Relazione di progetto di Programmazione ad Oggetti

Java Wulf

Realizzato da:

Vittorio Damiano Brasini

Luca Nicholae Ferar Tofan

Roberto Sopranzetti

Filippo Velli

Indice

Analisi

Requisiti

Il software che intendiamo realizzare si pone come obiettivo la realizzazione di un videogioco simile a Sabre Wulf del 1984.

Requisiti Funzionali

- Personaggio giocabile che si può muovere in otto direzioni ed attaccare in quattro da una vista top-down del mondo di gioco;
- Telecamera ancorata al personaggio;
- Personaggio giocabile che può recuperare o perdere punti vita raccogliendo oggetti o scontrandosi con i nemici rispettivamente;
- Oggetti e power-up che alterano lo stato del giocatore;
- Nemici con pattern di movimento e comportamenti differenti che possono essere scatenati dalle azioni del giocatore;

Requisiti non Funzionali

- Movimento fluido tra una stanza e l'altra, diversamente da giochi come The Legend of Zelda (1986);
- Frequenza di almeno 30 FPS;
- Classifica composta dai migliori punteggi;

Analisi e modello del dominio

Java-Wulf è un gioco di avventura bidimensionale con vista top-down. Nel gioco si impersonerà un esploratore che deve recuperare nel mondo di gioco i quattro pezzi dell'amuleto che gli consentiranno, una volta trovati tutti e riportati alla zona di partenza, di attivare il portale e di tornare a casa, concludendo la partita.

Il personaggio ha la possibilità di muoversi in 8 direzioni e di attaccare in 4. Per spostarsi in diagonale servirà dunque premere 2 tasti. Il giocatore può attaccare solo quando è fermo. Il raggio d'azione degli attacchi è di una casella davanti all'esploratore. Il personaggio possiede dei punti vita che sono rappresentati nell'interfaccia da dei cuori. Ad inizio partita si partirà con 3 cuori, o altro valore indicato nelle impostazioni dall'utente. Se il giocatore dovesse perdere tutti i punti vita, allora la partita dovrà terminare.

L'ambiente di gioco è costituito da una mappa bidimensionale suddivisa in 4 biomi. Ogni bioma si compone di diverse stanze collegate tra loro da corridoi. Le stanze si trovano in posizioni predefinite e al loro interno si possono trovare potenziamenti utili al giocatore oppure dei pezzi di amuleto. Di questi ultimi, però, se ne può trovare soltanto in una singola stanza per ciascun bioma, per un totale di 4 frammenti secondo tipo ce ne è una per bioma e al suo interno si si trova 1 frammento dell'amuleto. Nel loro punto d'incontro si trova una zona ulteriore, il tempio, dove si trova il portale con cui, una volta raccolti tutti e 4 i frammenti dell'amuleto si concluderà la partita. In questa area centrale i nemici non possono entrare.

Nel gioco si potranno ottenere 2 tipi di potenziamento: gli oggetti e i power-up. Ogni strumento conferisce al giocatore un determinato ammontare di punti all'ottenimento.

Gli oggetti sono dei potenziamenti di durata infinita, o permanenti o di usi limitati. Gli oggetti all'interno del gioco sono:

- Spadone: aumenta la gittata d'attacco e permette di uccidere le guardie e di stordire il lupo;
- Cuore: aumenta i punti vita massimi di un'unità;
- Ampolla: rigenera un cuore perso;
- Super ampolla: rigenera tutta la vita dell'esploratore;
- Scudo: difende l'esploratore da 2 colpi dei nemici;

I pezzi di amuleto sono anch'essi degli oggetti e possiedono la caratteristica di far cambiare colore all'esploratore se dovesse trovarsi nello stesso asse di uno di essi. Questo permetterà al giocatore di intuire la posizione del frammento.

Invece, i power-up sono dei potenziamenti di durata limitata e sono rappresentati da fiori di colori diversi in base ai loro effetti, che sono:

- Verde: raddoppia la velocità del giocatore;
- Giallo: moltiplica i punti guadagnati dal giocatore;
- Rosso: aumenta la forza d'attacco permettendo di uccidere le guardie;
- Blu: intangibilità;

L'ultimo power-up ottenuto è quello in vigore, sovrascrivendo eventuali power-up precedentemente ottenuti.

Nel mondo di gioco vi sono inoltre dei nemici, unità ostili al giocatore, che al contatto col esso gli arrecheranno danno. I nemici possono muoversi in solo 4 direzioni. Vi sono due tipi di nemici:

- I pedoni: appaiono nei corridoi, si muovono in direzione casuale. Si possono uccidere con 1 colpo di spada;
- Le guardie: normalmente immobili, appena il giocatore si troverà nella loro stessa stanza lo inizieranno ad inseguire. Possono essere stordite con un colpo di spada, ma necessitano dello spadone o degli effetti del fiore rosso per essere sconfitte;

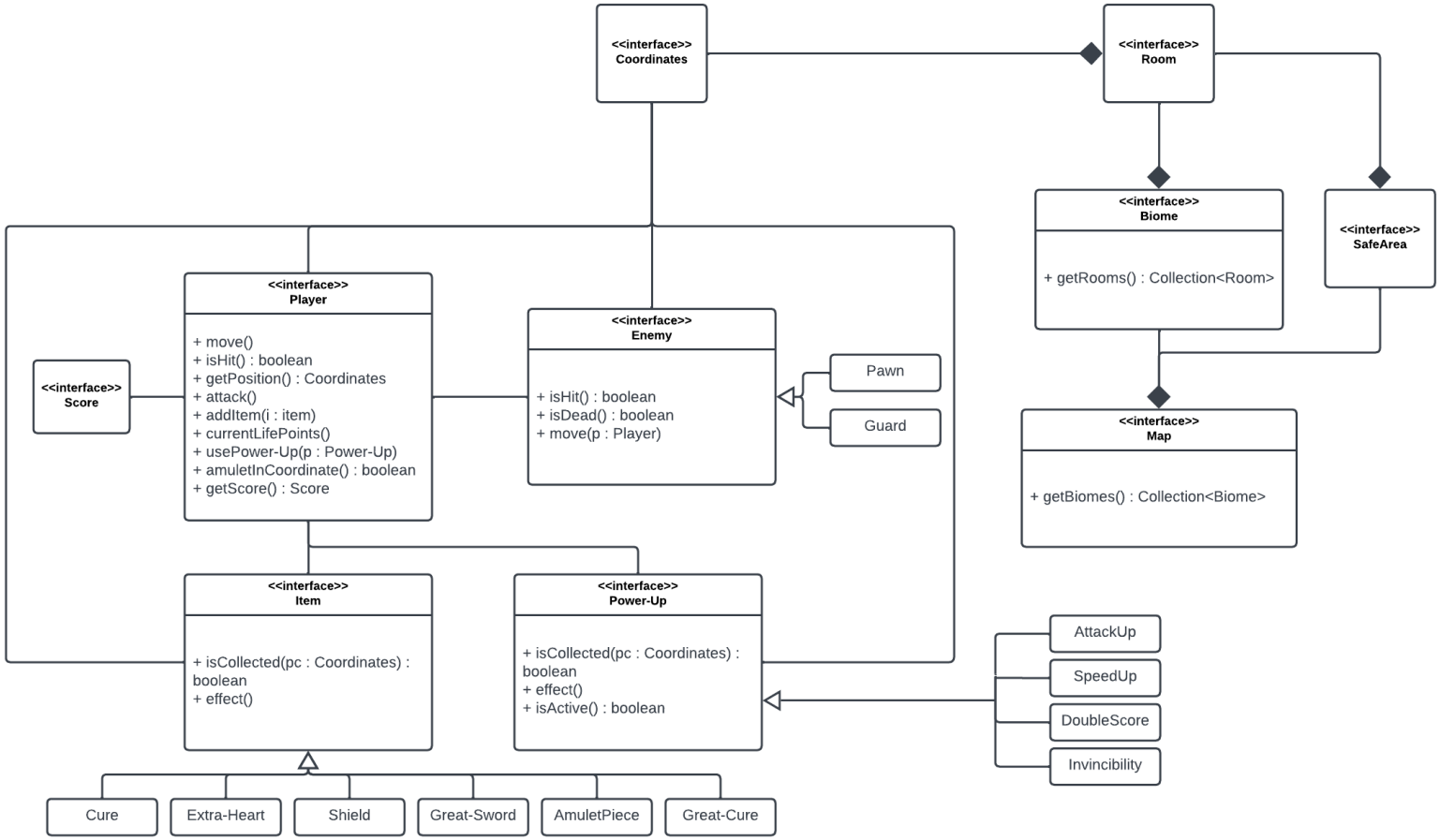


Figura 1.1: Schema UML derivato dall'analisi del dominio, con la rappresentazione delle entità principali ed i loro rapporti

Design

Architettura

L'applicazione adotta come pattern architetturale MVC, dove il Controller funge da tramite tra l'input dell'utente via View e la logica del gioco che è incapsulata all'interno del Model.

Il Model ha come suo punto di ingresso la classe Map che si occupa di restituire gli elementi di gioco che ha creato all'interno del package Model.

Il Controller è caratterizzato principalmente dalla classe GameLoop che si occupa di:

- aggiornare lo stato di gioco comunicando alla View lo stato corrente che ha prelevato dal Model per poter rappresentare il mondo di gioco;
- apportare cambiamenti al gioco dovuti all'input dell'utente derivante dalla View.

