Relario: Tales of Relano
Progetto per il corso di "Programmazione ad Oggetti"

Lorenzo Cinelli, Mihai Mazuru, Kimi Osti, Sara Panfini

30 gennaio 2025

Indice

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Il software realizzato è un videogioco 2D con vista dall'alto. Lo svolgimento gira attorno a un personaggio principale, controllato dall'utente, che deve attraversare le stanze di un castello per raggiungere lo scontro finale con il Re, di cui deve conquistare il trono.
Durante l'esplorazione delle stanze all'utente potranno essere affidate delle quest da completare per poter proseguire nel gioco.
Inoltre, gli verranno presentate delle stanze quasi completamente interattive. In particolare, ci saranno dei personaggi non giocanti (neutri, alleati o nemici) che potranno, al momento dell'interazione, mostrare un messaggio, donare degli oggetti oppure ingaggiare un combattimento. Inoltre, sarà possibile interagire con gran parte degli elementi di arredo presenti in stanza, tra cui elementi come armature o vasi, che possono contenere oggetti collezionabili, oppure tappeti o botole, che possono celare al loro interno dei nemici.
Il combattimento si svolge a turni. Ad ogni turno, il giocatore può decidere se attaccare o chiedere pietà al nemico, così come può navigare il suo inventario e usare oggetti senza perdere il diritto al turno. Il nemico, in caso venga chiesta pietà, potrebbe concederla oppure rifiutarla e attaccare immediatamente il giocatore, che perde il turno.

Requisiti funzionali

- I nemici all'interno del gioco saranno di varie tipologie, e dovranno offrire un comportamento variabile all'utente per quanto riguarda le richieste di pietà;

- L'arredo delle stanze viene generato casualmente garantendo l'assenza di sovrapposizioni, e le diverse tipologie di elementi di arredo devono offrire diversi scenari di interazione;
- Le quest devono essere di diverse tipologie e richiedere diverse azioni da parte del giocatore;
- Il combattimento finale deve offrire al giocatore una scelta, che comporta finali diversi.

Requisiti non funzionali

- Per offire un'esperienza gradevole all'utente, si mira a realizzare un software efficiente.
- Il software sarà portabile su tutti i maggiori sistemi operativi.

1.2 Modello del Dominio

Il dominio applicativo dell'applicazione viene modellato in ogni momento dal concetto di stanza, ovvero il "container" all'interno del quale si svolge la fase centrale del gioco. Nella stanza, oltre al personaggio principale, sono presenti altre entità, che possono essere personaggi viventi non giocanti (nemici o generici NPC) oppure elementi di arredo. Il giocatore può possedere nel suo inventario diverse tipologie di oggetti, che vengono anch'essi modellati come entità. In questo scenario, diventa possibile gestire, tramite la stanza e le informazioni che ogni entità offre, l'intero modello del dominio, estraendo le istanze di interesse per gestire le situazioni contingenti come il combattimento.
Per quanto riguarda l'arredamento, le entità si dividono in tre tipologie fondamentali: arredamento interattivo, che blocca il movimento ma permette l'interazione e può rilasciare un oggetto, che verrà aggiunto all'inventario del giocatore; arredamento calpestabile, che non ostruisce il movimento e permette l'interazione, ma può nascondere un nemico, con cui viene avviato il combattimento non appena vi si interagisce; arredamento passivo, che ostacola il movimento e non permette alcun tipo di interazione.
Per quanto riguarda i nemici, ad ognuno corrisponde un tipo, che ne definisce il livello di difficoltà. Quando viene sconfitto, un nemico rilascia un oggetto che viene aggiunto all'inventario del giocatore al momento della vittoria. Ad ogni nemico viene anche associato un comportamento in caso di richiesta di pietà, indipendente dal tipo e proprio di ogni singola istanza. Nel caso in cui

il giocatore venga risparmiato dal nemico, non ottiene il suo bottino.
Gli NPC modellano tutti i personaggi non ostili all'interno del gioco, con cui è possibile interagire in ogni momento. Anche loro possono rilasciare oggetti di inventario al momento dell'interazione, oppure mostrare messaggi, che potranno aiutare o meno il giocatore a completare la quest.
Il personaggio principale, che si muove nella mappa e interagisce con il resto delle entità presenti, può portare con sé alcuni oggetti di inventario ottenuti interagendo con le altre entità. Questi oggetti possono essere di vario tipo e, a seconda della tipologia, avere diversi effetti (cura, aumento del danno per le armi e protezione per le armature, oppure nessuno per gli oggetti collezionabili). Le armi e le armature, per essere effettive, devono essere equipaggiate, e hanno una durabilità limitata. Al momento dell'uso dell'oggetto, il suo effetto viene attivato sul giocatore. Un oggetto qualsiasi può anche essere scartato per liberare spazio nell'inventario, che ha capacità limitata.

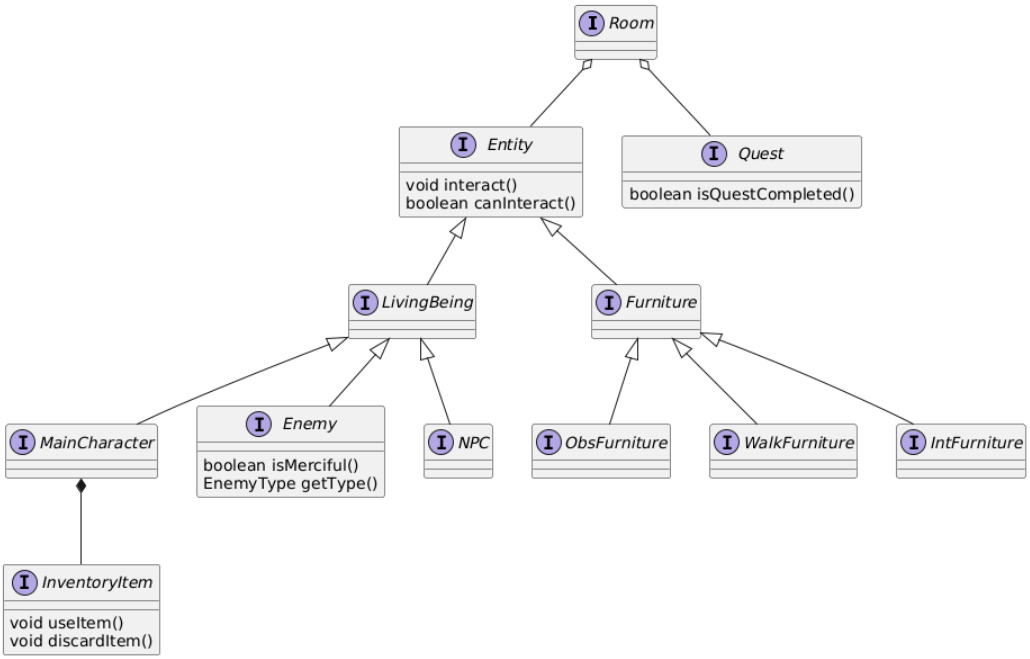

Figura 1.1: Schema UML del dominio applicativo

Capitolo 2

Design

2.1 Architettura

Il software si basa sull'architettura MVC nella sua declinazione standard. In particolare, ogni elemento dell'architettura offre un unico entry point verso l'esterno, in modo che gli accessi alle sue funzionalità possano essere uniformi e consistenti, offrendo un ulteriore grado di incapsulamento.
Il Model offre come proprio entry point l'interfaccia Room, che fa da scenario base per lo svolgimento della fase di esplorazione del gioco. All'interno della stanza infatti, sono presenti tutte le entità, che vengono modificate ad ogni tick del motore fisico tramite un metodo offerto dalla stessa Room, deputata a controllare anche se le singole entità siano in grado di muoversi al suo interno.
Il Controller, che conserva il riferimento alla stanza in cui attualmente si trova il gioco, gestisce al suo interno le transizioni di stato per le varie fasi del gameplay, interrogando la View per mostrare le interfacce corrette e richiedendo al Model eventuali modifiche. Il Controller è anche responsabile della temporizzazione dell'aggiornamento del motore di gioco, e della traduzione delle entità del Model in elementi rappresentabili correttamente dalla View.
La View offre un entry point centrale da cui è possibile richiedere di mostrare le varie interfacce o di ottenere l'accesso ai loro riferimenti per chiamare procedure proprie di tali istanze. Nell'architettura realizzata, la View agisce come elemento passivo ricevendo i dati da mostrare dal Controller tramite opportune interrogazioni. La gestione dell'input permette alla View di comunicare particolari eventi al Controller, che li gestirà e ne rifletterà eventualmente gli effetti sul Model.
Nella realizzazione dell'architettura MVC, modificare la View non impatta minimamente il Model, dal momento che è solamente il Controller a dialo-

gare con questa componente. Dall'altro lato, il Controller potrebbe essere impattato da una modifica della View radicale (come per esempio trasformare la GUI attiva in un'interfaccia reattiva, oppure la rimozione dei suoni), mentre non dovrebbe subire alcun cambiamento se venissero modificate le tecniche implementative della GUI - come per esempio una modifica della libreria grafica -, a patto che la GUI modificata sia in grado di rispettare il contratto stabilito dalle due interfacce (ad esempio accettare gli stessi tipi di chiamate parametrizzate).

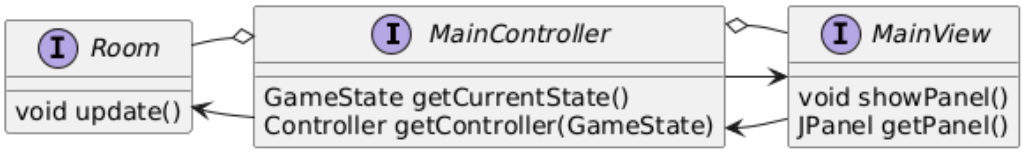

Figura 2.1: Schema UML degli entry point dei rapporti fra componenti di MVC

2.2 Design dettagliato

2.2.1 Lorenzo Cinelli

Collisioni e Interazioni

Problema: Il giocatore e gli altri esseri viventi possono muoversi all'interno di una stanza ma non possono coesistere contemporaneamente nello stesso punto della stanza insieme ad un altro essere vivente o a una parte di arredamento (non calpestabile). Il giocatore e gli altri esseri viventi possono interagire con un'entità (che può essere un altro essere vivente o un pezzo di arredamento) di fronte ad egli.

