entità del gioco. Si è poi proceduto a modellare le singole entità necessarie al gioco, ognuna di esse contiene caratteristiche uniche che ne giustificano l'esistenza. In alcuni casi si è fatto uso di *enumerazioni* per l'individuazione dei tipi e l'eventuale immagine da disegnare per il proprio tipo.

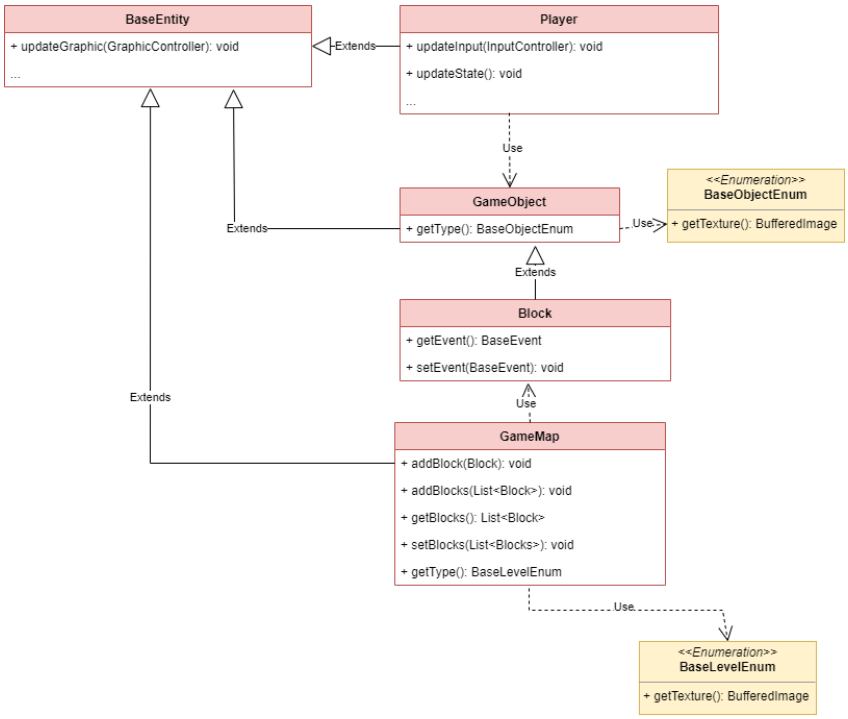

Figura 2.11: Rappresentazione UML della gestione dei modelli.

Rendering del gioco :

Problema: Il gioco deve essere rappresentato anche dal punto di vista grafico.

Soluzione: In accordo con il pattern MVC dovrà essere presente un controller creato, in questo caso, con la classe *BaseView*.

Il controller sfrutterà parzialmente il pattern ***Command***, (anche se il nome delle classi potrebbe far pensare all'utilizzo di *Composite*). *BaseEntity* è il client mentre *GraphicController* e *GraphicComponent* sono rispettivamente Command e Receiver.

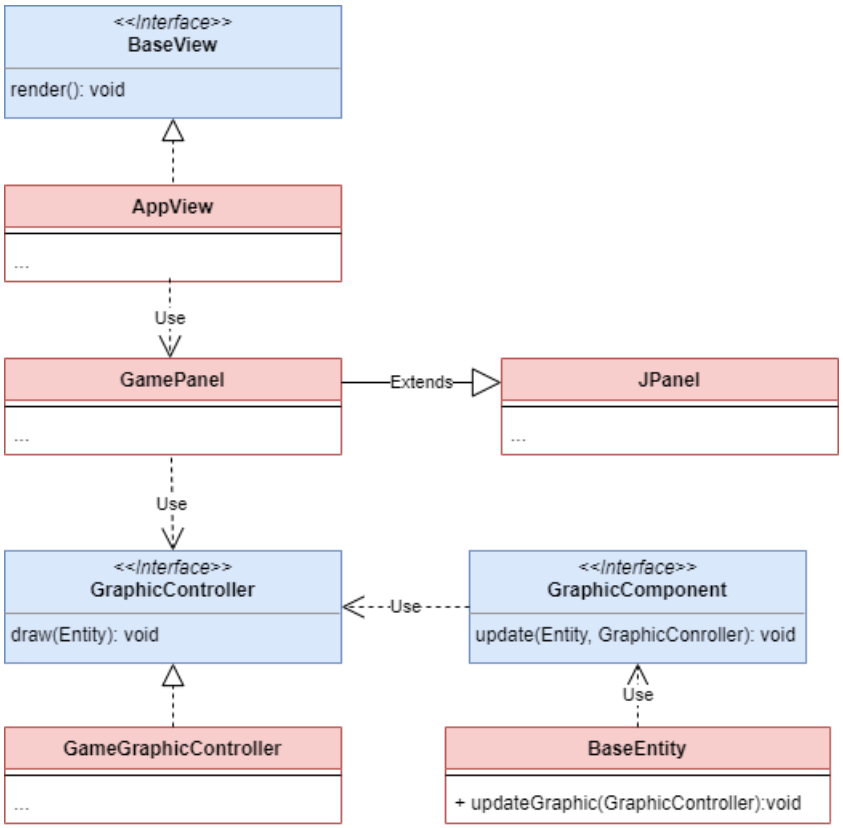

Figura 2.12: Rappresentazione UML della gestione del rendering.