Invece, la View, che ha come punto di ingresso GameMenuPanel, che si occupa di mostrare il menu di partenza da cui poi si può o lanciare la partita, o vedere la classifica dei migliori punteggi, o vedere una piccola guida sui comandi o uscire dall'applicazione.

Una volta avvivata la partita si usa GamePanel per interagire col gioco e per disegnare tramite dei Drawer i vari elementi di gioco.

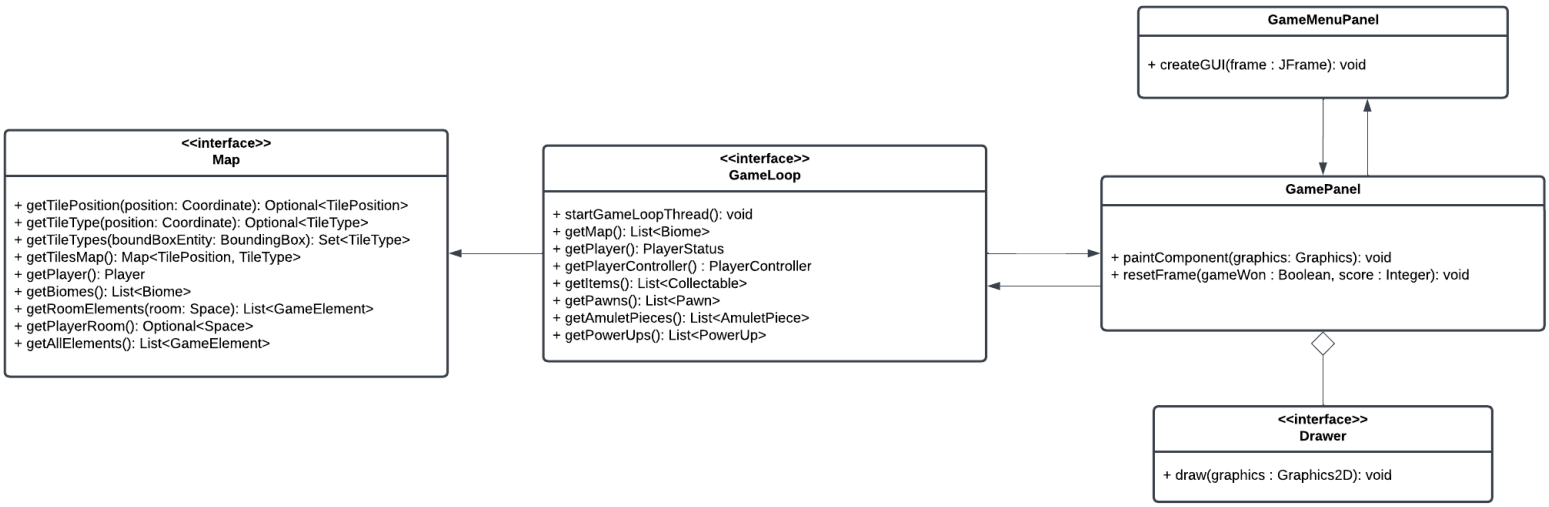


Figura 2.1: Schema UML dell'architettura dell'applicazione Java-Wulf

Design Dettagliato

Vittorio Damiano Brasini

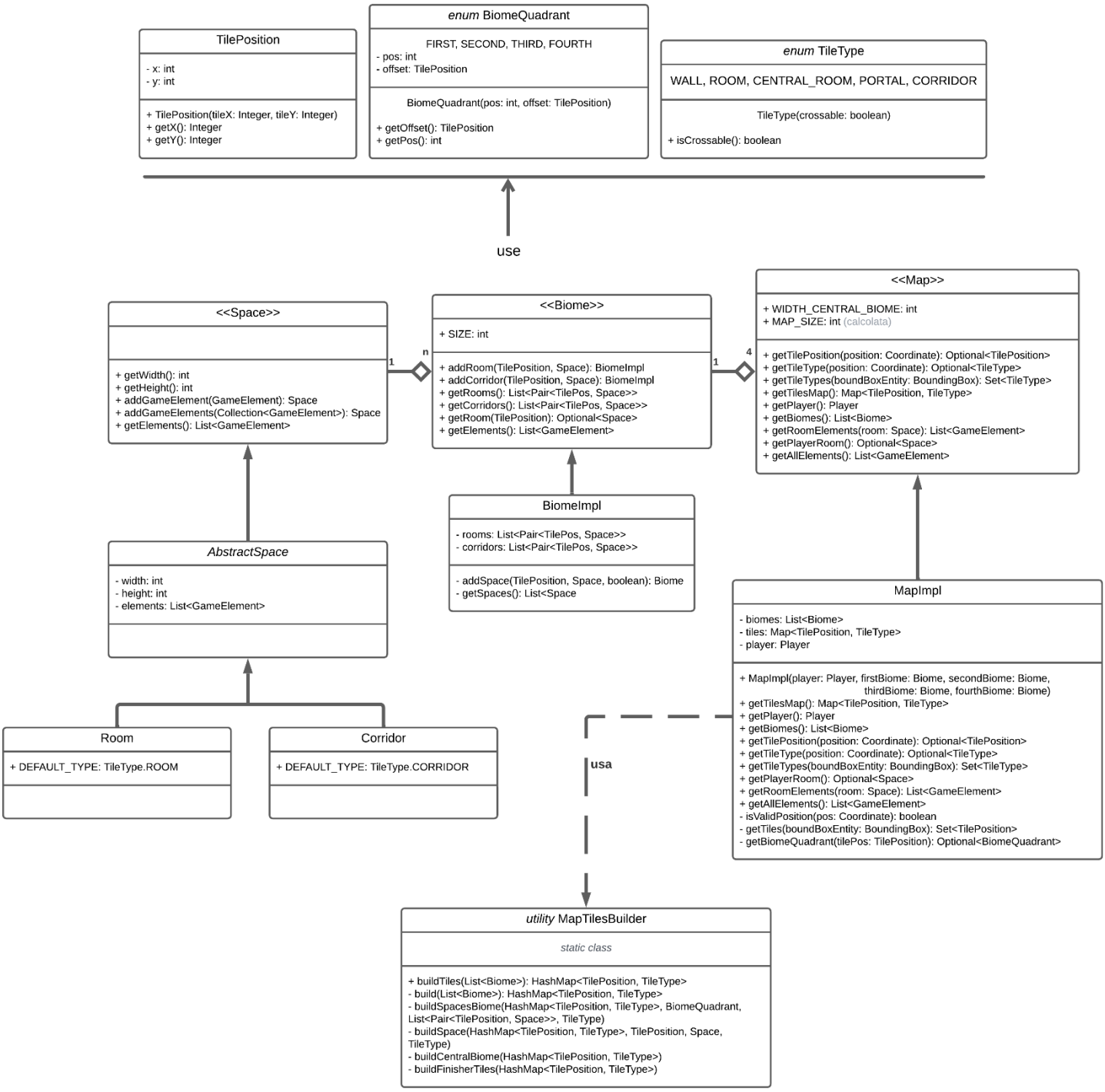


Figura 2.2.1.1: Qui di seguito il diagramma implementativo che sta alla base architetturale dell'infrastruttura "Mappa" (nonché dell'ambiente di gioco).

La Mappa: architettura generale

Per la realizzazione di quanto specificato nella sezione di analisi, ho adottato una progettazione di tipo “aggregativo” nel quale:

- Gli Spazi sono contenuti nei Biomi; si suddividono principalmente in Stanze e Corridoi. Oltre ad avere intrinseche caratteristiche (larghezza ed altezza), essi sono anche contenitori di GameElements (entità nemiche, oggetti e power-up).
- I Biomi sono *contenitori* di Spazi. A ciascuno spazio aggiunto nel Bioma, viene associata una posizione (che è relativa al Bioma stesso)

A scopo esemplificativo: risultato desiderato della Mappa. I quadrati neri sono i Biomi, quelli grigi sono le Stanze, mentre i rettangoli arancioni sono i corridoi.

- La Mappa è a sua volta *contenitore* di 4 Biomi e del giocatore. L’unico dettaglio che non si evince dal diagramma qui sopra riportato è la disposizione che concettualmente hanno i 4 biomi: in senso orario, disposti come se fossero i quadranti di un piano cartesiano (dal primo in alto a sinistra, all’ultimo in basso a sinistra).

  Opportuni metodi permettono di ottenere informazioni di tutti i componenti di cui la Mappa è composta, in particolare l’interfaccia di Mappa consente: di ottenere la stanza in cui si trova il giocatore, di ottenere tutti i GameElements di una specifica stanza, di ottenere una lista complessiva di tutti i GameElement presenti nella Mappa. Certe entità potrebbero aver bisogno di modificarsi qualora il giocatore entri nella stanza in cui esse sono contenute.

Le Tile

Per risolvere e semplificare il problema di come costruire la Mappa, ho scelto di adottare

un’architettura a “piastrelle”: ciascuna piastrella è elemento atomico della mappa (un “mattoncino”); esse hanno una misura di default (es. 24x24 pixel) e un tipo, definito da TileType. I tipi di piastrelle si contraddistinguono anzitutto per essere calpestabili o meno. Le entità di gioco (tra cui il giocatore) sfrutteranno un metodo dell’interfaccia di Map per essere al corrente se sono in prossimità di un muro o di qualche altra tipologia di piastrella (vedi il metodo Map.getTileTypes()).