Soluzione: Il controllo delle collisioni effettuato nel metodo *canMove* - che decreta se il movimento del soggetto è consentito oppure no - deve quindi prendere in considerazione tutti gli esseri viventi presenti in mappa, l'arredamento e la dimensione della stanza e verificare che il soggetto si muoverà in un posto consentito. Il controllo delle interazioni effettuato nel metodo *canInteract* - che decreta se ci si trova di fronte ad un'entità interattiva - prende in considerazione tutti gli esseri viventi presenti in mappa e l'arredamento. È stata fatta la scelta di evitare di avere interazioni attive da parte delle entità non giocanti (NPC), di conseguenza solo il protagonista interagirà attivamente con le varie entità, tuttavia con la stessa soluzione sarebbe possibile avere interazioni attive anche da parte degli altri esseri viventi.

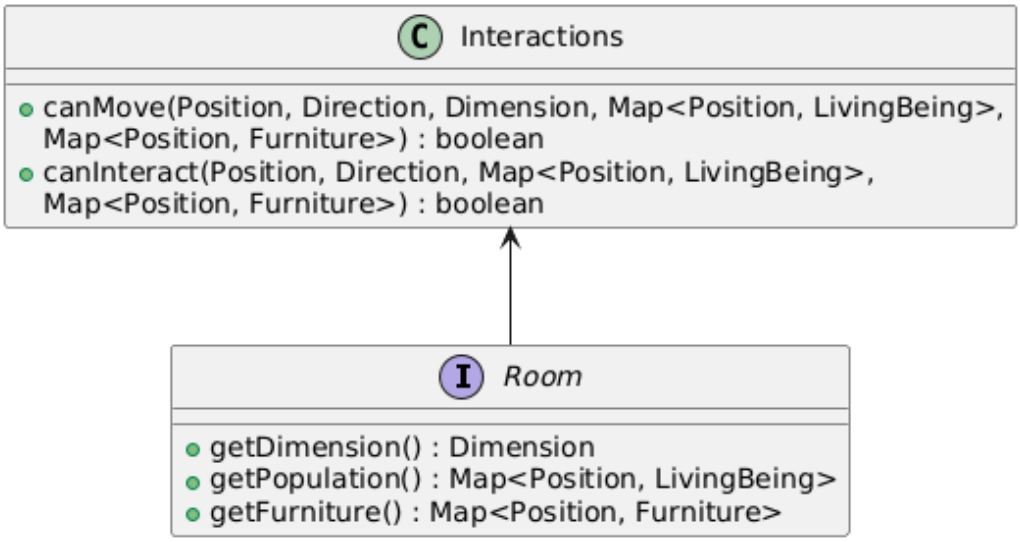

Figura 2.2: Schema UML delle classi e interfacce coinvolte nelle collisioni ed interazioni

Input da tastiera nell'utilizzo dell'Inventario

Problema: Il giocatore comanda le interazioni con l'inventario - utilizzabile durante la fase di esplorazione della mappa o di combattimento - tramite input da tastiera catturati dal *InventoryController*.

Soluzione: La comunicazione dell'input viene fatta attraverso il pattern *Observer*. La classe *InventoryViewImpl* fa da *observable* tramite il *KeyListener* correlato trasmette gli eventi registrati all'*InventoryController* il quale fa da *observer*, implementandone l'interfaccia. Nello specifico per avere una soluzione estendibile si utilizza il *GameKeyListener* - il quale estende l'interfaccia *KeyListener* - che filtra i tasti di input traducendoli in un evento da dare in pasto agli Observer.

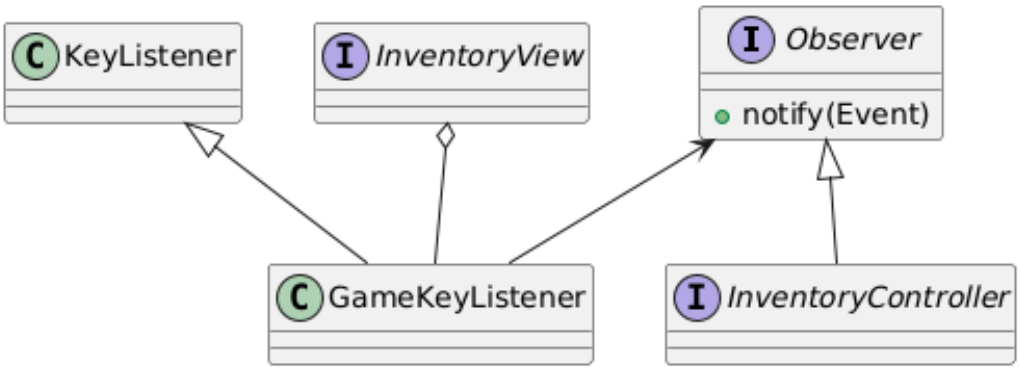

Figura 2.3: Schema UML delle classi e interfacce coinvolte nell'input per l'inventario

Creazione dei componenti per l'interfaccia dell'inventario

Problema: L'interfaccia grafica dell'inventario è composta da diversi componenti.

Soluzione: Per alleggerire la classe *InventoryViewImpl* è stata creata una classe per fornire i vari componenti. Questa classe - *InventoryViewFactoryImpl* - funge da factory per l'interfaccia dell'inventario, in quando è adibita alla creazione dei vari componenti dell'interfaccia, quali la barra dei comandi, la barra della vita, la lista degli oggetti contenuti, la descrizione del singolo oggetto e gli oggetti equipaggiati.

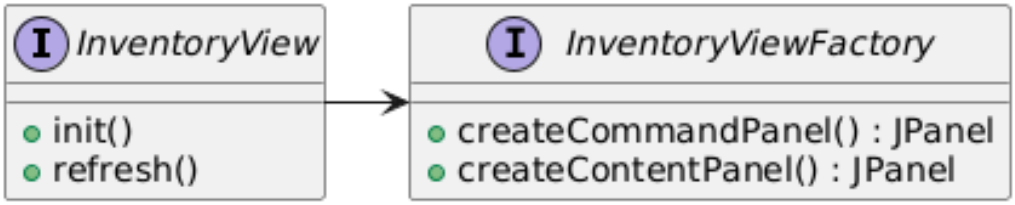

Figura 2.4: Schema UML delle classi e interfacce coinvolte per la creazione dei componenti dell'interfaccia dell'inventario

Scene di transizione

Problema: I vari stati del gioco sono intervallati da scene di transizione.

Soluzione: Le scene di transizione si possono suddividere in tre tipologie: l'introduzione, l'avanzamento di stanza ed il finale. Queste sono composte da una o più immagini oltre che eventualmente da un effetto sonoro. Più nello specifico le scene di fine gioco possono essere 3: due differenti in caso di vittoria, definite in base al modo in cui si è vinto il gioco, e la terza che rappresenta il caso di sconfitta. Le varie scene vengono chiamate dal metodo *show* dell'apposito controller, a cui si passa lo stato attuale del gioco. In base allo stato del gioco il controller stabilisce quale scena sia da mostrare e in quale stato andare in seguito alla visualizzazione della scena.

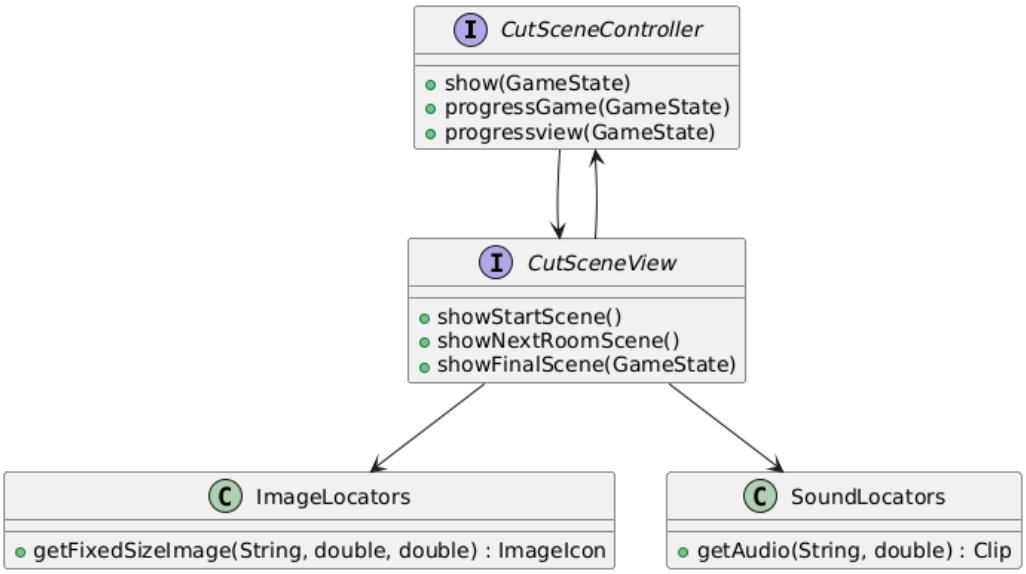

Figura 2.5: Schema UML delle classi e interfacce coinvolte per le scene di transizione

Animazioni di combattimento

Problema: Durante gli attacchi di un combattimento si mostrano delle animazioni accompagnate da un effetto sonoro.