Event :

Problema: Il gioco presenta la necessità di rappresentare degli eventi causati dall'interazione del giocatore con gli elementi di gioco, che possono causare delle modifiche all'interno del gioco.

Soluzione: Utilizzando il pattern ***Observer*** si è creata l'interfaccia ***EventListener*** che rappresenta il *soggetto* del pattern, mentre *BaseEvent* è l'*Observer* oltre che rappresentare l'evento che altererà il gioco.

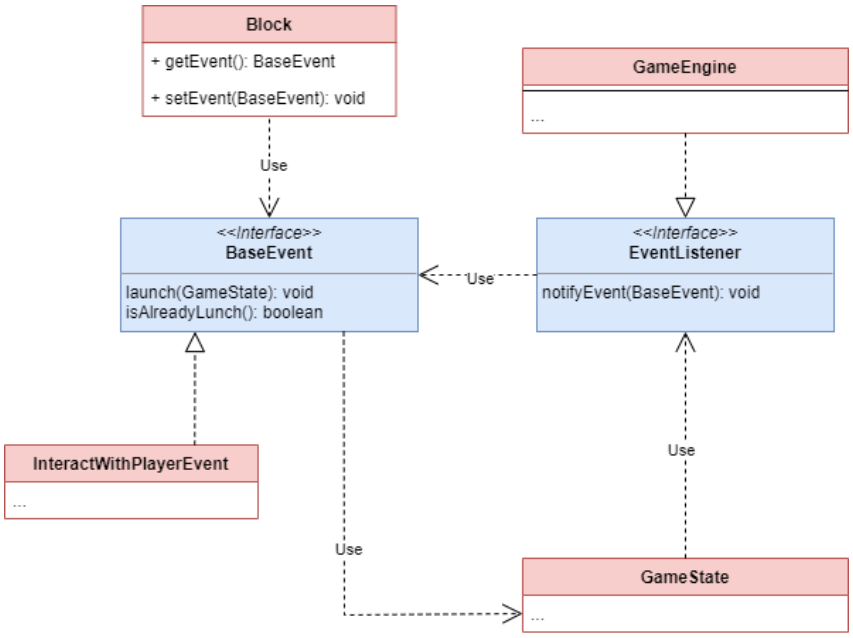

Figura 2.13: Rappresentazione UML della gestione degli eventi di gioco.

GameEngine :

Problema: Il gioco presenta la necessità di aggiornare i modelli di gioco e la corrispettiva grafica, in base all'evoluzione di esso.

Soluzione: Si è optato quindi per l'utilizzo parziale del pattern ***GameLoop***. Abbiamo applicato questo pattern al controller principale del gioco ovvero *GameEngine*, che comunica con i controller di View (*BaseView*) e Model (*GameState*), utilizzando il metodo *setup()* e il metodo *loop()*.

