Meta-relazione per "Programmazione ad Oggetti"

Bekic Dario, Bresciani Enea, Debertolis Lorenzo, Tamburini Federico

20 giugno 2023

Indice

Capitolo 1

Analisi

Requisiti

Il software mira alla costruzione di un videogame, dal nome Turn Base Tunnels. Il gioco si ispira ad altri titoli, avendo un sistema di esplorazione del mondo di gioco comune ai Dungeon Crawler[1], e un sistema di combattimento comune ai Conditional Turn-Based RPG[2].

Requisiti funzionali

- Il gioco deve prevedere il movimento del giocatore tra ambienti differenti.
- Il gioco deve prevedere la raccolta di oggetti in varie situazioni, ad esempio tramite interazioni con NPCs o a seguito di un combattimento.
- Il gioco deve prevedere meccaniche di personalizzazione del party, e degli equipaggiamenti dei vari personaggi.
- Il gioco deve avere una (o più) condizioni di fine.

Requisiti non funzionali

- L'applicativo dovrà essere compatibile sui principali sistemi operativi.

[1] `https://it.wikipedia.org/wiki/Dungeon_crawler`
[2] `https://it.wikipedia.org/wiki/Conditional_Turn_Based`

1.1 Analisi e modello del dominio

Il gioco è ambientato in stanze navigabili e dove è possibile avere interazione con diverse entità.
Queste entità variano da personaggi che offrono oggetti da poter acquistare a portali che portano in altri ambienti del gioco.
Particolari interazioni con personaggi "ostili" provocheranno l'inizio di una sequenza di gioco che porterà ad un combattimento a turni.
Il giocatore potrà utilizzare un team di personaggi che si differenziano per abilità e statistiche, e potranno essere equipaggiati con oggetti di vario tipo, come ad esempio armi o armature, per variarne abilità e statistiche.

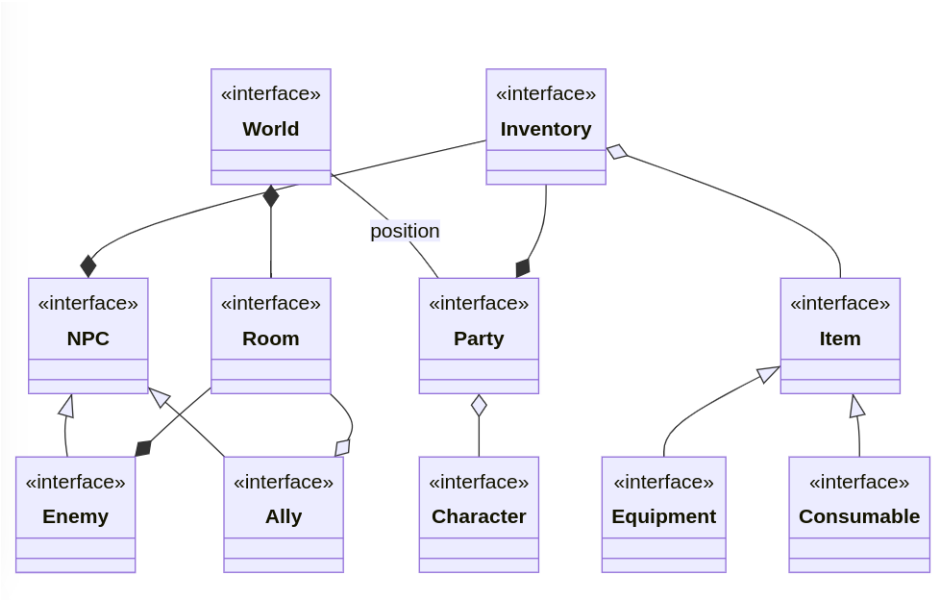

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Il pattern utilizzato per l'architettura è MVC.
Abbiamo 2 Manager principali che dividono le responsabilità: il GameStateManager che coordina la sezione **Model** e il ViewControllerManager che coordina le **View** e i **Controller**.
Il GameStateManager espone un oggetto di tipo StateModel che incapsula tutti gli oggetti riguardanti il model necessari al resto dell'architettura in quel momento.
Il ViewControllerManager prende questo oggetto e lo passa alla GameView e al ViewController attuali.
Il ViewControllerManager si sincronizza con il GameStateManager attraverso una proprietà che identifica il GameState corrente e l'oggetto StateModel che lo rappresenta.
Il ViewControllerManager rappresenta l'entry-point per chi vuole ordinare il rendering della GameView e ottenere la lista di input (incapsulati come Command) che l'associato ViewController ha ricevuto fino a quel istante di tempo. Per maggior informazioni riguardo a come funzionano più nel dettaglio i manager si rimanda al design in dettaglio:
GameStateManager
ViewControllerManager