Soluzione: La visualizzazione delle animazioni viene effettuata al momento dell'attaacco di un'entità verso l'avversario. Vengono visualizzate animazioni differenti a seconda della direzione dell'attacco (dal giocatore al nemico o viceversa). L'effetto sonoro riprodotto è scelto in modo pseudo casuale tra una libreria di possibili suoni.

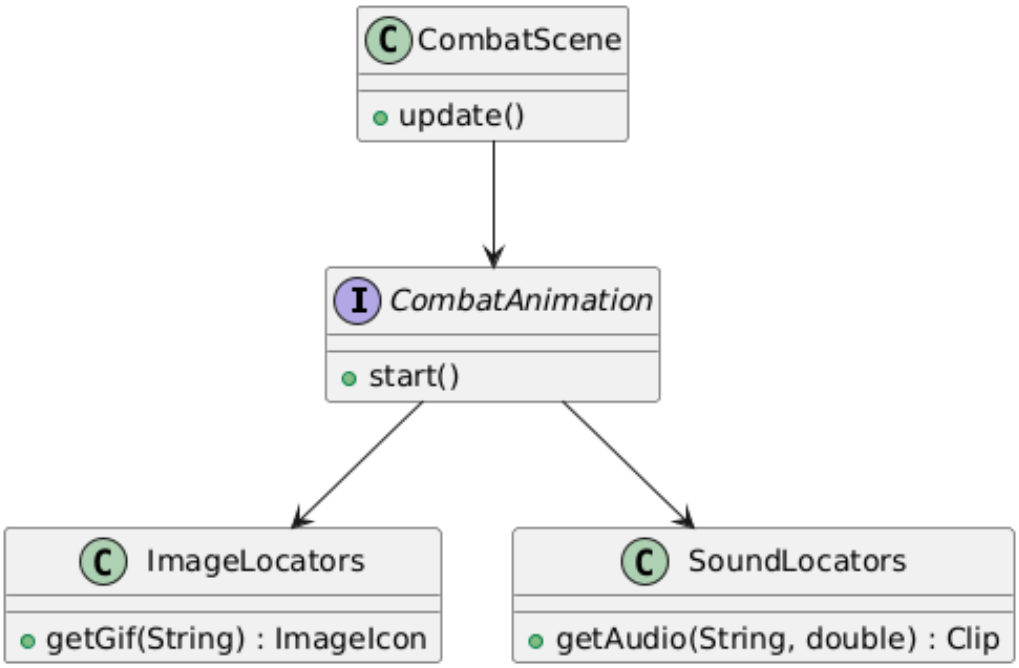

Figura 2.6: Schema UML delle classi e interfacce coinvolte per le animazioni di combattimento

Gestione delle colonne sonore

Problema: Durante le varie fasi del gioco sono presenti diverse colonne sonore.

Soluzione: Le varie classi di view riproducono la colonna sonora corrispondente allo stato del gioco in cui si trovano fino a ché lo stato non cambia. La gestione dei suoni passa da un sound locator, che prende le risorse e le trasforma in clip, e poi ogni clip è gestita internamente alla singola view in cui viene eseguita. Più nello specifico c'è una colonna sonora per il menù iniziale, una per la fase di esplorazione del gioco, e due differenti per la fase di combattimento, distinguendo se il nemico sia il Boss oppure no.

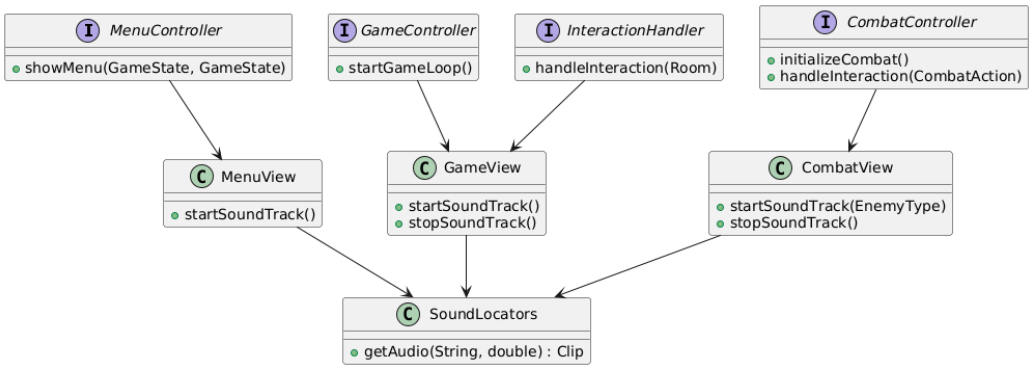

Figura 2.7: Schema UML delle classi e interfacce coinvolte per le colonne sonore

Caricamento delle risorse

Problema: Le risorse - immagini e suoni - devono essere trovate ed aperte.

Soluzione: Sono state create due classi - rispettivamente *ImageLocators* e *SoundLocators* - per restituire le risorse aperte ed utilizzabili alle diverse view. Queste classi vengono utilizzate per le scene di transizione, le animazioni di combattimento e le colonne sonore.

2.2.2 Mazuru Mihai

Furniture

Problema : Nella modellazione degli oggetti di arredo mi sono imbattuto nel problema di come realizzare diverse tipologie di arredi senza ripetizioni di codice. Questi oggetti sono caratterizzati da un nome, una descrizione, una posizione e un tipo, e invece si differenziano per le seguenti proprietà: attraversabilità e interattività.

Soluzione Ho optato per l'implementazione del pattern *Template Method*. L'ho applicato realizzando una classe astratta *Furniture* che contiene la logica comune a tutti gli oggetti di arredo (con i corrispettivi getter) e i metodi astratti *isWalkable()* e *isInteractive()*. L'implementazione di questi metodi viene lasciata ad ogni sottoclasse. Difatti la classe *WalkableFurniture* definisce oggetti di arredo con i quali si può interagire e che non ostacolano il passaggio delle entità viventi, mentre la classe *ObstructingFurniture* definisce oggetti con caratteristiche diametralmente opposte (bloccanti e non interattivi) e invece la classe *InteractiveFurniture* definisce oggetti con caratteristiche

intermedie (bloccanti ed interattivi).
La soluzione scelta consente inoltre l'estendibilità, facilitando la creazione di una nuova ipotetica classe con oggetti attraversabili e non interattivi.

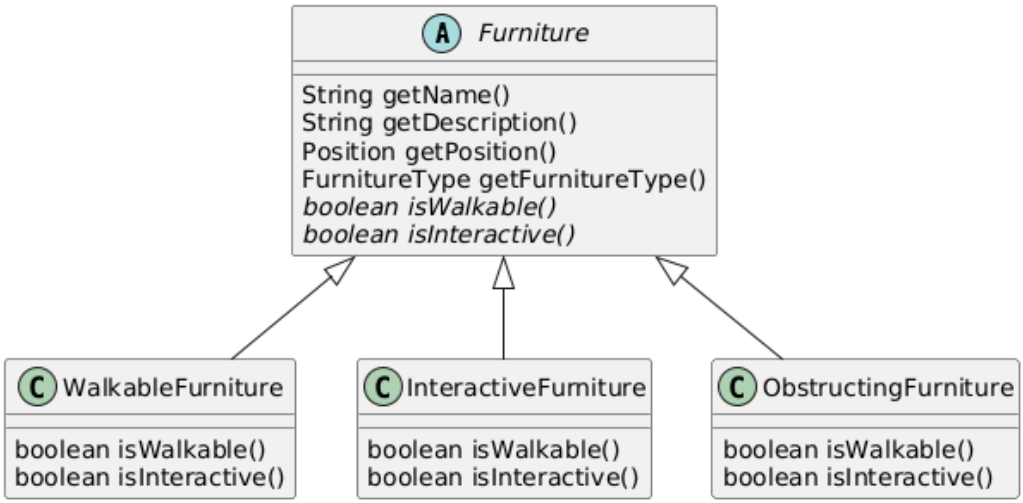

Figura 2.8: Schema UML dell'applicazione del pattern Template Method alla gerarchia delle Furniture.

Problema La creazione dei diversi oggetti di arredo attraverso una semplice "new" comporterebbe l'aggiunta di dipendenze al nostro codice, rendendolo più difficile e tedioso da testare, complicando la sua estendibilità ed eventualmente la sua manuntenzione.

Soluzione Proprio per questo ho scelto di utilizzare il pattern *Simple Factory* per la creazione degli arredi. Ho realizzato un'interfaccia che contiene i metodi per la creazione dei diversi tipi di arredi. Questi metodi vengono implementati dalla sua sottoclasse, la quale è in grado di creare oggetti randomici, specifici oppure solo di un determinato tipo. Gli unici parametri richiesti sono il tipo e la posizione dell'arredo, mentre le altre caratterische come il nome e la descrizione vengono gestite internamente alla factory attraverso una mappa che contiene per ogni tipo diverso di arredo una descrizione unica.