Tile Position

Il posizionamento delle Tile su spazio bidimensionale è mantenuto mascherato ed è utilizzato unicamente all’interno del Map *package*: la stessa mappa si occupa di convertire le coordinate (x, y) in pixel con le relative coordinate (x, y) in Tile (TilePosition).
Anche il posizionamento degli Spazi all’interno dei Biomi è definito da una posizione (x, y) in Tile: si noti il problema nascente di ottenere la posizione assoluta (rispetto l’intera Mappa) di una Stanza qualunque; problema che ho scelto di risolvere definendo l’enum. BiomeQuadrant, il quale restituisce, per ciascuna posizione del Bioma (primo, secondo, terzo, e quarto quadrante), un *offset* da applicare alla posizione (x,y) salvata per quel determinato Spazio all’interno del Bioma.
La mappa viene costruita su di una HashMap a partire dai Biomi passati nel costruttore: viene utilizzata la classe di *utility* MapTilesBuilder, che associa a ciascuna posizione in Tile (x, y) un determinato tipo di piastrella (ad es. **(1, 0) -> CORRIDOR**, **(1,1) -> ROOM**). Posizioni all’interno della Mappa mancanti sono da considerarsi muri (se la posizione **(3,4)** non è presente tra le chiavi della HashMap, è implicitamente presente una piastrella di tipo WALL).
Il beneficio di avere una HashMap contenente tutti i tipi di Tile per ciascuna posizione è utile soprattutto ad avere una efficiente struttura dati da consultare internamente per il rilevamento delle piastrelle che coincidono con il boundingbox di una entità; non si ignora il vantaggio di avere una rapida struttura dati completa e già scorporata di inutili dettagli implementativi, e che quindi può facilmente essere condivisa alla View.
Popolamento con le entità di gioco
Dal momento che si è deciso, per scelta di design, di incorporare dentro la Mappa tutti gli elementi di gioco, ho scelto di risolvere il problema del loro posizionamento “popolando” la Mappa solo e successivamente alla sua inizializzazione. È quindi la classe d’*utility*

Populator che, tenendo conto del posizionamento dei biomi e del loro offset, inserisce nemici ed oggetti stanza-per-stanza (ciò non sarebbe stato possibile prima del posizionamento dei Biomi all'interno della mappa, poiché non si conosce ancora in quale quadrante essi troveranno, con il conseguente dubbio che non si possono conoscere le coordinate assolute di un qualunque gameElement si volesse posizionare anticipatamente all'interno della mappa stessa).

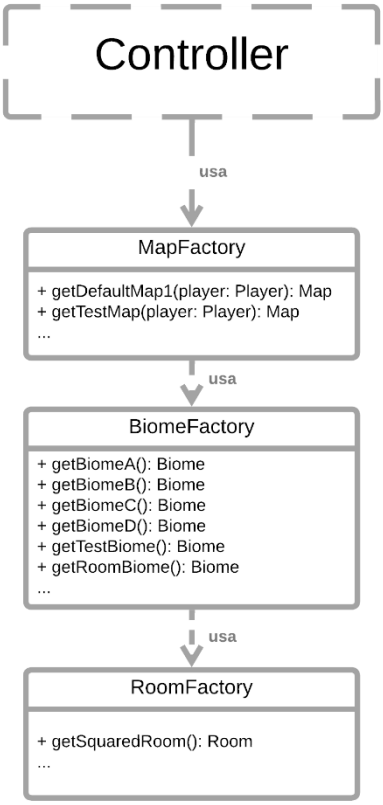
Rappresentazione grafica della Mappa

Si tenga presente che il completo *enviroment* di gioco, durante la partita, viene disegnato all'interno di un JPanel, in particolare in una sua sotto-estensione (che abbiamo nominato classe GamePanel).
Allo scopo di disegnare la Mappa in un contesto in cui il giocatore deve sempre rimanere al centro dello schermo, la difficoltà è stata quella di doverla rappresentare come se fosse un tappeto "scorrevole" sottostante al giocatore, che viene spostato ogni qual volta il giocatore stesso effettua un movimento. Intuibile come il disegnamento delle Tile dipenda dalla posizione assoluta del *player*. Non serve disegnare tutte le Tile della mappa, ma solo quello appresso al giocatore. Questa logica è stata scorporata dal pannello di gioco in una classe separata denominata "MapDrawer", che si serve unicamente della HashMap sopra citata (il concetto di *Drawer* è stato poi intelligentemente astratto da Filippo in una interfaccia generica, utilizzata poi da tutti i componenti del gruppo di progetto come strumento per disegnare i vari componenti di gioco).

Considerazioni sui Pattern noti utilizzati

Non segnalo l'utilizzo di alcun particolare Design Pattern, tal più qualche accenno:

- MapTilesBuilder è una classe di utility somigliante al *Builder* pattern design; ipoteticamente potrebbe esporre diverse interfacce di ingresso per la *build* della mappa (ad esempio la possibilità di costruire la mappa anche da file); mi sono limitato però ad esporre un'unica interfaccia che *builda* la mappa a partire dai quattro biomi, servendosi di un riferimento alla mappa stessa.



- Diverse factory contenute nel package *javawulf.map.factory*, e rispettivamente una per ciascun elemento modellato nella mia parte del progetto: uno per la Mappa, uno per i singoli Biomi e uno per le singole stanze. Suddette factory forniscono istanze di oggetti correlati tra loro, usandosi a cascata l'un l'altro (MapFactory >> BiomeFactory >> RoomFactory). Sono quindi riconciliabili a *Factory Method*: si faccia esempio al MapFactory, che fornisce un metodo per la generazione di una Mappa di *default* (nonché quella utilizzata alla consegna del progetto) e un metodo per la generazione di una Mappa di *testing*.

Gestione del tempo di gioco: il Game Loop

Uno dei primissimi problemi da affrontare è stato sicuramente quello del tempo. Per scandire il tempo di gioco abbiamo utilizzato un Loop, contenuto nel controller. Esso utilizza un ciclo di esecuzione continuo per aggiornare lo stato del gioco e impartire alla View il disegnamento del frame successivo sulla schermata. Il ciclo di gioco viene eseguito in un thread separato per mantenere l'interattività del gioco.

javawulf.controller.GameLoopImpl

*Ambiente di gioco e Mappa sono sinonimi; Tile e Piastrella sono sinonimi.

Luca Nicholae Ferar Tofan

Power Ups

Problema: Preoccuparsi della gestione dei power Ups per essere raccolti e modificare le statistiche e colore del player solo nel tempo richiesto ed evitare che più power Up siano attivi nello stesso momento.

Soluzione: Ho optato per l'utilizzo di un handler che facesse da tramite tra il player e il power Up raccolto, ho quindi utilizzato il **pattern Template method**, che accomuni la parte comune ai power Ups per terminare l'effetto e rendere autonomo per ogni tipo di power up il reset dell'effetto, in modo poi da sovrascrivere all'interno del handler il vecchio power Up con quello raccolto.

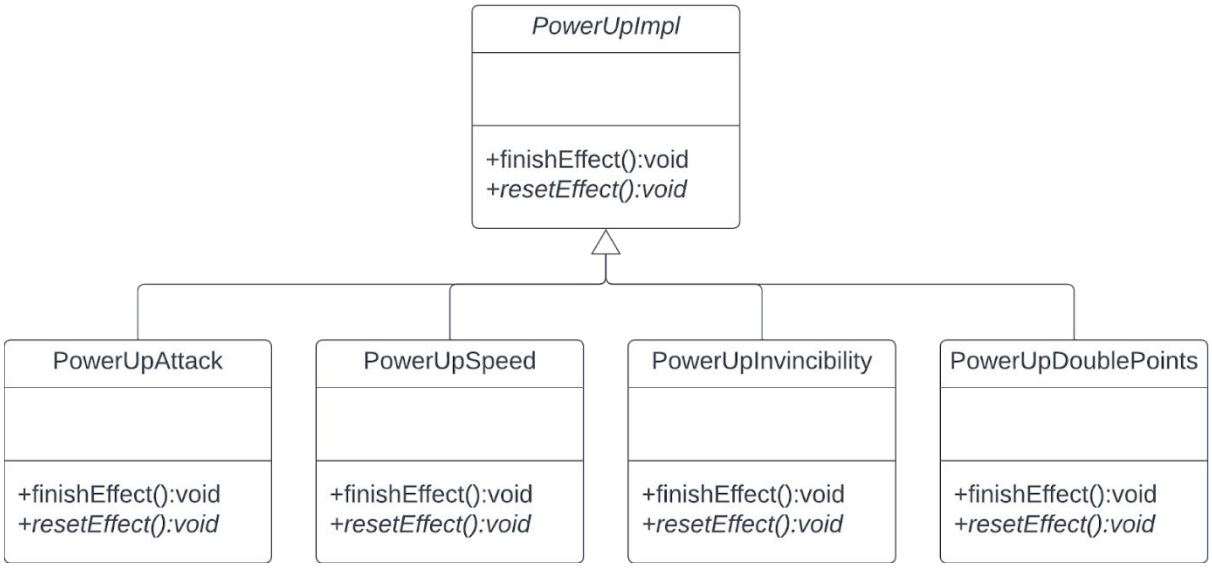


Figura 2.2.2.1: Schema UML dell'implementazione dei PowerUp

Problema: Preoccuparsi della creazione di power Ups uguali per: durata, punteggio e tipo ma non per coordinate in modo da permettere la popolazione della mappa dei power Ups in coordinate diverse tra loro, senza dover mettere sempre gli stessi parametri.