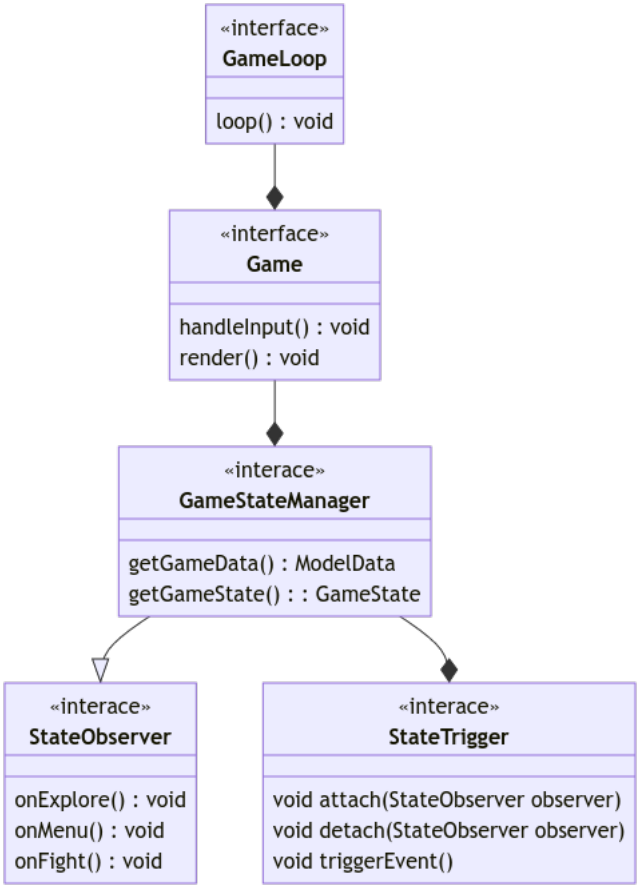

Figura 2.1: UML semplificato della parte Model dell'architettura

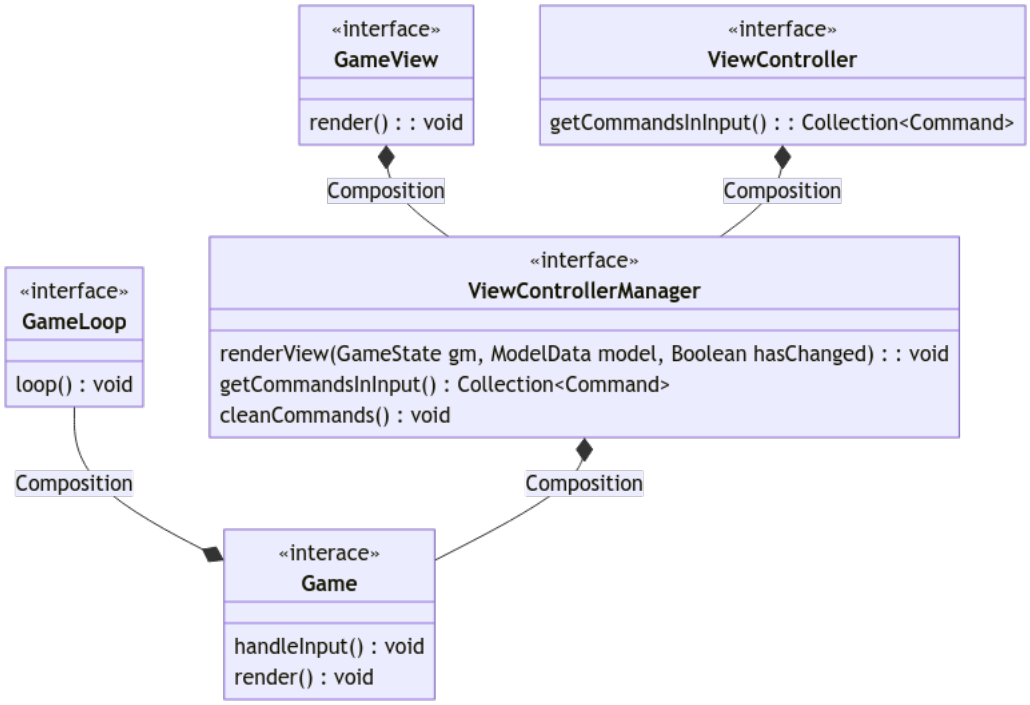

Figura 2.2: UML semplficato della parte View e Controller dell'architettura

2.2 Design dettagliato

2.2.1 Bekic Dario

Game Loop e Controller

Problema: rendere il gameloop separato dal resto dell'architettura per permettere un cambiamento di implementazione dello stesso efficace.

Soluzione: avendo trovato varie implementazioni del Game Loop e considerando la possibilità che potesse variare si è deciso di creare un interfaccia GameLoop che definisca la logica del singolo ciclo di loop. Il GameLoop utilizza l'interfaccia Game per chiamare le funzioni per gestire le operazioni sul gioco al più alto livello di astrazione(Render grafico, gestione dell'input ecc.). Si è deciso inoltre di dividere il componente chiamante del Game Loop dal Game Loop stesso, in modo tale da avere come unica responsabilità per l'oggetto del GameLoop quella di gestire la logica con cui si eseguono le operazioni sul gioco. Il Componente incaricato di chiamare il Game Loop è quindi anch'esso modificabile, senza dover modifacare altri componenti. Il componente incaricato di decidere se chiamare il loop del Game Loop o meno è chiamato GameLoopManager.

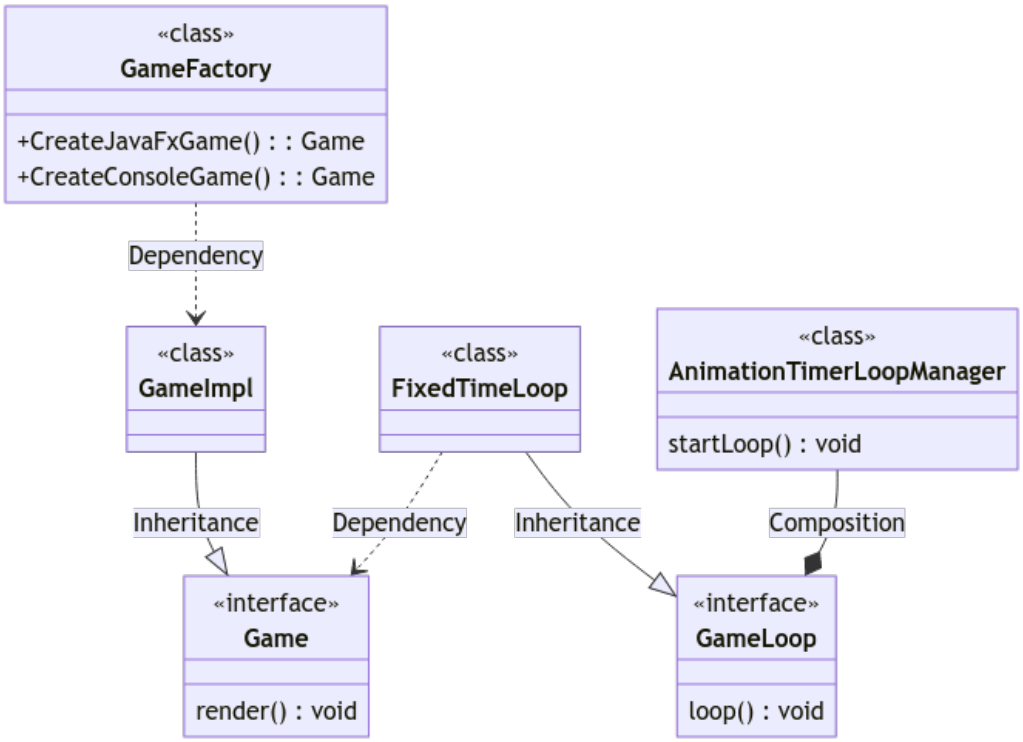

Figura 2.3: Diagramma del Game Loop e le sue relazioni.