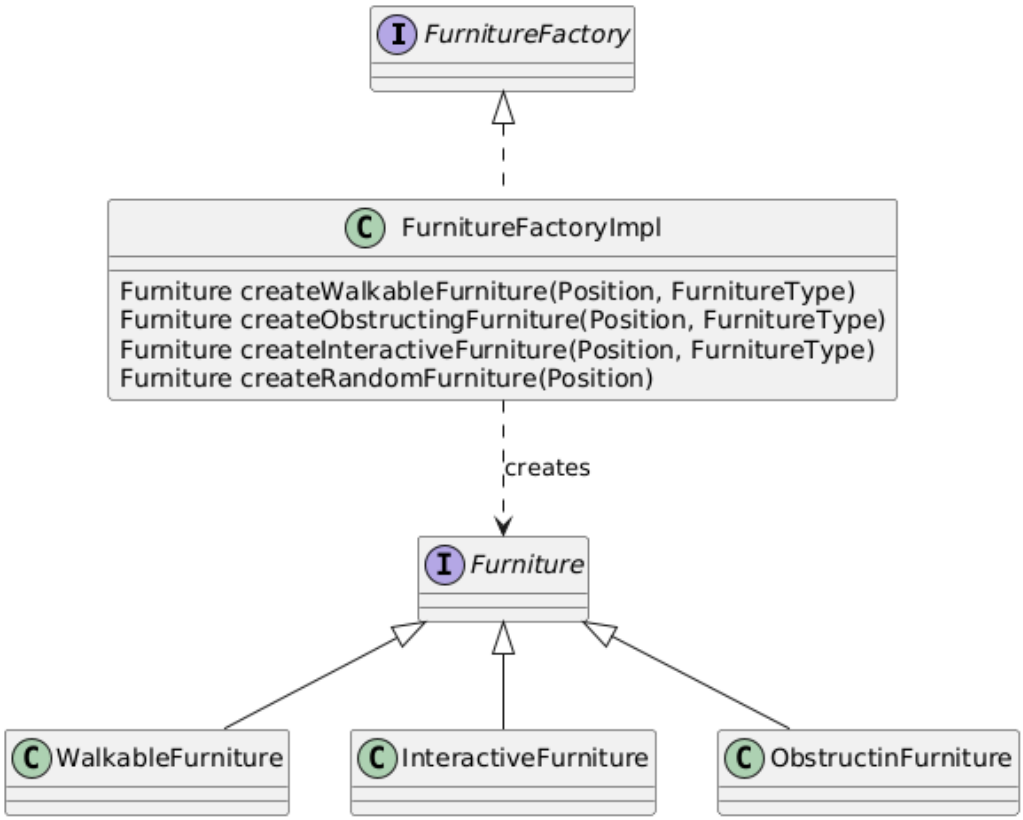

Figura 2.9: Schema UML dell'applicazione del patter Simple Factory per la creazione di Furniture.

Gestione pannelli di gioco

Problema Il nostro progetto si articola in diversi momenti di gioco e il problema iniziale era proprio quello di capire come gestire nel migliore dei modi il passaggio da una pannello di gioco ad un altro.

Soluzione Per risolvere questo problema il gruppo ha scelto di creare una *MainView* che si sarebbe occupata della creazione del frame, della creazione dei vari pannelli di gioco e delle transizione da un pannello all'altro. Io mi sono occupato dell'implementazione e ho optato per l'utilizzo di *CardLayout*, un manager layout che permette di associare ad ogni pannello una stringa di testo, attraverso la quale sarà possibile far emergere dalla "pila di carte" il pannello desiderato. L'invocazione del metodo per far emergere un certo pannello viene effettuata dal corrispondente controller.

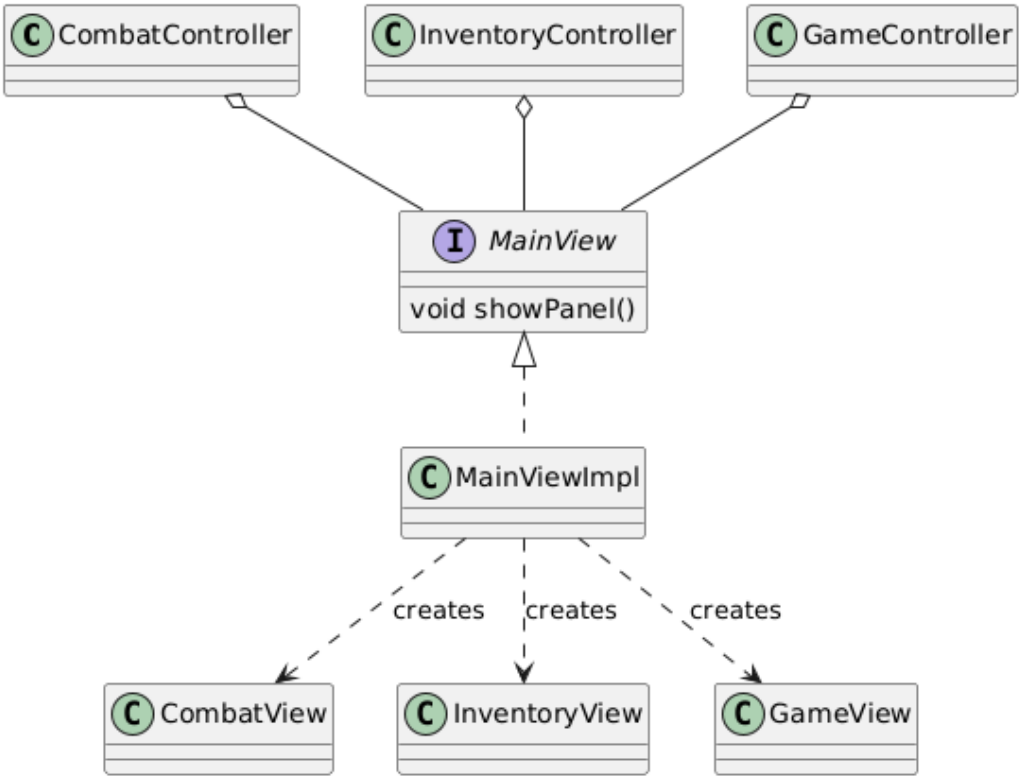

Figura 2.10: Schema UML che rappresenta MainView, alcuni pannelli di gioco, alcuni controller del gioco e le loro interazioni.

Gestione combattimento

Problema Il combattimento è uno dei momenti più importanti del gioco e la sua gestione in modo ottimale è fondamentale per garantire una gradevole esprienza all'utente. La gestione dei turni e la gestione delle azioni di combattimenti sono aspetti importanti per il raggiungimento di questo obiettivo.

Soluzione Le possibili scelte all'interno della fase di combattimento sono: attaccare, chiedere pietà oppure aprire l'inventario. Queste possibili azioni vengono gestite tramite un Action Listener che notifica il *CombatController*. Il controller in questione attraverso uno switch, al quale si accede solo se l'azione avviata precedentemente è finita, delega la gestione dell'azione agli appositi metodi privati oppure delega il lavoro al controller dell'inventario (nel caso si scelga di aprire l'inventario). I turni invece vengono gestiti in maniera automatica. Una volta che il giocatore ha attaccato, il nemico, se è

ancora in vita, attacca subito a sua volta. L'unico caso in cui è il nemico ad attaccare per primo è quando al giocare viene negata la richiesta di pietà e di conseguenza salta il turno. Sono stati aggiunti dei *Timer* per non accavallare le varie azioni e degli aggiornamenti testuli a fine di ogni azione, in modo da rendere il combattimento più intrattenete. Questi aggiornamenti testuali, insieme alle caratteristiche dei combattenti, vengono rese visibili e aggiornate attraverso una chiamata di "update" alla *CombatView*.

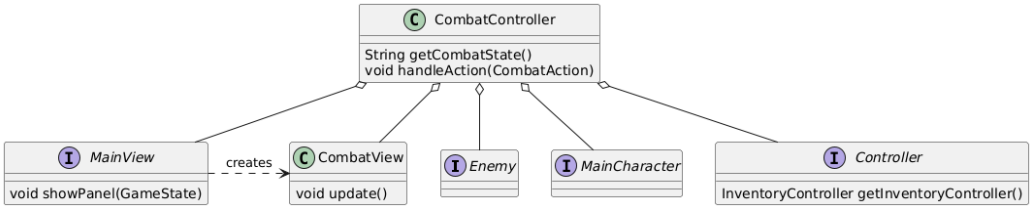

Figura 2.11: Schema UML che rappresenta il CombatController e le sue principali relazioni.

2.2.3 Kimi Osti

Questa sezione è stata redatta facendo riferimento, oltre ai pattern presentati dalla Gang of Four, anche al libro [Nys14], presentato dal Prof. Alessandro Ricci durante il seminario Game as a Lab.

Input da tastiera nella fase di esplorazione

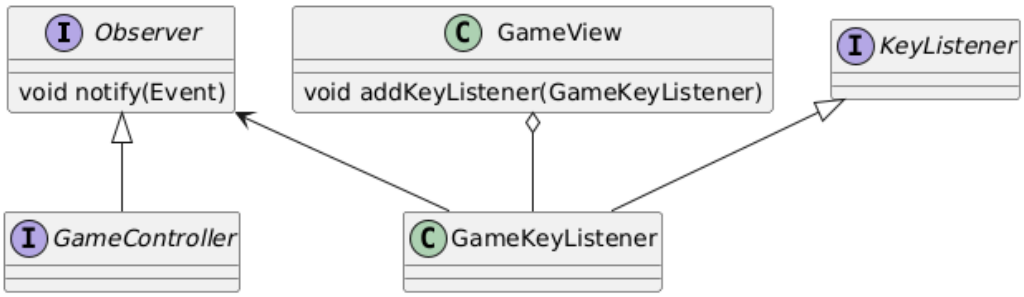

Figura 2.12: Schema UML delle classi e interfacce coinvolte

Problema Il giocatore comanda il personaggio principale tramite tastiera, e gli input devono essere catturati dal *GameController*.

Soluzione La collaborazione fra classi viene realizzata grazie al pattern *Observer*. In particolare, la classe *GameView* fa da observable, e tramite il *KeyListener* registrato trasmette i propri eventi al *GameController*, che fa da observer. In particolare, in questa soluzione viene sfruttato un *GameKeyListener* che implementa l'interfaccia *KeyListener*, filtrando i tasti di interesse per il gioco e traducendoli in un particolare *Event*, enumerazione definita all'interno del progetto per tracciare gli eventi rilevanti all'applicazione. Grazie a questa enumerazione, gli eventi sono trattati ad alto livello e si permette facilmente estendibilità verso la funzionalità opzionale della riassegnabilità dei tasti, a patto di permettere al GameKeyListener di modificare la mappatura da evento di pressione del tasto ed *Event*.