Soluzione: Mediante l'utilizzo del **Pattern Factory**, la quale in base al Power Up richiesto e la coordinata in cui lo si vuole, si occuperà di restituire l'oggetto.

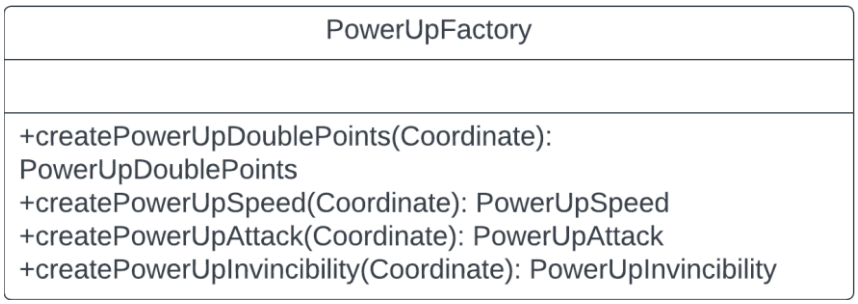


Figura 2.2.2.2: Schema UML dell'implementazione della PowerUpFactory

Classifica

Problema: Salvataggio e Lettura del file Contenente la classifica, il modo in cui il file veniva inizialmente salvato era di difficile lettura e mal formattato.

Soluzione: Questo è stato risolto formattando il file linea per linea con i vari risultati utilizzando il charset UTF-8, per la lettura invece si fa affidamento a convertInResult, che tramite manipolazione della stringa in ingresso è in grado di restituire un Result, pronto ad essere aggiunto alla Classica.

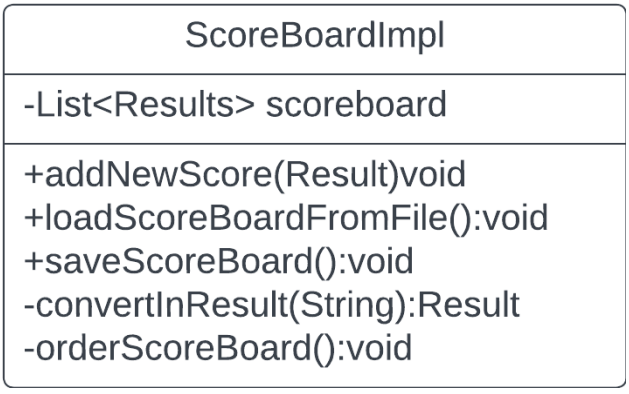


Figura 2.2.2.3: Schema UML dell'implementazione della ScoreBoardImpl

Roberto Sopranzetti

Generazione dei nemici

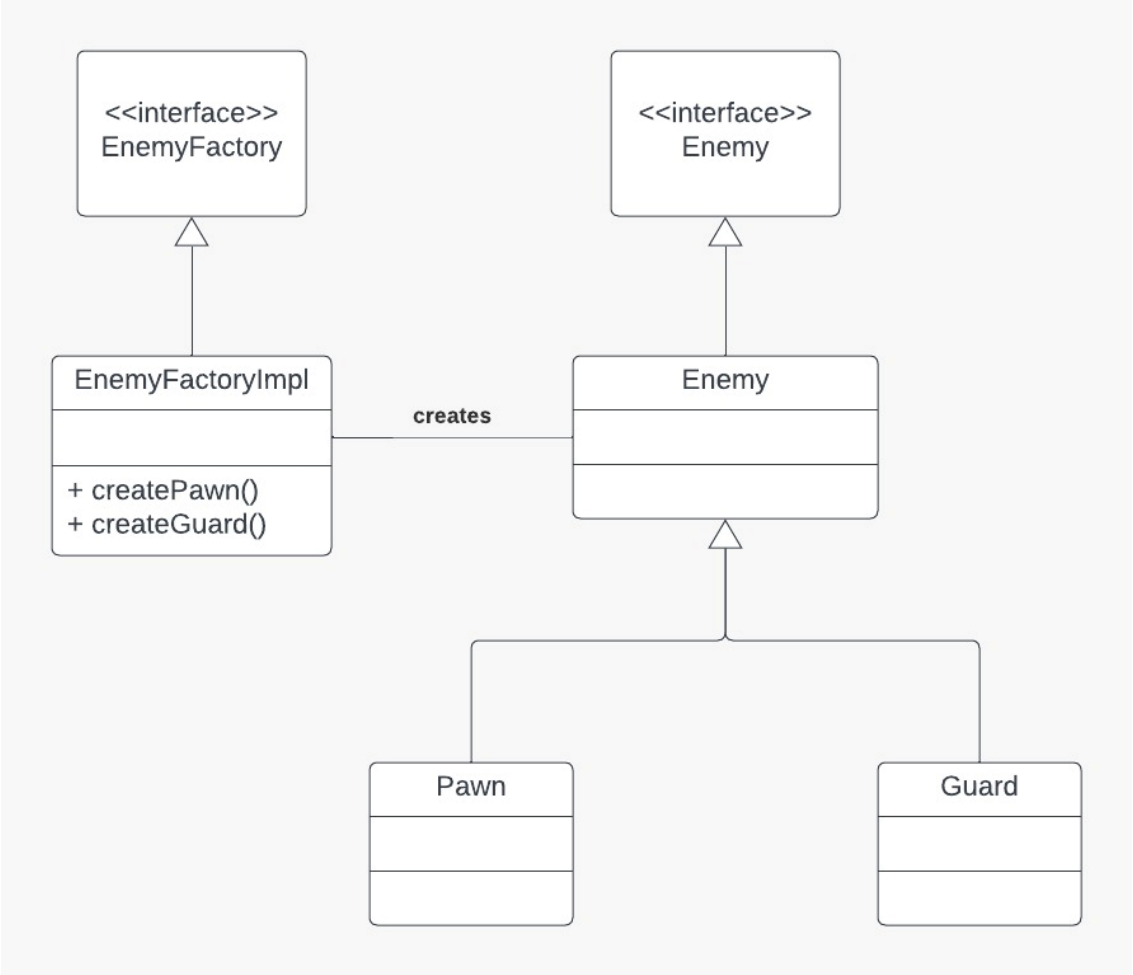


Figura 2.2.3.1: Schema UML per l'implementazione di EnemyFactory che permette di creare automaticamente i nemici

PROBLEMA: Durante la fase di popolazione del gioco, ci si è resi conto che la creazione manuale dei nemici richiedeva l'importazione dell'intero package dei nemici e l'istanziazione esplicita di ciascun tipo di nemico, aumentando la complessità e riducendo l'incapsulamento.

SOLUZIONE: Per affrontare questo problema e migliorare l'incapsulamento, ho adottato il pattern Factory Method. Questo mi ha consentito di automatizzare la creazione dei nemici senza la necessità di importare l'intero package. Ora, il populator può richiedere un nemico attraverso l'interfaccia del Factory Method, senza dover conoscere i dettagli implementativi specifici di ciascuna classe di nemico. In questo modo, ho semplificato la gestione e l'estensione della creazione dei nemici nel corso dello sviluppo del gioco.

Creazione degli oggetti

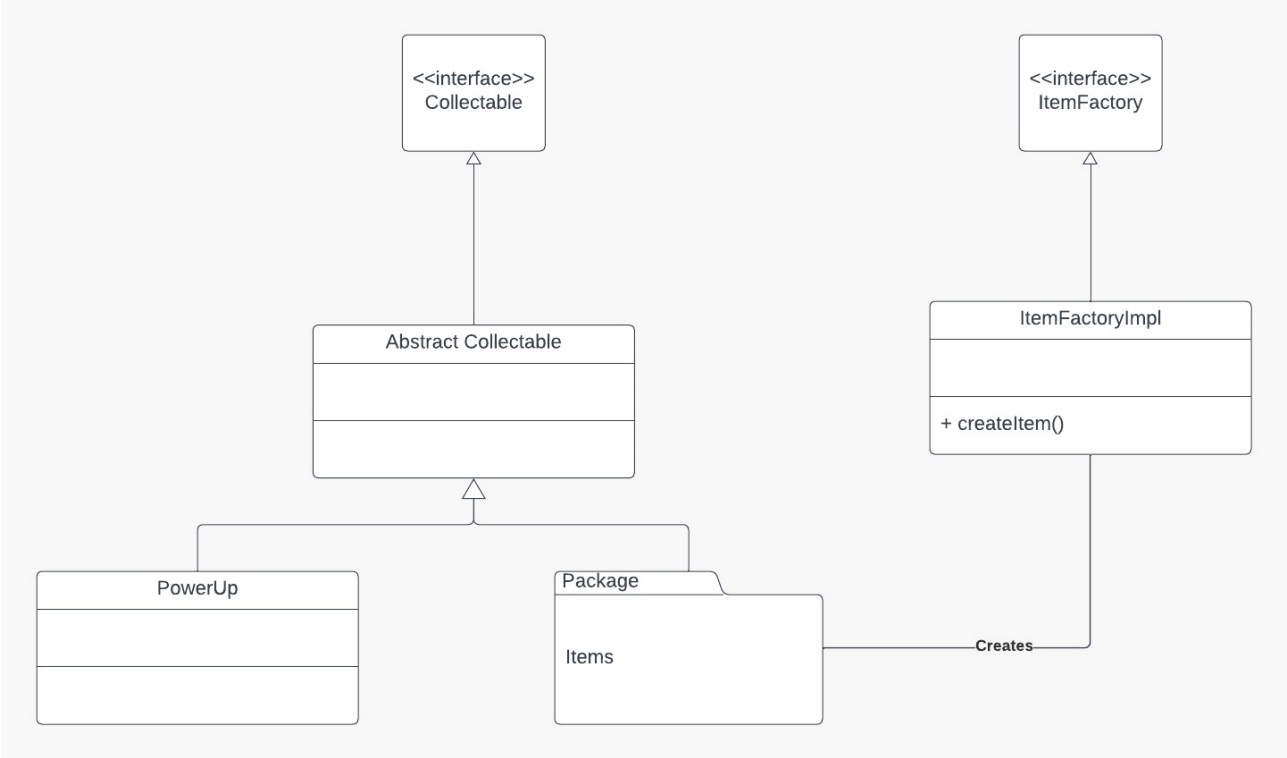


Figura 2.2.3.2: Schema UML dell'implementazione della ItemFactory

PROBLEMA: Nel processo di implementazione degli oggetti raccoglibili nel gioco, mi sono trovato di fronte alla stessa sfida riscontrata con i nemici: la gestione manuale della creazione di ciascun oggetto. Questo ha portato all'importazione dell'intero package degli oggetti raccoglibili e all'istanziazione esplicita di ogni tipo di oggetto, generando complessità e violando l'incapsulamento.