GameState e cambiamento di GameState

Problema: è necessario definire un modo flessibile in cui dividere i stati logici dell'applicazione.

Soluzione: ho deciso di dividere l'applicazione in modo tale che sia una Macchina a Stati Finiti. Ad ogni macro situazione è stata associata una entry dell'enumarazione chiamata GameState, che è il riferimento utilizzato da tutte le classi Manager per sincronizzare le proprie azioni. Per la transizioni degli stati è stato utilizzato un TransitionManager, interfaccia che fa da StateObserver cioè definisce dei metodi che vengono utilizzati dagli oggetti che vogliono cambiare lo stato del gioco, per notificare la volontà di modificarlo. Questi oggetti implementano l'interfaccia StateTrigger e durante l'inizializzazione del TransitionManager egli si registra in ogni oggetto che implementa questa interfaccia. Il TransitionManager quindi è responsabile di tenere traccia del GameState attuale e restituire l'oggetto StateModel corrispondente. L'interfaccia StateModel è stata introdotta come Wrapper per tutti gli oggetti di model che sono utili al resto dell'architettura(View e Controller) in quel particolare GameState, seguendo il principio di Interface Segregation, esiste per ogni GameState il corrispondente StateModel.

Di seguito l'UML (semplificato) dove si mostra il legame tra TransitioManager e il ModelManager e dove il Party è usato per mostrare come esso sia stato usato per implementare lo StateTrigger.

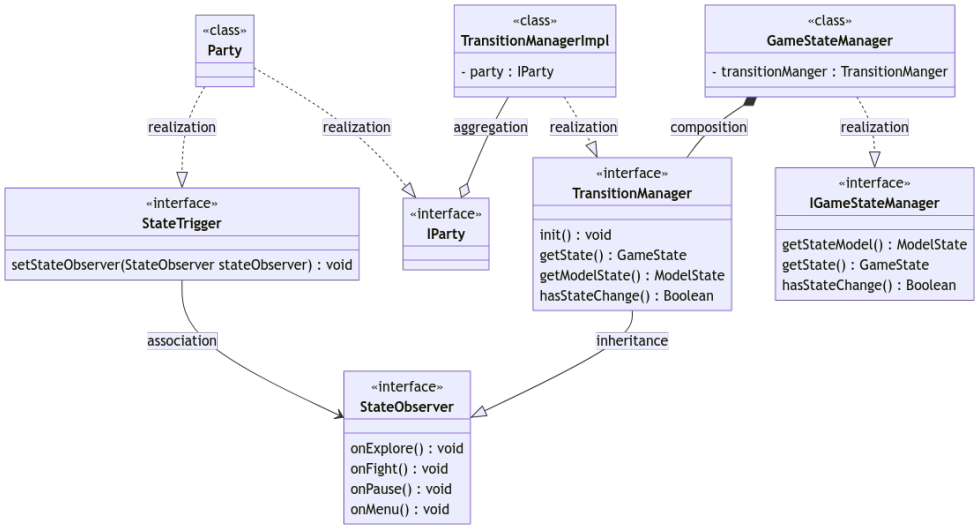

Figura 2.4: Cambio di GameState

GameState e gestione delle View e Input

Problema: separare i framework grafici dalla gestione delle view e dell'input e dal resto dell'architettura.

Soluzione: Per ogni GameState c'è almeno una GameView che lo rappresenta e che verrà creata dalla GameViewFactory ogni qualvolta il GameState o il ModelState cambieranno. Il componente che utilizza la GameViewFactory per creare le GameView è chiamato ViewControllerManager, e ogni qualvolta la GameView viene creata anche un corrispettivo ViewController viene creato e passato alla GameView, per la gestione dell'input. In questo modo si utilizza il pattern *Factory* per delegare la creazione dell'effettiva GameView e il pattern *Strategy* per la massimizzazione del riuso delle componenti in quanto basta modificare l'implementazione della Factory passata al ViewControllerManager per modificare il comparto grafico dell'applicazione.

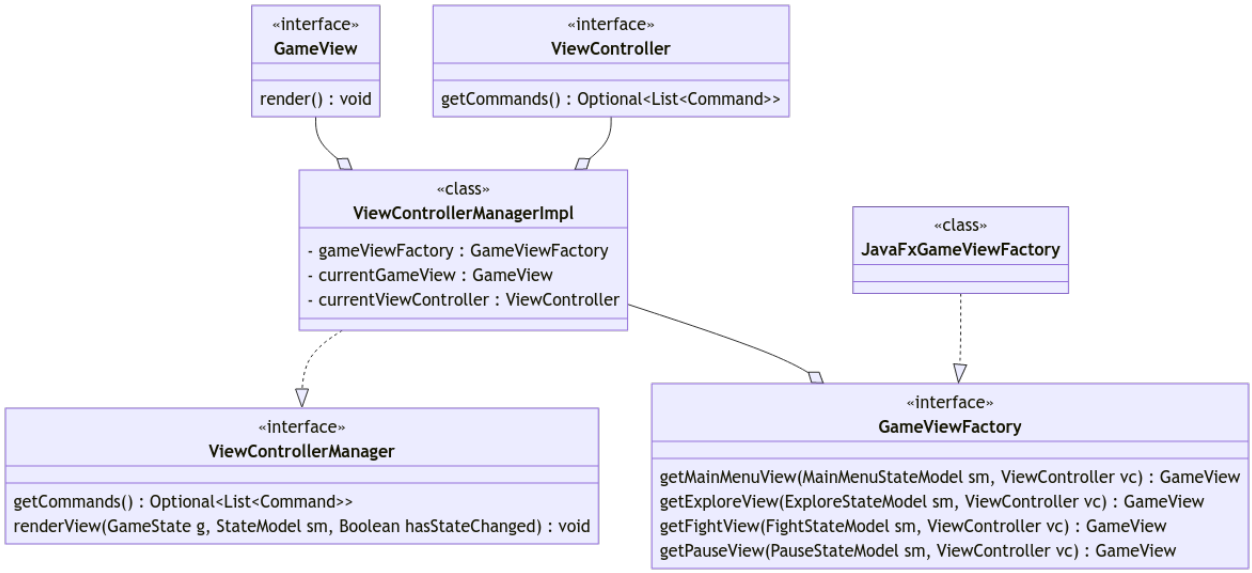

Figura 2.5: ViewControllerManager, GameView e ViewController

Interazione con il mondo

Problema: è necessario un modo per far interagire il giocatore con tutte le entità che può incontrare.