Creazione e gestione dei componenti per l'interfaccia di esplorazione

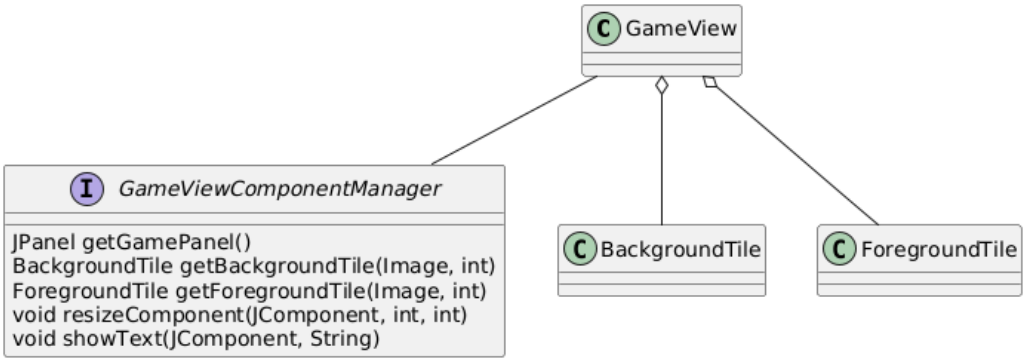

Figura 2.13: Schema UML delle classi e interfacce coinvolte

Problema Nell'interfaccia della fase di esplorazione, si deve renderizzare il contenuto della mappa, oltre a un banner contenente i comandi nella parte superiore dello schermo e un banner con eventuali messaggi di interazione nella fase inferiore. Ciò significa avere diversi pannelli, tutti accomunati dallo sfondo nero, e di cui i due superiori devono contenere eventualmente testo, mentre quelli centrali delle scene rappresentate dalla sovrapposizione di *BackgroundTile* e *ForegroundTile*

Soluzione Per alleggerire l'implementazione di *GameView* e renderla più comprensibile, si è creata una helper class in grado di gestire i componenti dell'interfaccia. In particolare, si sottolineano i metodi che fungono da Factory, ovvero i tre metodi *getGamePanel*, *getBackgroundTile* e *getForegroundTile* che vengono usati per creare i componenti che andranno poi a

formare l'interfaccia grafica nel suo complesso. Vengono inoltre definiti due metodi per ridimensionare i componenti e per mostrare il testo contenuto al loro interno.

Gestione del Main Loop

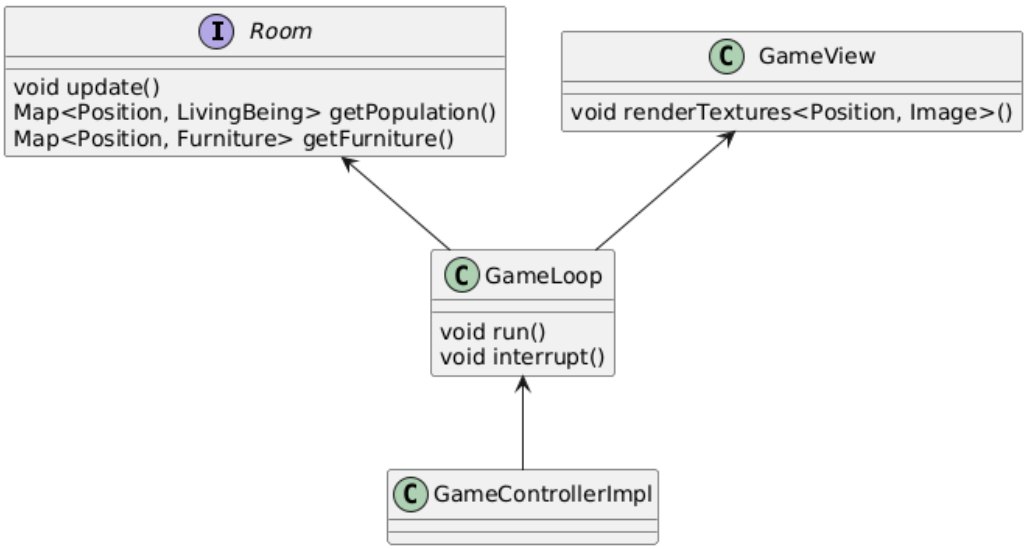

Figura 2.14: Schema UML delle classi e interfacce coinvolte

Problema Il Model, essendo presenti entità viventi all'interno della stanza, deve essere periodicamente aggiornato per rendere effettivo il movimento di tali entità, e in particolare il risultato dell'aggiornamento deve essere reso visibile all'utente tramite View.

Soluzione All'interno di *MainControllerImpl* viene istanziato il *GameLoop*, un thread separato che si occupa di gestire la temporizzazione degli aggiornamenti del Model, come presentato nel Game Loop Pattern in [Nys14]. In questo modo, la gestione dell'input rimane delegata a *GameControllerImpl*, mentre la temporizzazione dell'aggiornamento viene gestita da *GameLoop* per evitare dipendenze. Inoltre, la temporizzazione viene legata al tempo reale piuttosto che al tempo di CPU, per evitare rese diverse su processori di diverse potenze. *GameLoop* è anche deputata di comandare a *GameView* gli aggiornamenti necessari a rappresentare le modifiche del Model, e per motivi legati alla rappresentazione unitaria della posizione all'interno del Model l'aggiornamento della View viene effettuato di pari passo rispetto a quello del Model, anche se si sarebbe potuto svincolare per massimizzare il framerate

indipendentemente dalla velocità di aggiornamento del Model.
Nello scenario del Game Loop pattern viene applicato anche il pattern dell'Update Method, sempre presentato in [Nys14] e strettamente legato. In particolare, i metodi interessati sono *update* in *Room* e *renderTextures* in *GameView*.

Individuazione delle texture

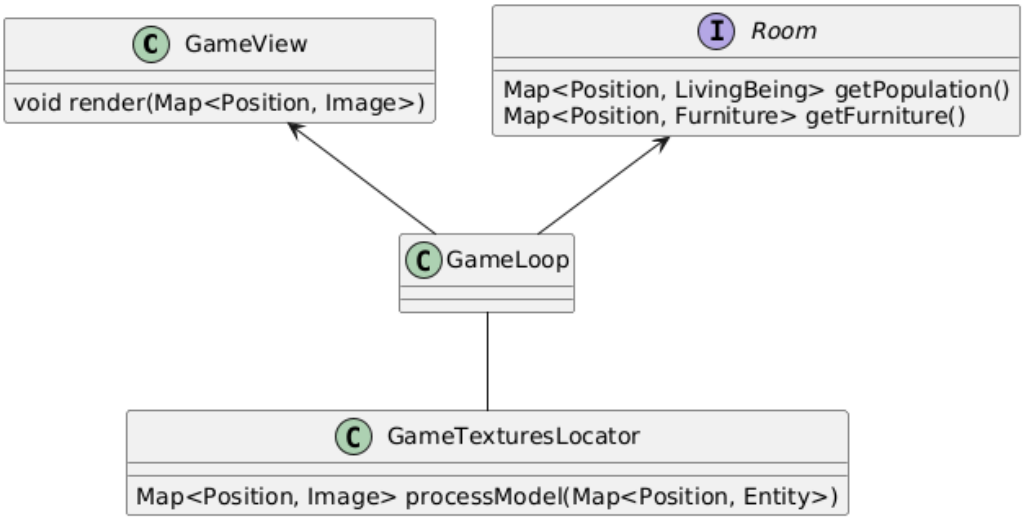

Figura 2.15: Schema UML delle classi e interfacce coinvolte

Problema Durante la fase di esplorazione del gioco, ad ogni frame, il contenuto della stanza deve essere renderizzato traducendo le entità del Model in elementi rappresentabili dalla View.

Soluzione Per risolvere il problema riducendo al minimo le dipendenze, si è introdotta una classe che fa da Locator per le texture. In questo modo, il Controller, reificato in *GameLoop*, può accedere al Model per interrogarlo sul contenuto della stanza, e inviare il contenuto del Model processato dal Locator alla View per il rendering. Ciò significa che il Main Loop rimane completamente agnostico rispetto al formato richiesto dalla View, delegando il compito di traduzione da classi di Model a risorse di View al Locator. In questo modo, una sostituzione anche in blocco della View comporterebbe necessità di modifica solo all'interno del Locator, evitando così di impattare Controller e Model.

2.2.4 Sara Panfini

Gestione dei Nemici

Problema Il problema da affrontare è la gestione dei nemici, che possono avere caratteristiche variabili, come la vita, il danno o gli oggetti che offrono come ricompensa dopo un combattimento, in base alla loro tipologia e livello di difficoltà. Inoltre, la creazione dei nemici dovrebbe essere gestita in modo da poter essere facilmente estendibile, così da consentire l'aggiunta di nuove tipologie di nemici senza dover necessariamente modificare il codice già esistente.

Soluzione La soluzione adottata è quella di utilizzare il *Factory Pattern* e implementare una factory di nemici nell'interfaccia *EnemyFactory* e nella sua relativa implementazione *EnemyFactoryImpl*.
Inoltre, sono state sfruttate delle enumerazioni per gestire le varie tipologie e difficoltà dei nemici.
Questa soluzione è stata scelta per rendere la creazione più flessibile e modulare, centralizzando la logica di creazione dei nemici in modo da avere un sistema maggiormente scalabile.