SOLUZIONE: Per affrontare questa problematica in modo efficace, ho esteso l'utilizzo del pattern Factory Method, già impiegato con successo per i nemici. Tuttavia, è importante notare che, nonostante la similitudine concettuale, le esigenze specifiche degli oggetti raccoglibili hanno richiesto un nuovo Factory Method, garantendo una gestione appropriata della creazione di tali elementi nel contesto del gioco.

Riuso codice dei raccoglibili

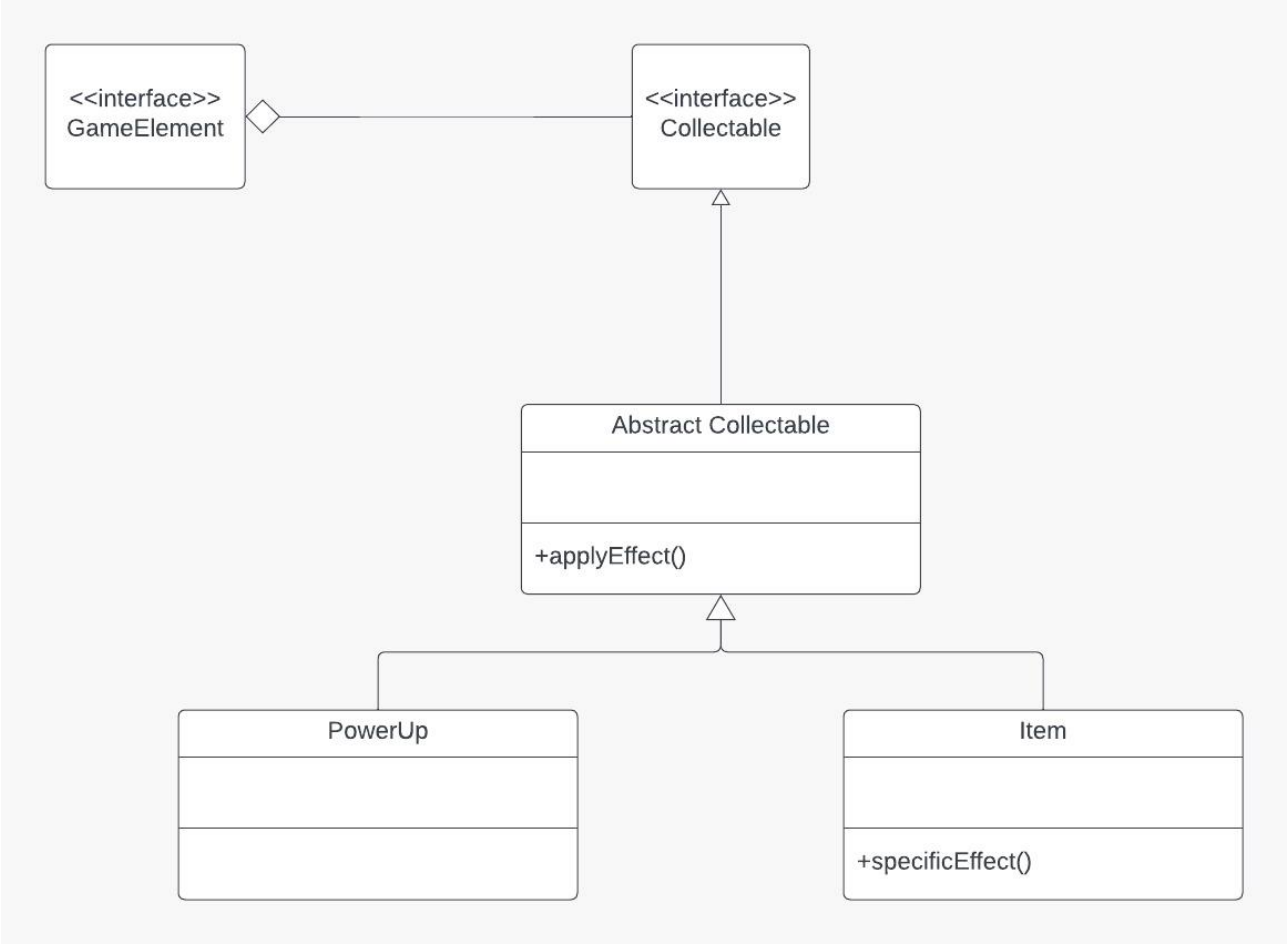


Figura 2.2.3.3: Schema UML per l'applicazione del pattern Template Method per la creazione degli oggetti e power-up

PROBLEMA: in fase di sviluppo mi sono accorto che tutti quanti i raccoglibili avevano in comune i loro attributi ed il modo in cui venivano raccolti, creando di fatto classi praticamente uguali tra di loro con solo poche modifiche.
SOLUZIONE: Poiché le uniche differenze erano metodi aggiuntivi, oppure l'effetto che causavano al giocatore ho pensato di usare il pattern Template Method creando AbstractCollectable per poter massimizzare il riuso. Il metodo astratto è applyEffect() che viene sovrascritto da ciascuna sottoclasse di AbstractCollectable.

Filippo Velli

Responsabilità del giocatore

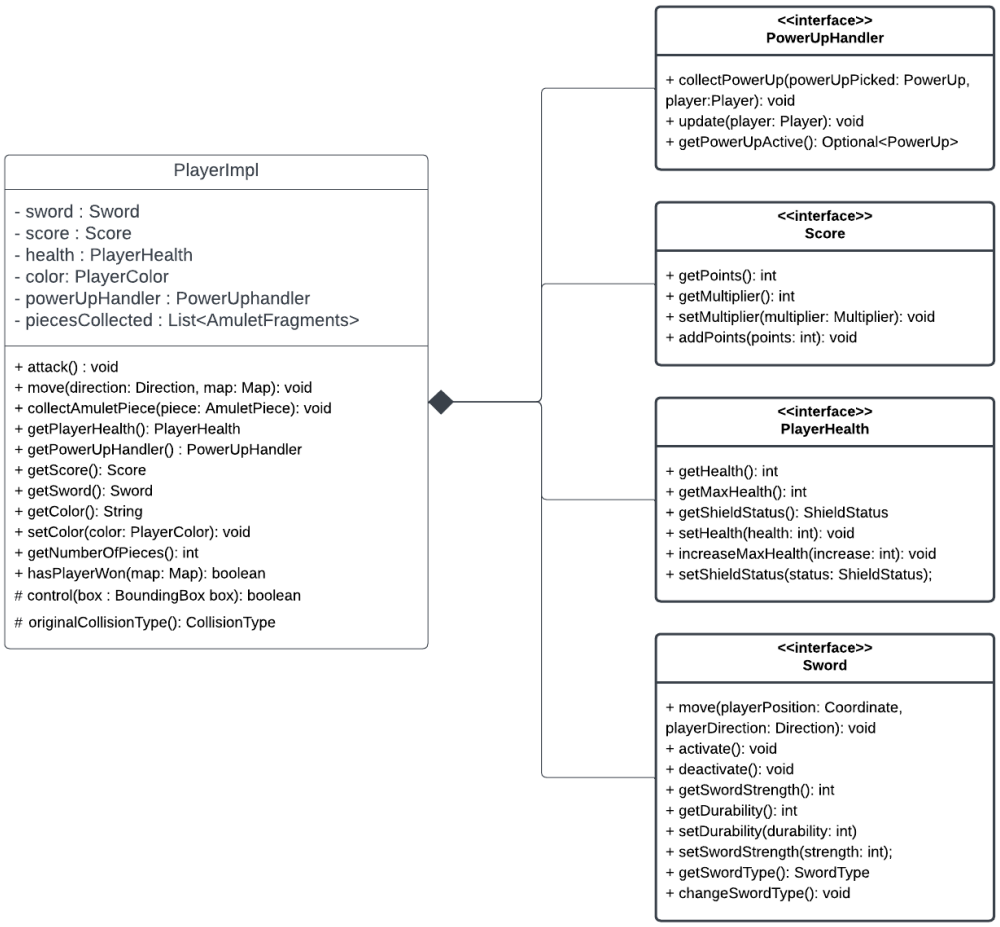


Figura 2.2.4.1: Schema UML per la gestione delle responsabilità del giocatore

PROBLEMA: il giocatore si ritrova con molteplici responsabilità diverse, tra cui la gestione dell'attacco, dei suoi punti vita, del punteggio e del power-up che ha attivo al momento.