Soluzione: inanzitutto le uniche entità con cui il giocatore può interagire sono oggetti di tipo SpatialEntity, che sono le entità che si trovano fisicamente dentro allo spazio attraversabile dal giocatore. Ho deciso di creare un interfaccia InteractionTrigger che è implementata da tutti gli oggetti che possono interagire, in qualche modo, con l'ambiente. Questo perché volevo distinguere gli oggetti capaci di interagire da quelli con cui si può interagire. Un esempio di InteractionTrigger è l'oggetto che rappresenta il giocatore cioè la classe Party. Non volendo dare troppe responsabilità alla classe Party per quanto riguarda la logica di interazione, ho creato un interfaccia InteractionComponent che incapsula la logica di interazione, su spunto datomi dal pattern *Component*. Ogni InteractionComponent definisce una logica di interazione. Questo è stato fatto per dare più spazio a ridefinizioni future sul concetto di interazione con l'ambiente, per esempio l'implementazione adottata per l'oggetto Party fa uso di un CollisionDetector per capire con quale entità è possibile interagire. Per interagire con un oggetto si ha bisogno di un suo riferimento e che lui implementi l'interfaccia Interactable, che definisce l'azione che un oggetto deve eseguire in caso qualcuno ci abbia interagito. Per dare più dinamicità a questa funzione si passa anche la SpatialEntity che ha causato l'interazione, per eventuali modifiche che l'oggetto con cui si è interagito vuole apportare.

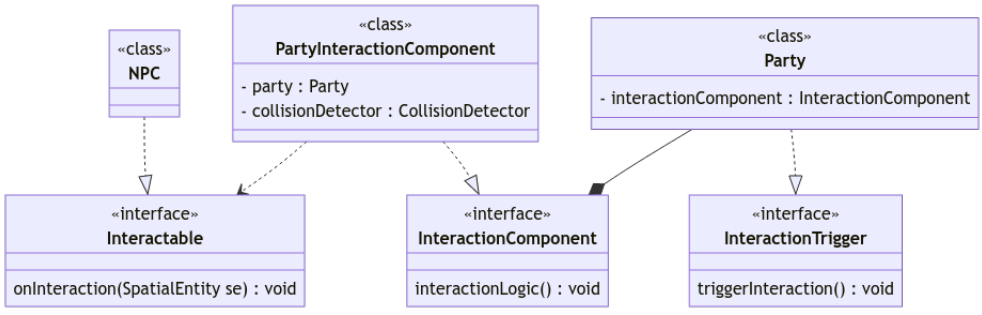

Figura 2.6: Interazione con il mondo

Il mondo e il movimento

Problema: avere una sistema per rappresentare il mondo e per muoversi in questo mantenendo una struttura semplice.

Soluzione: l'oggetto World è un insieme di oggetti Room e le Room contengono un insieme di SpatialEntity, che sono le entità che si trovano nella stanza. Per muoversi ho deciso di riutilizzare il meccanismo dell'interact creato per ogni entità nella stanza. Ho creato un oggetto di tipo RoomLink che implementa Interactable e nel suo onInteraction controlla se la SpatialEntity che lo ha provocato è un oggetto IParty e, se lo è, modifica il riferimento della sua stanza con il metodo setCurrentRoom(Room room). L'oggetto RoomLink contiene il riferimento a 2 Room per rappresentare il collegamento tra queste. In questo modo è possibile creare collegamenti unidirezionali(mettendo il RoomLink in una sola stanza del World) oppure usare diverse logiche di accesso all'altra Room oppure avere più di 2 Room collegate.

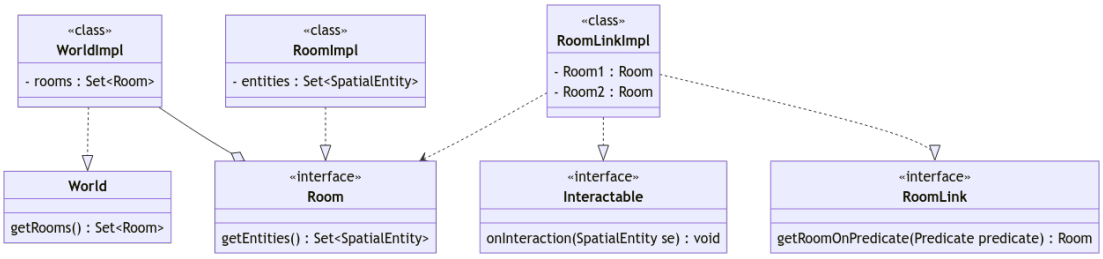

Figura 2.7: Il mondo e il movimento

2.2.2 Bresciani Enea

Menu modulare

Problema: I menu dovranno essere logicamente distaccati dalla view e permettere una loro costruzione modulare a model.

Soluzione: Il sistema per la gestione dei vari componenti di un menu si utilizza il *pattern Composite*. Le implementazioni di `MenuItem` possono essere modificate, e la modifica impatta direttamente sul comportamento del menu.

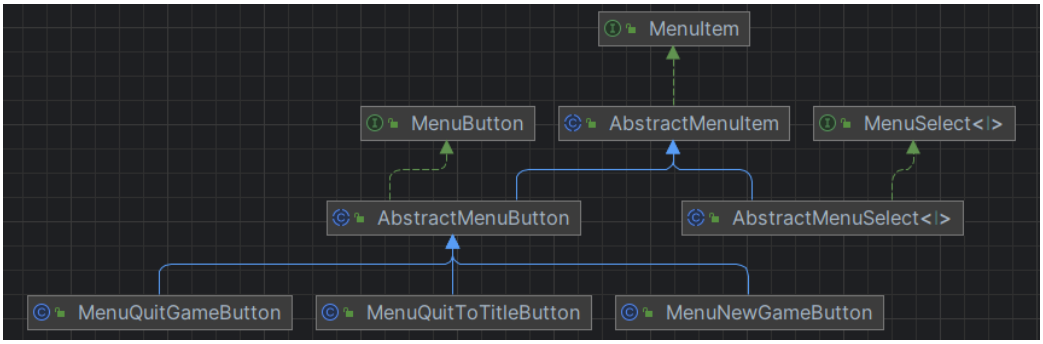

Figura 2.8: Rappresentazione UML del pattern Composite per i Menu di TBT

Factory degli NPC

Problema: In fase di sviluppo, sono state sviluppati più tipologie di npc, ognuno dei quali ha un comportamento peculiare.