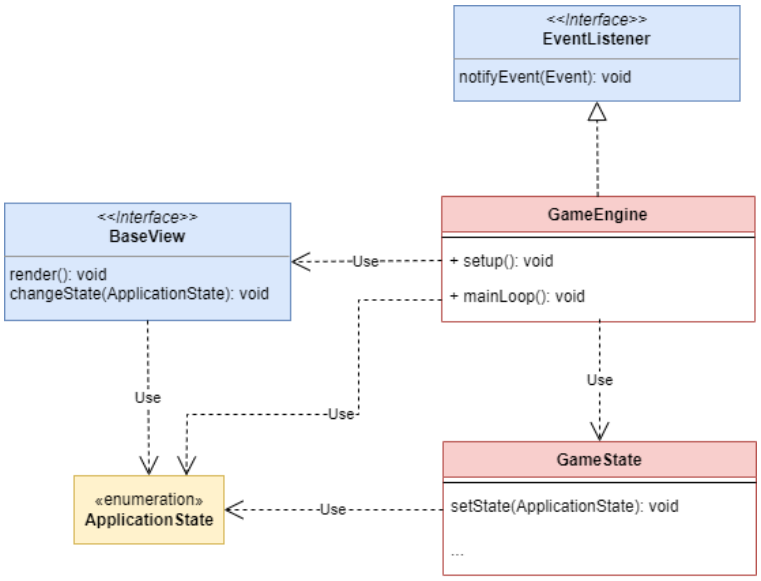

Figura 2.14: Rappresentazione UML della gestione del motore di gioco.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Molte problematiche complesse del gioco sono riscontrabili in casi molto difficili da ricreare in un ambiente automatizzato, per questo abbiamo deciso di effettuare una verifica manuale del corretto funzionamento di alcune funzionalità. Visto l'utilizzo diversificato di sistemi operativi da parte dei componenti del gruppo, è stato possibile controllare con facilità l'effettiva portabilità di nuove funzionalità come, ad esempio, il corretto rendering degli elementi gioco. Ogni componente del gruppo ha proceduto anche a creare alcuni test in JUnit riguardanti funzionalità della propria area di sviluppo:

EventsTest : funzionamento degli eventi, in particolare di *InteractWithPlayerEvent* e dell'effettivo funzionamento dei metodi previsti dall'interfaccia.

InputTest : funzionamento degli input relativi al *Player*; in particolare si controlla la direzione del player dopo aver ricevuto un certo comando.

TransportEventTest : funzionamento dell'evento *TransportEvent*. Si verifica che dopo la chiamata di questo evento la mappa corrente sia stata aggiornata.

CollisionTest : funzionamento della fisica del movimento e delle collisioni. Nello specifico controllo che il movimento sia corretto, ovvero nella direzione giusta, e che quando si ha una collisione il giocatore stia fermo.

3.2 Metodologia di sviluppo

Le fasi di analisi e di sviluppo del design architetturale sono state svolte in gruppo a più riprese. Una delle prime cose fatte è stata quella di realizzare una roadmap delle "versioni" che ci permettesse di lavorare in modo ordinato e preciso ma soprattutto utile per gli altri.

Per realizzare questa roadmap abbiamo individuato vari step del programma di sviluppo in cui il software da noi sviluppato facesse qualcosa secondo le nostre aspettative. Ad esempio, la prima "versione" avrebbe dovuto semplicemente ricevere input e convertirli in movimenti del giocatore oltre che a mostrarlo in qualche modo (il giocatore veniva rappresentato come un semplice rettangolo). Utilizzando questo concetto abbiamo deciso di alternare lo sviluppo del codice alla progettazione del progetto poiché, delineando delle versioni, era possibile partire da uno schema base per poi ampliarlo versione dopo versione.

Le parti inizialmente concordate sono state per lo più rispettate anche se a volte è stato più conveniente lavorare all'interno di classi sviluppate da altri come nel caso dello sviluppo delle collisioni tra giocatore e blocchi.

Dato l'utilizzo di GitHub abbiamo anche deciso di sfruttare alcune funzionalità che il sito ci offriva come, ad esempio, il sistema di issue e pull request. Questo sistema ci ha permesso di organizzare meglio il nostro lavoro, oltre che a sapere in ogni momento cosa si sarebbe dovuto sviluppare successivamente. Abbiamo deciso di sviluppare utilizzando i branch *main*, *develop* e i *feature branch* collegati ad una o più issue e ad una pull request verso il branch develop. Una volta raggiunto l'obbiettivo della nostra roadmap abbiamo utilizzato il sistema di release fornito da GitHub, effettuando un merge dal branch develop verso il branch main; in questo modo è sempre possibile provare vecchie versioni del gioco e osservare lo sviluppo nel corso del tempo.

Alessia Lombardi :

- Creazione del package *Input* per la gestione degli input
- Creazione delle classi *GameFactory*, *GameState*, *World* per la generazione degli elementi di gioco

- Creazione della classe *Dialog* per la visualizzazione di informazioni sulla schermata
- Gestione del passaggio tra mappe
- Gestione dei dialoghi multilinea

Giulia Montini :

- Creazione del package *Physics* per la gestione della fisica del movimento e del package *File* per la gestione dei file
- Gestione delle collisioni
- Creazione delle classi *SettingsLoader* e *SettingsSaver* per la gestione delle impostazioni
- Creazione delle interfacce *FileLoader* e *FileSaver* per la gestione dei file

Alcune soluzioni, come ad esempio il restart dell'applicazione, le ho trovate su StackOverflow.

Lorenzo Signoretti :

- Creazione dei modelli base del gioco
- Creazione del controller principale del gioco
- Creazione della classe *Settings* per la gestione delle impostazioni
- Gestione dei cambi di stato della *View*
- Gestione del ridimensionamento della *View* in base alle impostazioni scelte
- Creazione del controller principale della *View*

L'integrazione tra *GameEngine* e *GameState* è stata pensata insieme ad Alessia Lombardi.