SOLUZIONE: ho applicato il Single Responsability Principle per delegare funzionalità più specializzate a delle altre classi, ovvero Sword per l'attacco, PlayerHealth per la gestione sua salute e dello scudo attivo, Score per il punteggio e il suo incremento e PowerUpHandler, che è stato realizzato da Luca, che gestisce il power-up che possiede correntemente il giocatore applicandone gli effetti.

Gestione delle collisioni tra giocatore e nemici e della durata dell'invulnerabilità

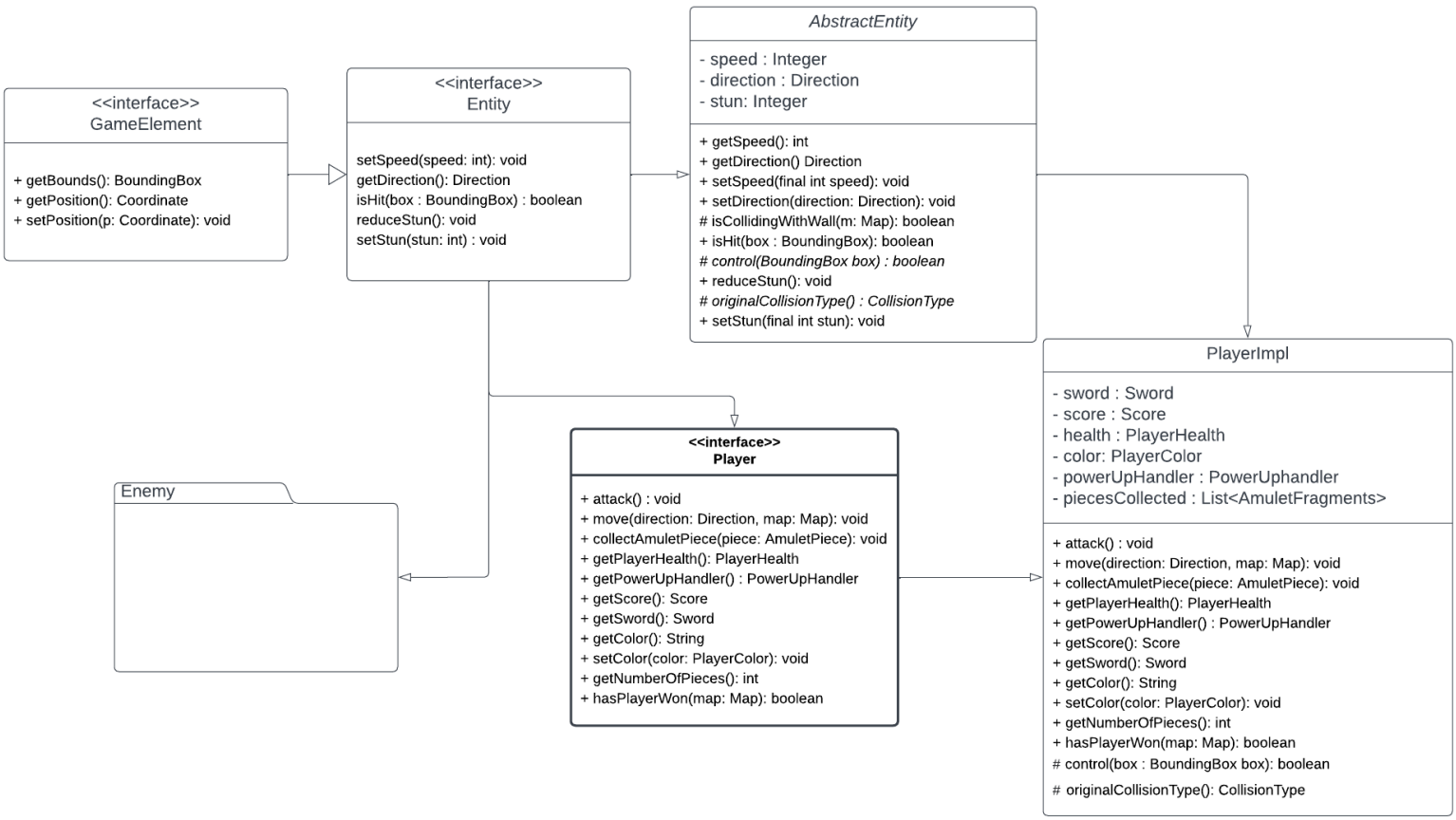


Figura 2.2.4.2: Schema UML per l'adattamento di Template Method all'interno di AbstractEntity e delle sue sottoclassi

PROBLEMA: sia i nemici che il giocatore all'interno del modello hanno bisogno sapere se hanno interagito i primi con la spada del giocatore e il secondo con i nemici stessi per poter determinare il danno. Inoltre, c'è la necessità di gestire l'attivazione del periodo di invulnerabilità con la disattivazione della collisione e il suo ripristino alla fine di questo lasso di tempo.

SOLUZIONE: si è utilizzato il pattern Template Method all'interno della classe astratta AbstractEntity che, partendo dal metodo isHit inizialmente realizzato da Roberto e poi da me espanso, permette alle sottoclassi di specificare la logica da applicare allo scheletro degli algoritmi.

Comunicazione dello stato del giocatore

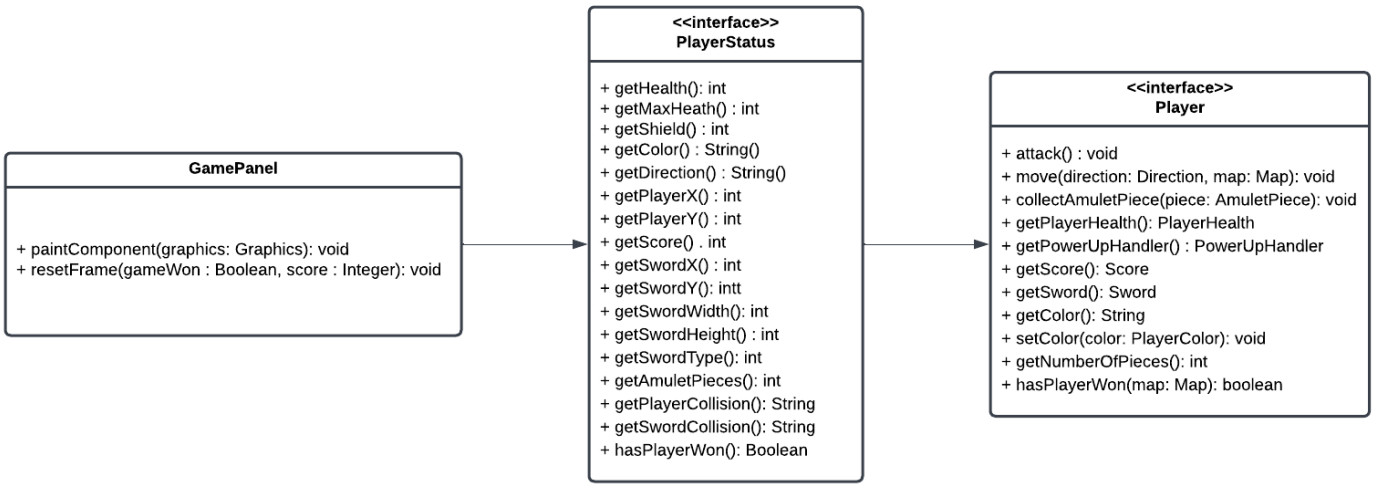


Figura 2.2.4.3: Schema UML che illustra l'uso di Adapter con PlayerStatus

PROBLEMA: si deve comunicare lo stato del giocatore senza fornire metodi che lo modifichino alla View per permettere una corretta rappresentazione nel mondo di gioco sia del personaggio che della HUD.

SOLUZIONE: si è utilizzato il pattern Adapter per creare un tramite PlayerStatus che non solo non riveli l'interfaccia di Player all'esterno ma fornisca i metodi attesi dalla View per poter disegnare correttamente la schermata di gioco.