Soluzione: Dato che i tipi di NPC esistenti potrebbero variare nel tempo (aggiungerne uno non compromette la struttura), si è ritenuto necessario utilizzare il Factory Method per centralizzare la creazione di tutte le implementazioni di NPC.

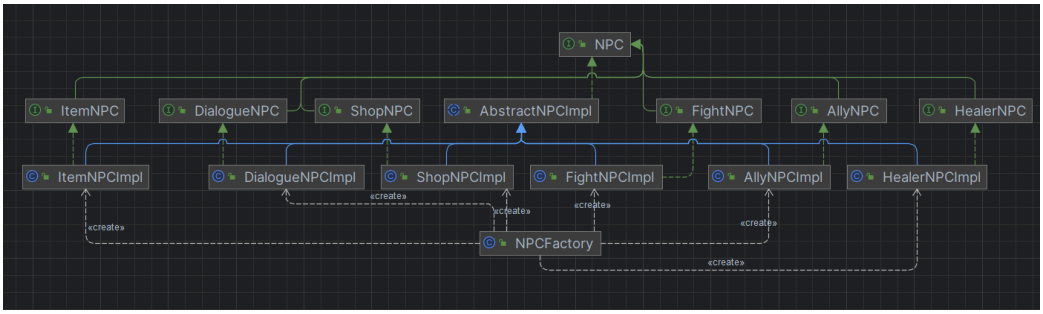

Figura 2.9: Rappresentazione UML del pattern Composite per i Menu di TBT

2.2.3 Debertolis Lorenzo

Entities

Problema: modellizzare le varie entità del gioco (personaggi e oggetti, principalmente) in modo modulare

Soluzione: è stata creata una gerarchia tra classi, identificando le funzionalità e caratteristiche chiave delle varie entità così da poterle raggruppare in modo efficiente per eliminare la duplicazione di codice. Sono quindi state individuate 3 tipologie di entità:

- entità semplici, "adimensionali", che non devono comparire sulla mappa di gioco
- entità con una posizione nella stanza e con una dimensione, ma che non si possono spostare
- entità con posizione e dimensione, e che possono essere spostate

Allo stesso modo sono stati organizzati gli oggetti (Item) e i personaggi (Character) in modo da poter massimizzare il riutilizzo di codice e permettere il raggruppamento di più classi che devono avere un comportamento simile.

La maggiorparte delle entità vengono generate tramite Factory, gli item in particolare vengono generati in modo singleton, per evitare di avere molteplici istanze dello stesso identico oggetto con il relativo spreco di RAM. Queste stesse factory sono state scritte in modo da caricare i campi delle classi leggendo le informazioni contenute in file json.