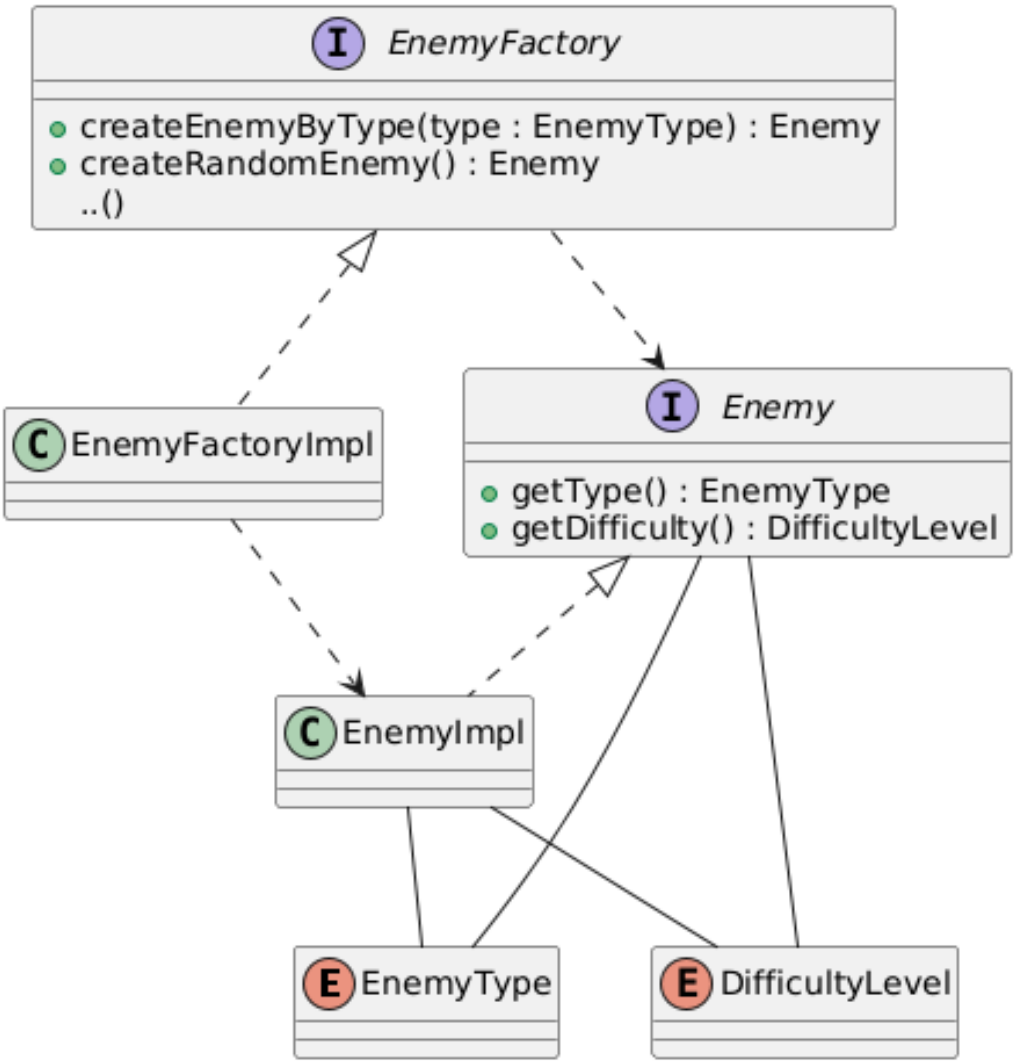

Figura 2.16: Schema UML delle classi e interfacce coinvolte

Gestione degli NPC neutri

Problema Durante il gioco gli NPC sono in grado di supportare diversi tipi di interazione; ad esempio, possono regalare oggetti o assegnare le quest al giocatore. Il sistema dovrebbe, quindi, gestire i diversi comportamenti in maniera efficiente, cioè in modo che sia possibile introdurre facilmente nuove tipologie di interazioni.

Soluzione Per riuscire a gestire al meglio i comportamenti variabili degli NPC, è stato utilizzato lo *Strategy Pattern*. In questo modo si riescono a

stabilire diverse tipologie di comportamenti, che questi personaggi possono assumere in risposta all'interazione con il giocatore. In particolare, l'interfaccia *NpcBehavior* offre un metodo base per interagire con tutti gli NPC e deve essere poi implementata dalle classi che vanno a gestire le specifiche tipologie di comportamento (*DefaultBehavior*, *LootBehavior*, ...).
Inoltre, per crearli (con i comportamenti predefiniti) viene nuovamente utilizzato il *Factory Pattern*.
Questa soluzione implementativa rende la gestione degli NPC facilmente estendibile, dato che per definire ulteriori comportamenti si possono direttamente creare nuove particolari implementazioni dell'interfaccia *NpcBehavior* e l'intera logica di creazione degli NPC si trova nella classe *NpcFactoryImpl*.

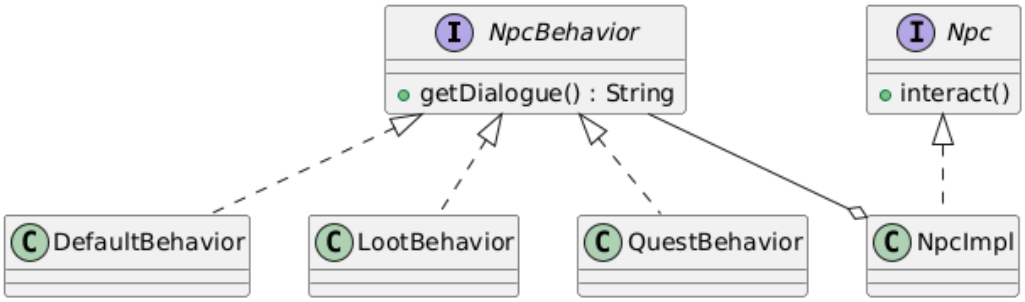

Figura 2.17: Schema UML delle classi e interfacce coinvolte

Gestione della mappa di gioco

Problema La problematica da affrontare è la gestione delle stanze del castello (mappa del gioco). Durante la sua avventura, il giocatore deve, infatti, essere in grado di esplorare le diverse stanze, ognuna delle quali ha al suo interno elementi di arredo e personaggi non giocanti con cui è possibile interagire. Il sistema deve, perciò, essere in grado di garantire un corretto posizionamento e anche una distribuzione logica di tutte le entità entro i confini della stanza.
In aggiunta, è necessario avere un sistema che gestisca l'eventuale assegnazione delle quest a ciascun ambiente di gioco.

Soluzione La soluzione proposta prevede l'implementazione dell'interfaccia *Room* e della relativa classe. Questa classe fornisce la rappresentazione di una generica stanza e si occupa di mantenere tutte le entità presenti e la loro relativa posizione al suo interno, sfruttando anche l'enumerazione *CellType* che gestisce lo stato di ogni cella della mappa.

La creazione delle stanze viene realizzata dalla classe *RoomGenerator*, che ne gestisce anche il popolamento e l'eventuale assegnazione delle quest.
Per gestire le quest è stata realizzata la classe *QuestManager*, che si occupa di associare o meno una specifica missione ad ogni stanza. Questa soluzione permette di racchiudere in una unica classe l'intera logica di assegnazione delle quest e ne facilita anche l'introduzione di nuove tipologie.
Per popolare ciascuna stanza vengono usate le classi *FurnitureGenerator* e *LivingBeingsGenerator*. Queste classi si occupano della generazione e della disposizione, in maniera pseudo-casuale, di tutte le entità - attraverso l'uso delle specifiche *Factory* -, verificando la validità e la disponibilità delle posizioni che gli vengono assegnate. La classe *LivingBeingsGenerator*, inoltre, sfrutta una logica di suddivisione di ciascuna stanza in aree per fare in modo che la distribuzione dei personaggi sia in linea di massima equilibrata. Con questo approccio si è cercato di ottenere una generazione delle stanze abbastanza dinamica, che consente, cioè, di averne configurazioni diverse, e di centralizzare la logica necessaria in modo da agevolare anche future possibili modifiche.

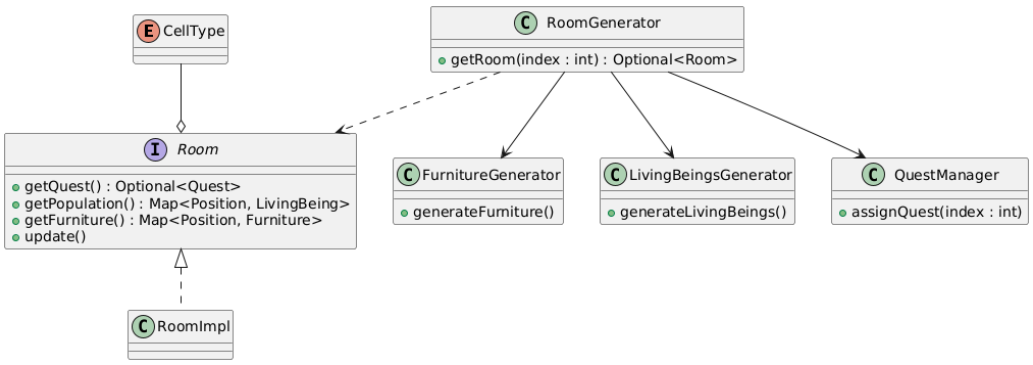

Figura 2.18: Schema UML delle classi e interfacce coinvolte

Gestione delle Quest

Problema Il problema è gestire le varie tipologie di quest assegnate alle stanze del gioco. È necessario creare un sistema che consenta di definire e verificare il completamento di ogni tipo di quest cercando di evitare ripetizioni di codice e di avere una buona estendibilità.