3.3 Note di sviluppo

Alessia Lombardi :

Features avanzate utilizzate:

- Uso di lambda expression

- Uso della libreria google.Gson

Giulia Montini :

Features avanzate utilizzate:

- Uso di lambda expression
- Uso della libreria google.Gson
- Uso di Stream, Optional e altri costrutti funzionali

Lorenzo Signoretti :

Features avanzate utilizzate:

- Uso di lambda expressions
- Uso di Stream, Optional e altri costrutti funzionali

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Alessia Lombardi : Giunta al termine del progetto sono complessivamente soddisfatta del risultato ottenuto, anche se avrei potuto realizzare meglio alcune parti (ad esempio usando maggiormente features avanzate). Ciò nonostante ritengo che il progetto mi sia servito molto dal punto di vista didattico per comprendere meglio le tecniche di programmazione col paradigma a oggetti apprese durante il corso. Lo ritengo inoltre molto utile per un futuro approccio al mondo del lavoro, soprattutto per quanto riguarda lo sviluppo in team.

Il gruppo ha lavorato in un ambiente sereno e costruttivo. Non sono mancate le difficoltà che però sono state superate grazie alla collaborazione e alla comunicazione costante da parte di tutti i membri.

Sono consapevole di avere ancora molto lavoro da fare per migliorare il mio stile di programmazione. Sarebbe interessante continuare il progetto una volta raggiunta una maggiore padronanza del linguaggio, rendendolo più completo (ad esempio aggiungendo altri livelli più complessi, magari con la presenza di alcuni nemici da gestire) e realizzando i requisiti opzionali inizialmente previsti.

Giulia Montini : Lavorare a questo progetto mi ha permesso di comprendere a pieno quali sono gli aspetti dello sviluppo di un lavoro in team. Per quanto riguarda il lavoro svolto mi ritengo

soddisfatta del risultato ottenuto nonostante sia consapevole che si può sempre migliorare.

In ambito lavorativo non svolgo il ruolo di programmatore quindi grazie a questo progetto ho approfondito alcuni aspetti della programmazione ad oggetti che non avrei avuto modo di affrontare; ad esempio, ho trovato interessante l'approccio con i file gson.

A livello di gruppo penso sia stato decisamente efficace il modo in cui abbiamo organizzato e suddiviso il lavoro.

Spesso e volentieri ci siamo dati supporto per la gestione di parti complicate e, in forza di questo, mi sento di poter dire che ognuno di noi ha dato il meglio di sé per la riuscita di questo progetto.

Confrontandoci in questi ultimi giorni è emerso che ci sarebbe piaciuto riuscire a sviluppare anche le parti che avevamo scelto inizialmente come opzionali e non escludiamo di poterlo fare in futuro.

Lorenzo Signoretti : Alla fine di questo progetto mi ritengo parzialmente soddisfatto del lavoro realizzato poiché credo che, a posteriori, avrei potuto sviluppare meglio alcuni concetti come l'utilizzo dei pattern.

Sono molto contento dell'organizzazione del gruppo che ha consentito di lavorare in un ambiente chiaro e collaborativo e, personalmente, mi sono messo in gioco il più possibile cercando di individuare più linee guida possibili.

Il team è stato unito e ci siamo stati di supporto nei momenti più complicati dello sviluppo.

Tutto sommato sono abbastanza soddisfatto del mio lavoro ma, col senno di poi, credo che alcune scelte non le rifarei, questo anche perché il progetto è stato enormemente formativo e mi ho consentito di imparare tanto; perciò la mia visione attuale è diversa da quella iniziale ma almeno mi sento di poter dire di essere un programmatore migliore.

In futuro mi piacerebbe continuare lo sviluppo di questo gioco, apportando le modifiche che farei con l'esperienza acquisita, implementando le funzionalità opzionali che non siamo riusciti a sviluppare e prendendo anche in considerazione l'utiliz-

zo di librerie apposite oltre che un cambiamento di linguaggio utilizzato poiché sono molto intrigato dall'utilizzo di *C++*.

Appendice A

Guida utente

Figura A.1: Menù principale del gioco
Nel menù principale è possibile decidere se iniziare una partita o selezionare le impostazioni.

Figura A.2: Menù delle impostazioni

Il menù delle impostazioni consente di selezionare impostazioni grafiche come la dimensione desiderata. Cliccando sul tasto salva il gioco verrà riavviato e le impostazioni applicate (funzionalità disponibile solo se il gioco è stato eseguito dal .jar).

Figura A.3: Schermata di gioco
Tramite l'utilizzo del tasto invio è possibile concludere un dialogo. L'interazione con gli oggetti di gioco avviene tramite il tasto spazio, mentre con la tipica combinazione di tasti W, A, S, D è possibile muovere il personaggio di gioco.