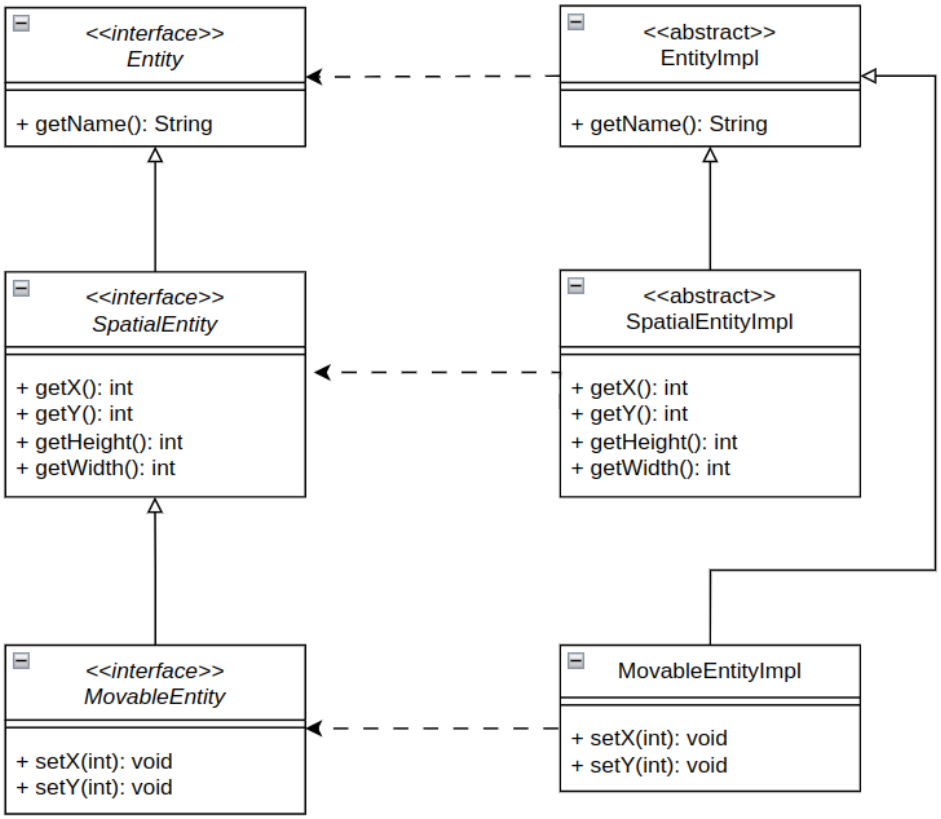

Figura 2.10: Rappresentazione UML delle classi relative alle entità

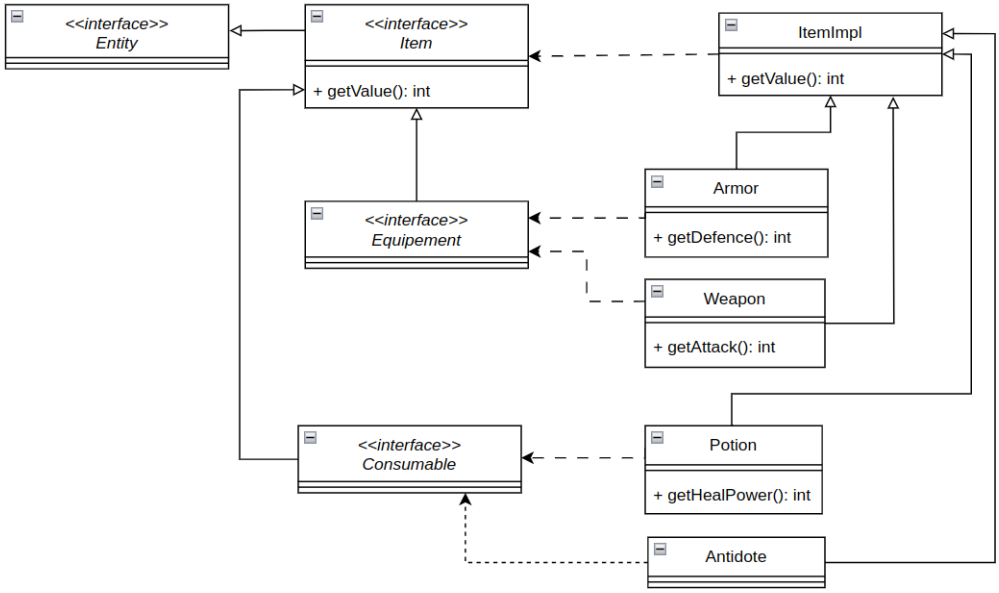

Figura 2.11: Rappresentazione UML delle classi relative agli Item

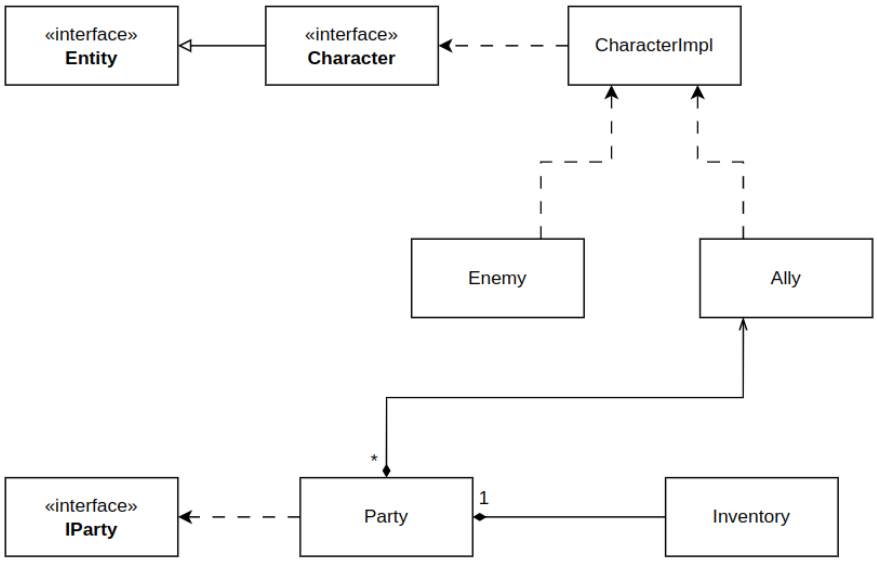

Figura 2.12: Rappresentazione UML delle classi relative ai characters e party

Lettura risorse e possibilità di modificare oggetti

Problema: gestire delle risorse presenti nel classpath da utilizzare come default e permettere al utente di poter aggiungere/modificare items

Soluzione: è stato implementato un ConfigManager che permette il parsing di file json, e tramite *MainResourceMAnager* vengono caricati i file dal classpath al primo avvio, e una copia dei file viene inserita nella giusta cartella nella home utente così che, volento, l'utente possa modificare questi file a piacimento e all'esecuzione successiva del gioco il manager carichi questi file modificati invece che quelli stock. I file presenti nel classpath e quelli presenti nella home utente sono accessibili anche individualmente tramite *SystemResourceManagerImpl* (per il caricamento di file da classpath) e *ResourceFilesManagerImpl* (per la lettura dei file dalla home utente). Per quanto riguarda l'organizzazione dei file nella home utente si è deciso di seguire lo standard di ogni sistema operativo:

- Windows: `"%LOCALAPPDATA%/TurnBasedTunnels/"`
- Linux: `"$XDG_CONFIG_HOME/TurnBasedTunnels/"`
- MacOS X: `"Library/Application Support/TurnBasedTunnels/"`

o, per windows e linux, le relative path di default specifiche del sistema nel caso le variabili d'ambiente non fossero trovate