Comunicazione dei comandi da tastiera al gioco

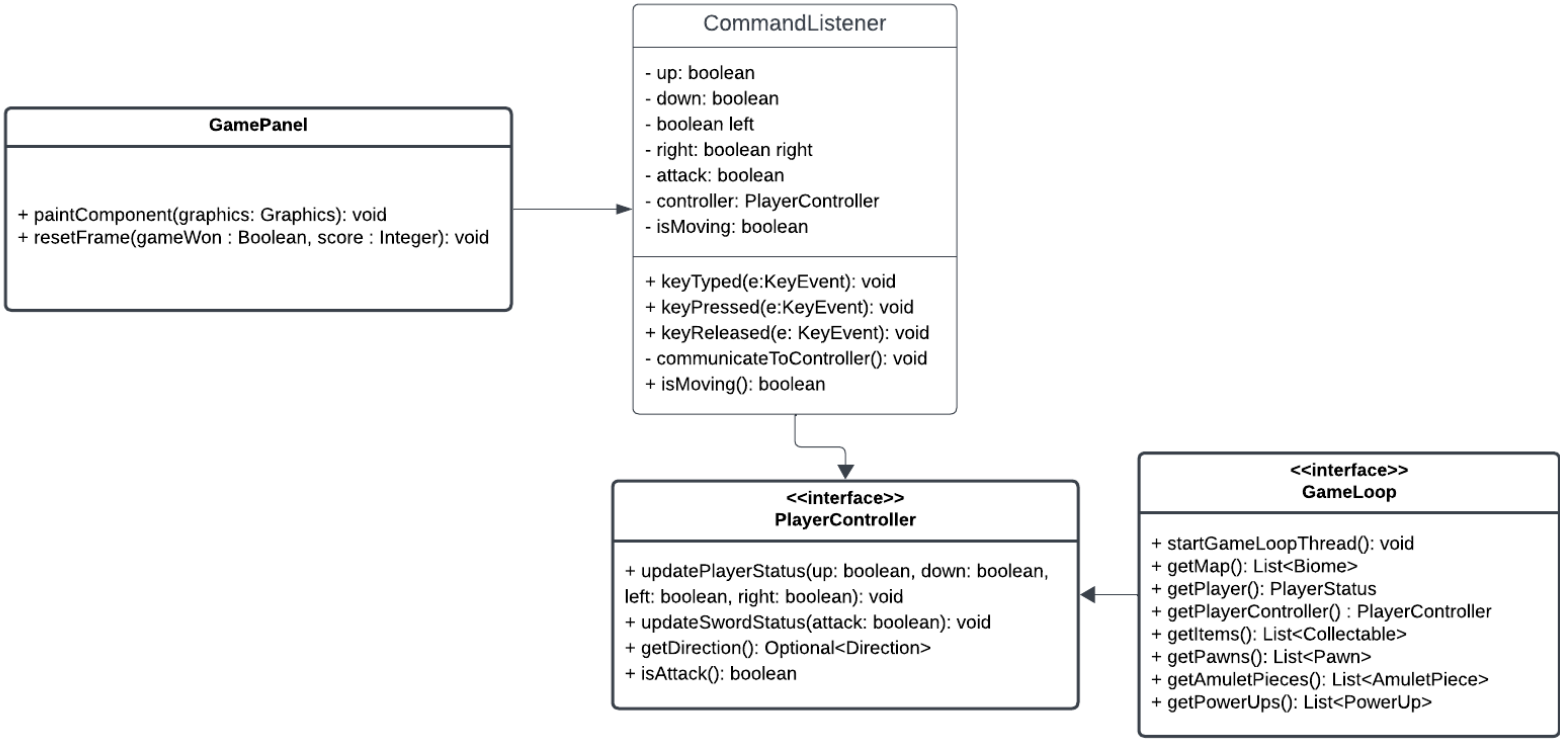


Figura 2.2.4.4: Schema UML che illustra l'uso di Command con l'interazione tra CommandListener e PlayerController

PROBLEMA: devono essere comunicati al gioco i comandi provenienti dalla tastiera per far effettuare al personaggio giocabile le azioni desiderate.
SOLUZIONE: è stato usato il pattern Command nella classe PlayerController che prende i comandi provenienti da CommandListener e li comunica a GameLoop per far agire il personaggio come dalle indicazioni fornite dall'utente.

Disegno del mondo di gioco

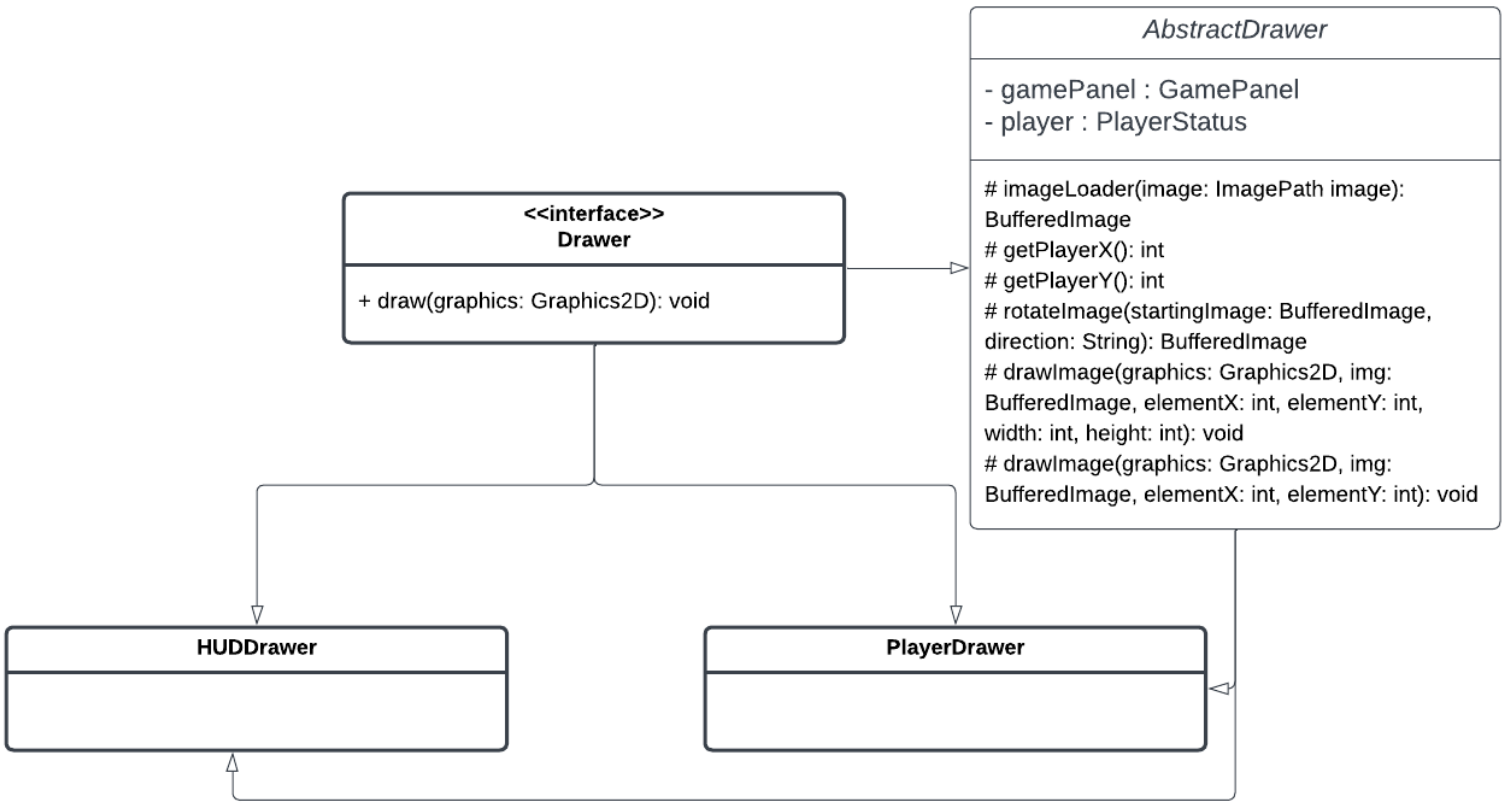


Figura 2.2.4.5: Schema UML per l'uso del pattern Template Method per il disegno del mondo di gioco

PROBLEMA: l'applicazione richiede che gli elementi vengano rappresentati all'interno del game panel e tutte le istanze di Drawer necessitano di funzionalità comuni da diversificare in base al caso specifico per poter completare il loro compito.
SOLUZIONE: si è utilizzato il pattern Template Method per massimizzare il riuso, creando una classe astratta AbstractDrawer che fornisce lo scheletro dell'algoritmo per rappresentare gli elementi di gioco e altre funzionalità utili, tra cui la rotazione delle immagini e il caricamento delle immagini richieste. Le implementazioni di Drawer che usano AbtsractDrawer realizzate da me sono PlayerDrawer e HUDDrawer (e per questo ho messo solo queste nel UML).

Sviluppo

Testing automatizzato

Per verificare il corretto funzionamento delle componenti dell'applicazione si è scelto di creare una serie di classi di test che sfruttano la suite per testing automatizzati JUnit.

Funzionalità testate automaticamente:

- Player: viene verificato il funzionamento dei suoi metodi di movimento, di attacco, di danno e di ottenimento di un pezzo di amuleto.
- PlayerHealth: vengono verificati il recupero e la perdita dei punti vita, l'incremento della vita massima e la gestione del danno quando lo scudo viene attivato.
- Sword: si verifica che il suo movimento e quello del giocatore sono correttamente allineati, che avvengano correttamente sia il cambio di tipo sia di forza e che al cambio di tipo corrisponda un cambio nella sua area di collisione.
- Score: viene verificato che l'aumento di punteggio funzioni correttamente anche sotto effetto di un moltiplicatore di punteggio
- Pawn: si verificano il suo movimento, il cambio di direzione del movimento e che subisca danno quando viene colpito da Sword.
- EnemyFactory: si verifica la corretta generazione dei nemici.
- Item: viene verificato che un Item venga effettivamente raccolto da Player, che tutti i tipi di Item applichino i loro effetti sul Player.
- ItemFactory: si verifica la corretta generazione di tutti i tipi di Item.
- PowerUp: viene verificato che un Power-Up venga effettivamente raccolto da Player, che tutti i tipi di Power-Up applichino i loro effetti e che la loro durata funzioni correttamente.
- PowerUpFactory: si verifica la corretta generazione di tutti i tipi di power-up.
- Map: viene testata la corretta generazione della mappa, delle sue componenti (biomi, stanze e corridoi) e l'interazione col giocatore.
- PlayerController: viene verificato che i metodi generino i cambiamenti richiesti al Player del Model.

- ScoreBoard: vengono testati l'inserimento di un nuovo punteggio in classifica e che venga ordinata correttamente.