Soluzione Come per gli NPC, anche in questo caso si è fatto uso dello *Strategy Pattern* per modellare il concetto di obiettivo. Si ha, quindi, l'interfaccia *ObjectiveStrategy* che viene implementata dalle classi che rappresentano gli

specifici obiettivi da completare per superare le quest. In questo modo, la singola classe *QuestImpl* riesce a gestire tutte le tipologie di quest, che vengono poi create con la *QuestFactory*.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per quanto riguarda il testing automatizzato, si è sfruttata la libreria JUnit e si sono realizzati test su quasi tutte le classi di Model e Controller. Ciò è stato fatto per garantire il corretto funzionamento dell'applicazione e per avere la certezza che gli eventuali problemi riscontrati durante il gioco non fossero dovuti a mancanze di logica implementativa, quanto a problematiche di visualizzazione.
La View, invece, è stata testata manualmente in fase di sviluppo, e poi in fase di collaudo del software, perché personalmente non siamo riusciti ad approfondire le dinamiche di testing automatizzato che avrebbero permesso di testare automaticamente anche quella parte del software.
In linea generale, gli aspetti su cui il testing si è maggiormente soffermato sono stati i seguenti:

- Generazione della mappa, movimento e interazioni al suo interno. In particolare, si è verificato che il movimento e le interazioni non generassero comportamenti imprevisti e che si riflettessero correttamente sugli aggiornamenti del Model.
- Combattimento. In particolare, il testing si concentra sul mantenimento dell'ordine dei turni, per evitare comportamenti imprevisti, e sul corretto inserimento in inventario del bottino dei nemici sconfitti.
- Protagonista. In particolare, si è testato il sistema di gestione della vita del protagonista, così come del suo inventario. Nei controlli sull'inventario, si è verificato il corretto utilizzo degli oggetti curativi, così come delle armi e armature, per garantire comportamenti consistenti in fase di combattimento.

- Input utente. Si sono testati i Controller che svolgono la funzione di Observer, per garantire la corretta gestione degli input.
- Creazione delle istanze. Avendo fatto largo uso del *Factory Pattern*, si è deciso di testare i metodi di generazione degli oggetti per verificare l'effettiva consistenza delle istanze create e garantire prevedibilità in fase di gioco.

3.2 Note di sviluppo

3.2.1 Lorenzo Cinelli

- **Espressioni Lambda:** Utilizzate nella classe *InventoryControllerImpl* nei metodi *notify* e *regress* per gestire rispettivamente gli eventi osservati sul key listener ed il cambio di stato del gioco in uscita dall'inventario. Permalink: `https://github.com/KimboCoffee/OOP23-relario/blob/18fd3851aabd2d45b26bb726cd73f204721d9190/src/main/java/it/unibo/oop/relario/controller/impl/InventoryControllerImpl.java#L57-L77` Permalink: `https://github.com/KimboCoffee/OOP23-relario/blob/18fd3851aabd2d45b26bb726cd73f204721d9190/src/main/java/it/unibo/oop/relario/controller/impl/InventoryControllerImpl.java#L138-L142`
- **Stream ed espressione Labmda:** Utilizzati nella classe *InventoryControllerImpl* nel metodo *getItemsNames* per restituire una lista di stringhe da una lista di oggetti di inventario. Permalink: `https://github.com/KimboCoffee/OOP23-relario/blob/18fd3851aabd2d45b26bb726cd73f2047` `src/main/java/it/unibo/oop/relario/controller/impl/InventoryControllerImpl.java#L83-L86`
- **Optional:** Utilizzati nella classe *InventoryItems* nel metodo *getEquippedItem* per poter passare il campo optional dell'inventario sia per l'armatura che per l'arma, i quali possono essere equipaggiati come no. Permalink: `https://github.com/KimboCoffee/OOP23-relario/blob/18fd3851aabd2d45b26bb726cd73f204721d9190/src/main/java/it/unibo/oop/relario/model/inventory/InventoryItems.java#L33-L34`
- **Reflection:** Utilizzata nella classe *InventoryItems* nel metodo *getDurability* per controllare se l'oggetto di inventario passato sia un oggetto equipaggiabile, e di conseguenza se ha una durabilità. Permalink: `https://github.com/KimboCoffee/OOP23-relario/blob/18fd3851aabd2d45b26bb726c`

src/main/java/it/unibo/oop/relario/model/inventory/InventoryItems.java#L51

- **Espressioni Lambda:** Utilizzate nella classe *CutSceneControllerImpl* nei metodi *show*, *progressGame* e *progressView* utilizzate rispettivamente per mostrare la scena corretta in base allo stato del gioco, caricare il controller successivo mentre viene mostrata la scena e mostrare la view successiva in base allo stato successivo una volta terminata la scena. Permalink: https://github.com/KimboCoffee/OOP23-relario/blob/18fd3851aabd2d45b26bb726cd73f204721d9190/src/main/java/it/unibo/oop/relario/controller/impl/CutSceneControllerImpl.java#L27-L56

- **Espressioni Lambda:** Utilizzate nelle classi *CutSceneViewImpl*, *CombatScene* e *CombatAnimationImpl* per indicare l'azione da far eseguire ai timer una volta scaduti. Permalink: https://github.com/KimboCoffee/OOP23-relario/blob/18fd3851aabd2d45b26bb726cd73f204721d9190/src/main/java/it/unibo/oop/relario/view/impl/CutSceneViewImpl.java#L84 Permalink: https://github.com/KimboCoffee/OOP23-relario/blob/18fd3851aabd2d45b26bb726cd73f204721d9190/src/main/java/it/unibo/oop/relario/view/impl/CutSceneViewImpl.java#L94-L97 Permalink: https://github.com/KimboCoffee/OOP23-relario/blob/18fd3851aabd2d45b26bb726cd73f204721d9190/src/main/java/it/unibo/oop/relario/view/impl/CutSceneViewImpl.java#L122-L125 Permalink: https://github.com/KimboCoffee/OOP23-relario/blob/18fd3851aabd2d45b26bb
src/main/java/it/unibo/oop/relario/view/impl/CombatScene.java#L117 Permalink: https://github.com/KimboCoffee/OOP23-relario/blob/18fd3851aabd2d45b26bb726cd73f204721d9190/src/main/java/it/unibo/oop/relario/view/impl/CombatAnimationImpl.java#L55

- **Espressione Lambda:** Utilizzata nella classe *CombatAnimationImpl* nel costruttore per gestire la risorsa da utilizzare come animazione. Permalink: https://github.com/KimboCoffee/OOP23-relario/blob/18fd3851aabd2d45b26bb726cd73f204721d9190/src/main/java/it/unibo/oop/relario/view/impl/CombatAnimationImpl.java#L46-L50

Le classi *ImageLocators* e *SoundLocators*, per caricare risorse da file, sono state scritte prendendo spunto da Internet. Permalink: https://github.com/KimboCoffee/OOP23-relario/blob/18fd3851aabd2d45b26bb726cd73f204721d9190/src/main/java/it/unibo/oop/relario/utils/impl/ImageLocators.java#L23-L43 Permalink: https://github.com/KimboCoffee/OOP23-relario/blob/18fd3851aabd2d45b26bb726cd73f204721d9190/src/main/java/it/unibo/oop/relario/utils/impl/SoundLocators.java#L28-L50

3.2.2 Kimi Osti

Di seguito si presentano singoli esempi di uso di costrutti avanzati di Java. Ciò non impedisce che all'interno del codice appaiano più situazioni in cui vengono usati.

- **Java Wildcards** al permalink `https://github.com/KimboCoffee/OOP23-relario/blob/d67fe167022e08cef63d110da0710154eb543292/src/main/java/it/unibo/oop/relario/utils/impl/GameTexturesLocator.java#L42`
- **Lambda Expressions e Optional** al permalink `https://github.com/KimboCoffee/OOP23-relario/blob/d67fe167022e08cef63d110da0710154eb543292/src/main/java/it/unibo/oop/relario/model/inventory/InventoryImpl.java#L40`
- **InputStream** al permalink `https://github.com/KimboCoffee/OOP23-relario/blob/d67fe167022e08cef63d110da0710154eb543292/src/main/java/it/unibo/oop/relario/view/impl/UserGuide.java#L54`
- **Gestione dei Thread** al permalink `https://github.com/KimboCoffee/OOP23-relario/blob/cb9e0fd37a803cebfb509ff1b08d7df99b5ee8fc/src/main/java/it/unibo/oop/relario/controller/impl/GameLoop.java`

Codice reperito online

Per la realizzazione della classe BackgroundTile (Permalink: `https://github.com/KimboCoffee/OOP23-relario/blob/d67fe167022e08cef63d110da0710154eb543292/src/main/java/it/unibo/oop/relario/view/impl/BackgroundTile.java`) si è preso spunto da questa pagina di forum online.

3.2.3 Mazuru Mihai

- **Stream**: Utilizzato per il recuperare il valore corrispondente ad una nota chiave in una mappa. Permalink `https://github.com/KimboCoffee/OOP23-relario/blob/d67fe167022e08cef63d110da0710154eb543292/src/main/java/it/unibo/oop/relario/view/impl/MainViewImpl.java#L84-L95`
- **Stream**: Utilizzato per filtrare gli arredamenti in base al loro tipo (Walkable, Obstructing, Interactive). Permalink `https://github.com/KimboCoffee/OOP23-relario/blob/d67fe167022e08cef63d110da0710154eb543292/`

src/main/java/it/unibo/oop/relario/model/entities/furniture/impl/FurnitureFactoryImpl.java#L128-L131

- **Lambda function**: Utilizzato per l'Action Listener di JButton. Permalink https://github.com/KimboCoffee/OOP23-relario/blob/d67fe167022e08cef63d src/main/java/it/unibo/oop/relario/view/impl/MenuView.java#L87-L109

- **Optional**: Utilizzo per gestire i nemici all'interno degli arredamenti. Permalink https://github.com/KimboCoffee/OOP23-relario/blob/d67fe167022e08cef63d110da0710154eb543292/src/main/java/it/unibo/oop/relario/model/entities/furniture/impl/WalkableFurnitureImpl.java#L12-L73

Codice preso da altri

Modellazione molteplici menu: Utilizzo model per gestire molteplici menu. Permalink https://github.com/luca-casadei/OOP23-the-exiled/blob/3980cb23272808019d315d20593faae31b543f89/src/main/java/unibo/exiled/model/menu/MenuModelImpl.java#L6-L33

3.2.4 Sara Panfini

Seguono alcuni esempi di parti di codice in cui vengono usate funzionalità avanzate.