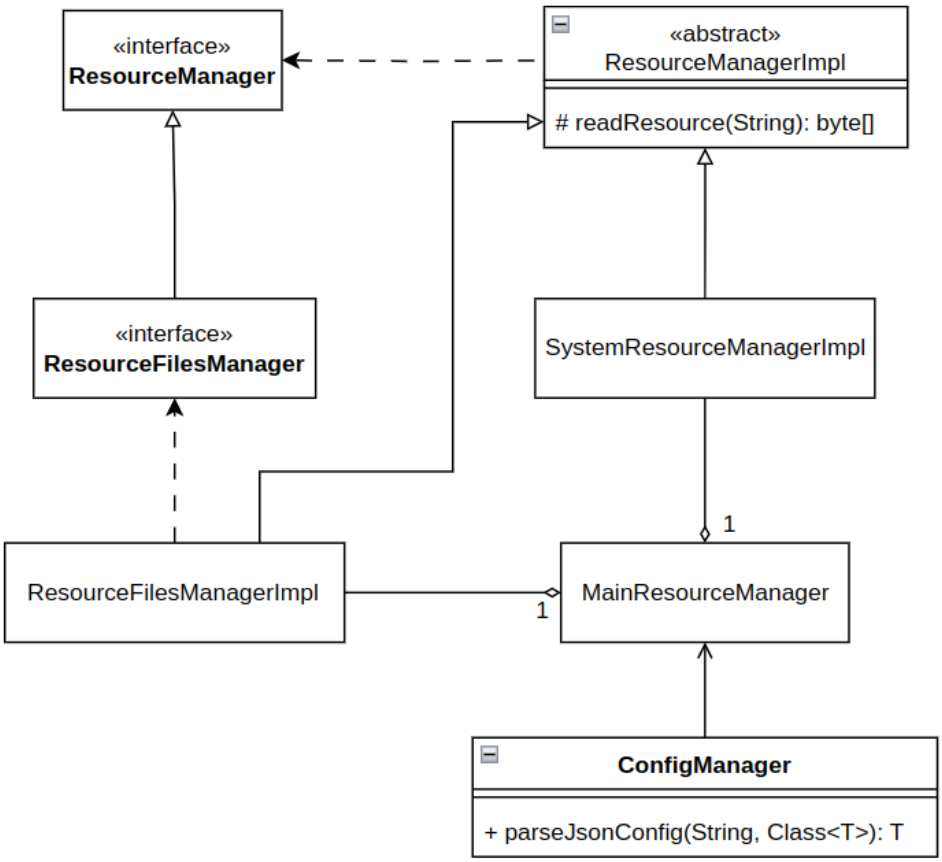

Figura 2.13: Rappresentazione UML delle classi relative al caricamento delle risorse

2.2.4 Tamburini Federico

Problema: avere un modo per gestire il combattimento con dei nemici una volta interagito con essi
Soluzione: L'oggetto FightModelImpl gestisce la creazione dei nemici e successivamente, una volta inizializzato il party, anche degli alleati. Esso conterrà tutte le informazioni su quella precisa istanza di combattimento. Una volta inizializzato il combattimento, i comandi ricevuti dalla view verranno reinderizzati ad un FightControllerImpl, che contiene al suo interno un

oggetto di tipo FightState, a cui verranno reinderizzati i comandi che verranno infine eseguiti in un FightModelImpl, modificandone il suo contenuto o restituendo suoi dati per essere mostrati in output in view.

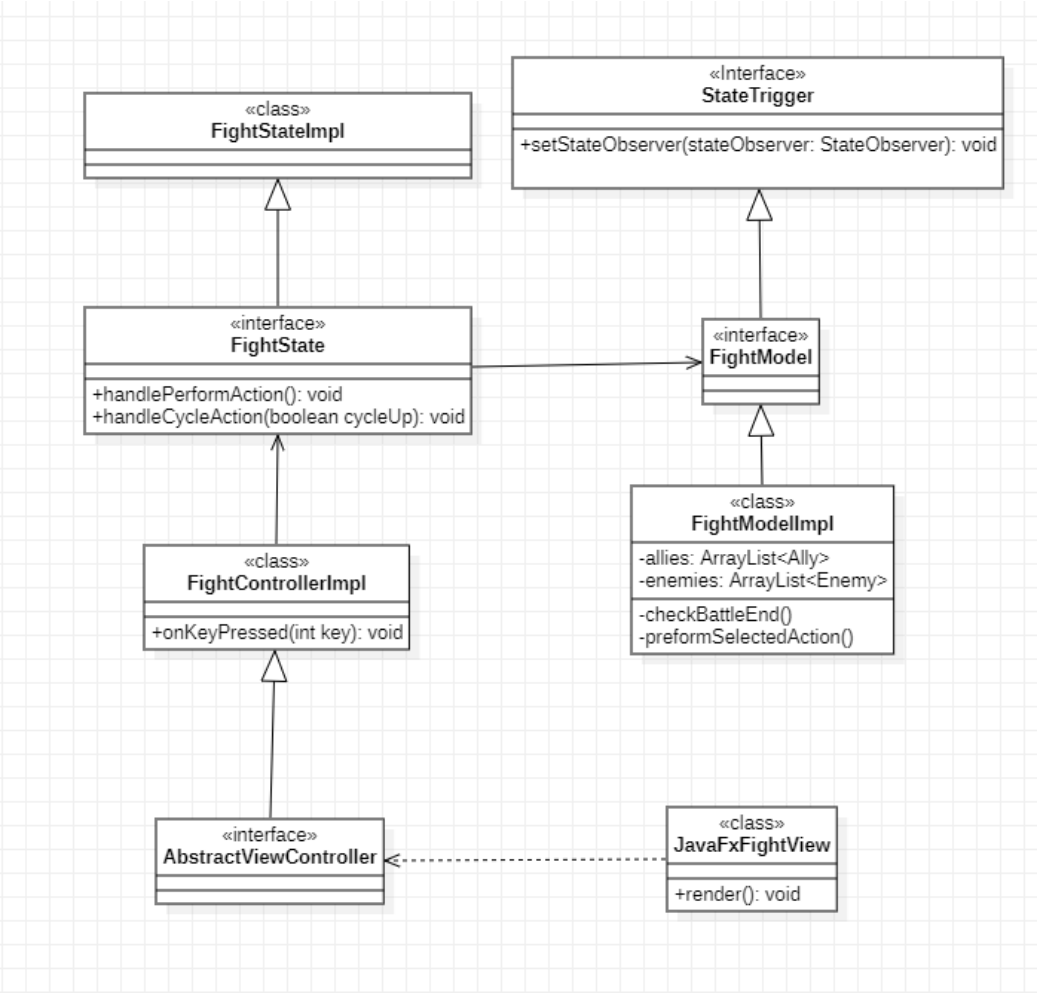

Figura 2.14: Rappresentazione UML delle classi relative alle entità

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per quanto riguarda il testing abbiamo deciso di testare le seguenti funzionalità:

- Il corretto funzionamento del model del Fight. (*FightModelImplTest*)
- Il test delle funzioni basilari del model del Menu.
- Il test delle varie factory di oggetti e skill.
- Il test delle funzioni basilari del model dello Shop.

3.2 Metodologia di lavoro

3.2.1 Bekic Dario

Mi sono occupato in autonomia di:

- Definizione e implementazione dell'architettura generale. Creazione del GameLoop, creazione del sistema a stati finiti (*GameState*), creazione del sistema di transizione degli stati (*TransitionManager*), creazione del sistema di gestione multi-framework (*ViewControllerManager*).
- Creazione del mondo di gioco (*World*) e degli ambienti (*Room*), e implementazione della gestione del movimento in questi nella classe del Party.
- Creazione del sistema di interazione (*Interactable*/ *InteractionComponent*/ *InteractionTrigger*) con il mondo e implementazione dello stesso per l'oggetto che rappresenta il giocatore (*Party*).