Note di sviluppo

Vittorio Damiano Brasini

Nessuna nota di sviluppo particolare. Utilizzo sporadico di Optional[(1)][(2)][(3)][(4)][(5)] e di Stream[(1)][(2)][(3)]

Luca Nicholae Ferar Tofan

Utilizzo di Stream:

https://github.com/FilVel/OOP23-java-wulf/blob/82543d6925351bf53cc219c56d934559966a9a9b/src/main/java/javawulf/scoreboard/ScoreBoardImpl.java#L89

Utilizzo di Optional:

https://github.com/FilVel/OOP23-java-wulf/blob/82543d6925351bf53cc219c56d934559966a9a9b/src/main/java/javawulf/model/powerUp/PowerUpHandlerImpl.java#L12

Utilizzo di lamda:

https://github.com/FilVel/OOP23-java-wulf/blob/82543d6925351bf53cc219c56d934559966a9a9b/src/main/java/javawulf/view/gameMenu/GameMenuPanel.java#L195

Roberto Sopranzetti

1) Utilizzo di Stream e Lambda expressions. Usate più volte, questo è un esempio:
https://github.com/FilVel/OOP23-java-wulf/blob/8ff41eacc12843594ae8881ae61911e01281df9d/src/main/java/javawulf/controller/GameLoopImpl.java#L68-L71

2) Utilizzo di Wildcard:

https://github.com/FilVel/OOP23-java-wulf/blob/8ff41eacc12843594ae8881ae61911e01281df9d/src/main/java/javawulf/view/ItemDrawer.java#L22

Filippo Velli

1) Uso di Optional:

https://github.com/FilVel/OOP23-java-wulf/blob/df496d05763a7e61b57967a23d2057a640b2ed33/src/main/java/javawulf/controller/PlayerControllerImpl.java

https://github.com/FilVel/OOP23-java-wulf/blob/df496d05763a7e61b57967a23d2057a640b2ed33/src/main/java/javawulf/model/BoundingBoxImpl.java

2) Uso di Lambda:

https://github.com/FilVel/OOP23-java-wulf/blob/4e1f81a8cbf23f34d229df98ef8b5d0572ecec21/src/main/java/javawulf/view/GamePanel.java#L95

https://github.com/FilVel/OOP23-java-wulf/blob/4e1f81a8cbf23f34d229df98ef8b5d0572ecec21/src/main/java/javawulf/controller/GameLoopImpl.java#L130

Sviluppo di codice scritto originariamente da altri (link originale: http://www.java2s.com/Tutorial/Java/0261__2D-Graphics/RotatingaBufferedImage.htm#google_vignette):

https://github.com/FilVel/OOP23-java-wulf/blob/8ff41eacc12843594ae8881ae61911e01281df9d/src/main/java/javawulf/view/AbstractDrawer.java#L62

Commenti finali

Autovalutazione e lavori futuri

Vittorio Damiano Brasini

Durante la fase di sviluppo mi sono occupato di definire l'ambiente di gioco (Mappa) e di esporre un'interfaccia utile alle entità di gioco ed ai personaggi. Ho suddiviso — almeno in fase iniziale — la parte di Model, di Controller e di View, e mi sono posto il problema di dover definire il tempo di gioco e l'aggiornamento periodico dei suoi stati e dei suoi componenti; ciò mi ha portato alla definizione del GameLoop. Quest'ultimo a mio avviso avrebbe dovuto essere scorporato in più metodi privati per non intaccarne la leggibilità della struttura.
Mi sono inoltre occupato del popolamento randomico dell'ambiente di gioco con le entità sviluppate dagli altri miei compagni di gruppo.
In fase iniziale di sviluppo del progetto, ho incontrato io in primis diverse sfide che credo abbiano partecipato a influenzare il risultato finale. Il tempo a disposizione è stato limitato, impedendomi di rifinire i dettagli del codice, eliminare ripetizioni e ottimizzare i cicli in modo più efficiente. Mi dispiace particolarmente di aver dovuto scrivere la relazione nell'ultimo giorno prima della consegna, a dimostrazione del fatto che forse si sarebbe potuto impiegare qualche giorno in più nella cura del progetto a scapito della *deadline* stabilita (parlo a titolo personale).
In conclusione, mi ritengo comunque soddisfatto del lavoro svolto, quantomeno apprezzando il risultato finale ottenuto.

Luca Nicholae Ferar Tofan

Essendo questo il mio primo vero e proprio progetto di gruppo, sono contento del mio risultato, ma devo essere sincero nel dire che mi è dispiaciuto e non mi è sembrato giusto nei confronti dei miei colleghi la mia parziale assenza nella prime parti di sviluppo.

Questo mi ha portato a fare in una settimana, un lavoro che avrei dovuto spargere su molte più giornate, ma penso comunque di aver dato un buon contributo nella fase di analisi e progettazione.

Comunque, tutto il codice che ho sviluppato è stato utile alla realizzazione del progetto e nessuno dei miei compagni ha dovuto sviluppare parti che avevamo inizialmente concordato assieme.

Riassumendo ciò sono contento di quello che ho realizzato ma avrei preferito farlo con altre tempistiche.

Punti di forza:

Tutte le funzioni promesse sono state sviluppate.

Possibilità di estensione dei power ups.

Corretto riutilizzo del codice dove è stato possibile.

La classifica si aggiorna automaticamente.

Punti di debolezza:

Dovuto a fattori esterni non sono riuscito a dedicare il tempo che ho voluto per la progettazione e estetica del menu di gioco.

La classifica non segue MVC ma viene fatto tutto all'interno di essa.

Il mio compito all'interno del team è stato di sviluppare Power Ups, Scoreboard e il Menu di gioco.

In futuro vorrei lavorare meglio sul menu e sulla classifica.

Roberto Sopranzetti

Il mio ruolo nel gruppo è stato gestire la creazione degli oggetti e dei nemici presenti nel mondo di gioco. Ho avuto la responsabilità di definire la loro struttura, i loro comportamenti, la loro logica di aggiornamento nel GameLoop e l'implementazione della loro rappresentazione grafica sulla mappa di gioco.

Lavorare a questo progetto è stato piuttosto difficile, con il tempo avuto a disposizione ho potuto implementare solamente le parti del mio progetto obbligatorie, mentre una parte opzionale (l'archetipo di nemico Guard) non son riuscito a completarla in tempo, ho fatto in tempo solamente a testare la sua factory e che venisse effettivamente creato bene.

Avrei voluto anche poter implementare il super nemico Wolf che avrebbe inseguito il giocatore per un po' tutta quanta la mappa e che non sarebbe stato possibile ucciderlo, ma solamente a stordirlo con lo spadone, mentre per gli oggetti avrei voluto implementare la minimappa che permetteva al giocatore di capire dove si trovava senza dover correre alla cieca per tutto il mondo di gioco.

Son soddisfatto di come sia riuscito a lavorare con i membri del gruppo e a creare parti di codice non troppo staccate tra loro, anzi mi hanno permesso a volte di poter seguire il principio DRY ed il principio KISS.

Filippo Velli

Nel gruppo il mio ruolo era quello realizzare il concetto di giocatore, definendone le funzionalità nel Model, il suo aggiornamento nel GameLoop e la sua rappresentazione grafica. Inoltre, ho gestito anche la rappresentazione della HUD e l'invio dei comandi dalla View al resto dell'applicazione durante la partita.

Nel corso del progetto ho cercato di dare la massima collaborazione ai miei colleghi fornendo loro supporto in real time e diventando, di conseguenza, una sorta di coordinatore del gruppo. Sicuramente si poteva fare di più e meglio anche perché questa espansione del mio ruolo, dovuta anche alla discontinuità della presenza di parte del gruppo che ha creato ritardi e incomprensioni, mi hanno appesantito personalmente e probabilmente ridotto la qualità del mio lavoro. Inoltre, la parte grafica poteva essere più curata così come la qualità del mio codice che avrebbe potuto utilizzare funzionalità più avanzate del linguaggio Java.

Tenuto conto di tutti questi limiti, mi ritengo abbastanza soddisfatto dei risultati, riuscendo a seguire diversi principi della programmazione ad oggetti, garantendo una certa estendibilità e riusabilità del codice, e ad utilizzare dei design pattern all'interno della mia parte di codice.

Se dovessi continuare a sviluppare il progetto ulteriormente integrerei la possibilità di personalizzare i comandi, un redesign della HUD ed eventualmente una funzionalità multigiocatore.

Guida Utente

Figura 5: Schermata di gioco

Per controllare il personaggio si usano le frecce direzionali e per attaccare si usa il puntante (,) virgola.

I cuori rossi rappresentano i punti vita del giocatore. Appena subisce un danno diventa grigio quello rosso più a destra. Se il giocatore ha uno o più punti vita scudo questi appariranno accanto ai cuori, ma non si potranno recuperare usando le ampolle. Accanto ai cuori vi è la spada che indica il suo stato attuale: se è grigia è la spada normale; se è gialla è la GreatSword, che aumenta la gittata dell'attacco del giocatore.

Dopo le spade vi sono tre indicatori il primo è per i power-up; infatti, cambia colore appena si raccoglie un fiore e questo ne determina l'effetto (nero = nessuno, rosso = potenziamento forza, verde = aumento della velocità, giallo = punteggio duplicato, blue = invulnerabilità). Il secondo indica se il giocatore è invulnerabile dopo un attacco (grigio = nessuna invulnerabilità, giallo = invulnerabile). Il terzo se si illumina di viola vuol dire che il giocatore è in coordinata con un pezzo di amuleto.