- **Espressioni Lambda e Stream**
  https://github.com/KimboCoffee/OOP23-relario/blob/cb9e0fd37a803cebfb509ff1b0 src/main/java/it/unibo/oop/relario/model/inventory/InventoryItemFactoryImpl.java#L84-L86
  https://github.com/KimboCoffee/OOP23-relario/blob/cb9e0fd37a803cebfb509ff1b0 src/main/java/it/unibo/oop/relario/model/map/RoomImpl.java#L158-L161
  https://github.com/KimboCoffee/OOP23-relario/blob/cb9e0fd37a803cebfb509ff1b0 src/main/java/it/unibo/oop/relario/model/quest/DefeatEnemyObjective.java#L32-L35

- **Optional**
  https://github.com/KimboCoffee/OOP23-relario/blob/cb9e0fd37a803cebfb509ff1b0 src/main/java/it/unibo/oop/relario/model/map/RoomGenerator.java#L57-L58
  https://github.com/KimboCoffee/OOP23-relario/blob/cb9e0fd37a803cebfb509ff1b0

src/main/java/it/unibo/oop/relario/model/map/FurnitureGenerator.
java#L55

- **Record**
https://github.com/KimboCoffee/OOP23-relario/blob/cb9e0fd37a803cebfb509ff1b
src/main/java/it/unibo/oop/relario/model/entities/enemies/EnemyFactoryImpl.
java#L100

- **Java Wildcards**
https://github.com/KimboCoffee/OOP23-relario/blob/cb9e0fd37a803cebfb509ff1b
src/main/java/it/unibo/oop/relario/model/quest/ObjectiveStrategy.
java#L25

Codice preso da altri

Per la realizzazione della classe *Pair*: https://bitbucket.org/mviroli/
oop2023-esami/src/c5edbdde70b6a4ead339098297c82ca7372c96f6/a01a/
e1/Pair.java#lines-9:55

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Lorenzo Cinelli

Sono molto incerto sull'essere soddisfatto o meno di questo progetto, in quanto vedo il mio contributo molto marginale. Ho sempre cercato di dare il massimo sia in fase di analisi, che di design, progettazione e sviluppo, e sono soddisfatto di ciò che ho fatto, tuttavia ritengo che il carico di lavoro non sia stato minimamente bilanciato, specialmente sulla suddivisione di Model (che nella pratica non ho avuto). Di fatti appena cominciato a pensare al design della mia parte del progetto mi sono reso conto di avere una parte molto più ridotta rispetto agli compagni (specialmente nel Model come detto sopra), ma nonostante l'abbia cercato di far notare più volte per provare a suddividere nuovamente e meglio il lavoro non sono stato preso in considerazione seriamente. A causa di questo ritengo anche che questo progetto non mi abbia fatto migliorare quanto mi aspettassi, nonostante comunque abbia imparato molte cose. Escludendo l'inconveniente della suddivisione del lavoro, con gli altri membri del gruppo mi sono trovato molto bene. È stato facile discutere soluzioni implementative e collaborare per parti comuni. Del progetto complessivo posso dirmi mediamente soddisfatto, in quanto nonostante le problematiche siamo riusciti a portare a termine il software in gruppo - questo è il primo progetto grosso che svolgo in gruppo - ed in quanto il tema videogiochi è uno dei temi a cui sono maggiormente interessato.

4.1.2 Mazuru Mihai

Spero che il mio contributo all'interno del team sia stato significativo. All'inizio, a causa dell'ansia per questo progetto, ho avuto difficoltà a essere

d'aiuto nelle fasi decisionali. Tuttavia, con il tempo, man mano che acquisivo maggiore confidenza con il progetto e con le persone con cui lavoravo, sono riuscito a sbloccarmi e a mettermi in gioco.

Mi auguro inoltre di aver trasmesso agli altri membri del gruppo la mia determinazione, dimostrando il mio impegno nel partecipare a ogni incontro e contribuendo attivamente alla realizzazione del MainController e della MainView.

Per quanto riguarda il resto del team, li considero colleghi straordinari e persone sempre disponibili, sia nel fornire chiarimenti che nel darmi una mano quando ne avevo bisogno.

4.1.3 Kimi Osti

Personalmente, questo è stato il primo progetto di gruppo di queste dimensioni a cui mi sono dedicato, e in particolare è stato il primo progetto a cui mi sono dedicato in ambito di videogiochi, tema che mi ha sempre molto affascinato e a cui vorrei dedicare anche la mia carriera futura. In particolare per questo motivo, ho trovato stimolante la fase di sviluppo del software e particolarmente gratificante riuscire a produrre un risultato finale funzionante.
Per quanto riguarda lo sviluppo del software - guardando indietro alle fasi iniziali del progetto - posso dire di ritenermi abbastanza soddisfatto. Durante il corso del progetto ho principalmente appreso alcune delle dinamiche nello sviluppo di software di dimensioni più grandi rispetto ai soliti progetti individuali a cui mi ero dedicato in precedenza, e ho anche capito alcuni concetti che - prima di iniziare a lavorare su questo progetto - non avrei saputo affrontare. Pertanto, posso dire che questo progetto mi è servito individualmente a maturare ulteriormente, e che se dovessi tornare indietro probabilmente lavorerei in modo diverso non tanto per modificare il risultato finale (che si è dimostrato abbastanza vicino a quello che mi ero prefigurato prima di iniziare il lavoro), quanto più in virtù delle dinamiche che ho appreso durante il lavoro e che avrebbero permesso di produrre software in maniera più efficiente.
Per quanto riguarda le dinamiche di gruppo, posso dirmi abbastanza soddisfatto, soprattutto per il fatto che, anche se in alcune fasi vi sono state divergenze di vedute su alcune tematiche, non si sia mai arrivati a situazioni critiche che abbiano pregiudicato lo svolgimento del lavoro. Fin da prima di presentare il progetto, però, sono stato io a presentare l'idea su cui poi si è sviluppato il gioco, e durante il progetto - complice la fiducia che i miei compagni hanno riposto in me - ho assunto il ruolo di "coordinatore" del lavoro. Questo ha però inevitabilmente significato che gran parte delle idee

su cui si basa il modello del gioco siano arrivate da me, e che poi io mi sia trovato - soprattutto nelle fasi finali - ad affrontare personalmente le fasi di sviluppo più delicate, come per esempio la soluzione di possibili bug sottolineati dagli strumenti di analisi statica del codice oppure la soluzione del problema di mancato caricamento delle risorse che si è presentata quando abbiamo assemblato il jar per la consegna finale.
Nel complesso però, mi sento di dire che il risultato finale rispecchia abbastanza l'idea che avevo del gioco prima di iniziare lo sviluppo, e che ho trovato stimolante il processo di realizzazione. Per come è strutturato il software ci sarebbe margine per allargarlo e portarlo a qualcosa di più di una demo da qualche minuto, anche se dubito che possa accadere. D'altro canto, personalmente vedo rafforzata la mia passione verso il mondo dello sviluppo di videogiochi, e sono volenteroso di portare avanti questa passione anche con progetti personali che possano permettermi di approfondire ulteriormente parti dello sviluppo che per motivi di suddivisione del lavoro ho toccato meno.

4.1.4 Sara Panfini

Questo progetto è stato sicuramente uno degli scogli più grandi che ho dovuto affrontare nel mio percorso fino ad ora.
Inizialmente, non mi sentivo preparata e non credevo di avere le conoscenze per affrontare un progetto di questa portata. Ciò ha reso le prime fasi di lavoro particolarmente complesse per me e, perciò, durante quel periodo sento di non aver contribuito abbastanza e di non essere stata di grande aiuto al gruppo. Con il tempo, però, sono riuscita a prendere consapevolezza e ritengo di essere stata in grado di apportare un buon contributo al progetto, anche se avrei sicuramente potuto gestire meglio alcune cose, come l'organizzazione del lavoro e la gestione dei tempi.
Nonostante, generalmente, io non mi trovi a mio agio con i lavori di gruppo, considero questa comunque come un'esperienza, nel complesso, positiva. Ci sono, ovviamente, state opinioni contrastanti su alcuni aspetti dello sviluppo del progetto, ma, nonostante ciò, abbiamo sempre cercato di lavorare insieme nel rispetto degli altri componenti del gruppo.
In generale, quindi, questo progetto ha rappresentato una grande sfida per me e, a prescindere dalle varie difficoltà, sono soddisfatta di come sono riuscita ad affrontarla. Lavorare allo sviluppo di un sistema di queste dimensioni mi ha stimolato molto e mi ha anche iniziato al mondo dei videogiochi, che non conoscevo e che ora mi affascina molto. Nel complesso, credo che questa esperienza mi abbia fatto imparare molto e mi abbia fornito gli strumenti

per riuscire ad affrontare progetti futuri con maggiore preparazione e con un approccio più consapevole.

4.2 Difficoltà incontrate e commenti per i docenti

Permettere la consegna anche durante i periodi di lezione.

Appendice A

Guida utente

Durante il gioco l'utente deve esplorare le stanze del castello e eventualmente completare una quest per poter passare alla stanza successiva.
Nei menu, come nel combattimento, l'input viene accettato tramite mouse, con dei clic sui bottoni che rappresentano l'azione desiderata.
Durante l'esplorazione, il giocatore si sposta con WASD, interagisce con la cella che guarda con E e apre l'inventario con I.
Per superare ciascuna stanza, deve completare la quest (se presente) che gli viene presentata dall'NPC fermo all'ingresso. Dopo aver completato la quest, deve raggiungere il tappeto situato a destra della stanza, che corrisponde all'uscita, ed interagire da sopra la casella al suo estremo destro per poter proseguire nella stanza successiva.
All'interno dell'inventario, gli oggetti vengono scelti con le frecce verticali, e si possono equipaggiare con ENTER oppure scartare con BACKSPACE. Un oggetto che non ha effetto, quando viene usato, viene "consumato" e non risulta quindi piu' disponibile.

Bibliografia

[Nys14] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014.