- Creazione della *GameView* per lo stato di *Explore* in JavaFX.

3.2.2 Bresciani Enea

Mi sono occupato in autonomia di:

- Definizione e implementazione dell'architettura MVC dei Menu, con View in JavaFx.
- Definizione e implementazione degli npc (questi non hanno una view autonoma).
- Definizione e implementazione Della parte di Control e View (JavaFX) dell'Inventory View

3.2.3 Debertolis Lorenzo

Mi sono occupato in autonomia di:

- Definizione e implementazione entità (items, character, parte del party)
- Definizione e implementazione *Resource Manager*
- Definizione e implementazione dello shop

3.2.4 Tamburini Federico

Mi sono occupato in autonomia di:

- la parte di combattimento dell'applicazione. Classi e interfacce realizzate: JavaFxFightView, FightModel, FightModelImpl, FightModelImplTest, FightControllerImpl, FightState, FightStateImpl.
- la gestione del looting dei nemici a fine battaglia.

3.2.5 DVCS e gestione del lavoro.

All'inizio dello sviluppo abbiamo un Fork del repository principale per ciascun membro e ognuno ha lavorato sul proprio Fork principalmente. Ogniqualvolta si implementava una feature, veniva creata una pull request sul repo principale e dopo valutazione delle modifiche di queste veniva fatto il merge. Ognuno ha potuto iniziare a lavorare alle proprie parti sin da subito, con implementazioni dummy delle interfacce che abbiamo concordato a monte.

3.3 Note di sviluppo

3.3.1 Bekic Dario

Uso di Optional:

https://github.com/Turn-Based-Tunnels/OOP22-tbt/blob/dc01cac4ec3d99aa5fd5dad43f1
src/main/java/it/tbt/model/world/impl/WorldImpl.java#LL50C8-L50C8

Uso di Stream:

https://github.com/Turn-Based-Tunnels/OOP22-tbt/blob/dc01cac4ec3d99aa5fd5dad43f1
src/main/java/it/tbt/view/javaFx/JavaFxExploreView.java#L149

Uso di Lambda:

https://github.com/Turn-Based-Tunnels/OOP22-tbt/blob/e276d8765d1918a1613d498cdb2
src/main/java/it/tbt/model/world/interaction/PartyInteractionComponent.
java#LL35C1-L35C1

Uso di Reflection:

https://github.com/Turn-Based-Tunnels/OOP22-tbt/blob/dc01cac4ec3d99aa5fd5dad43f1
src/main/java/it/tbt/controller/viewcontrollermanager/impl/GameViewManagerImpl.
java#L121

Uso di JavaFX:

https://github.com/Turn-Based-Tunnels/OOP22-tbt/blob/dc01cac4ec3d99aa5fd5dad43f1
src/main/java/it/tbt/view/javaFx/JavaFxExploreView.java#L33

3.3.2 Bresciani Enea

Uso di lambda:

https://github.com/Turn-Based-Tunnels/OOP22-tbt/blob/4e6b4ea835c76898afc0251db46
src/main/java/it/tbt/model/menu/impl/MenuNewGameButton.java#L32

Uso di javaFx:

https://github.com/Turn-Based-Tunnels/OOP22-tbt/blob/59dacab6688c03447dc0662d86b
src/main/java/it/tbt/view/javafx/JavaFxMenuView.java

3.3.3 Debertolis Lorenzo

Uso di stream:
https://github.com/Turn-Based-Tunnels/OOP22-tbt/tree/master/src/main/java/it/tbt/controller/resources
Uso di javaFx
https://github.com/Turn-Based-Tunnels/OOP22-tbt/blob/master/src/main/java/it/tbt/view/javafx/JavaFxShopView.java
Uso di factory singleton:
https://github.com/Turn-Based-Tunnels/OOP22-tbt/tree/master/src/main/java/it/tbt/model/entities/items/factories

3.3.4 Tamburini Federico

Uso di JavaFx

https://github.com/Turn-Based-Tunnels/OOP22-tbt/blob/85c37ddfc814090ab76ca5ed018
src/main/java/it/tbt/view/javafx/JavaFxFightView.java

Capitolo 4

Commenti finali

4.0.1 Bekic Dario

In generale, ritengo di aver svolto discretamente il mio compito, anche se credo che ci sia sempre margine di miglioramento. Penso che alcune parti di model siano state progettate senza sufficiente attenzione. Penso che ci sia stata poca coesione tra i membri, relativo poco interesse nel progetto e questo mi ha portato a dover usare più tempo del previsto, portando comunque un risultato non ottimale.

4.0.2 Bresciani Enea

I punti di forza del mio prodotto ritengo siano un'ottima modularità del model da me implementato. La debolezza più grande è una adesione ai pattern identificati in modo non ferreo. Onestamente non credo il progetto verrà portato avanti in futuro, e non sono certo tutte le parti siano così flessibili a futuri aggiornamenti.

4.0.3 Debertolis Lorenzo

Le classi e l'architettura che ho sviluppato sono particolarmente espandibili, ma sfruttano parecchio il concetto di ereditarietà, sarebbe interessante in futuro provare a reimplementare le varie entità cercando di evitarla puntando al usare quasi unicamente la composizione. Con del lavoro aggiuntivo si potrebbe poi sfruttare il *ConfigManager* e le relative classi dedicate alla lettura di file per gestire in modo centralizzato anche le risorse utilizzate nelle altre parti del gioco. Sicuramente c'è del margine di miglioramento nell'architettura sia delle parti che ho implementato personalmente, sia nella struttura stessa del gioco, e pure la gestione del repository è decisamente migliorabile

4.0.4 Tamburini Federico

Riconosco di non esser riuscito a stare 100% al passo con gli altri membri del gruppo, ma penso comunque di essermi impegnato, di aver fatto la mia parte nel progetto e di non aver creato problemi a nessuno degli altri membri.

Appendice A

Guida utente

Menu e Inventario
Nei Menu e nell'inventario la navigazione avviene tramite WASD/frecce direzionali, mentre si utilizza SPACE/ENTER per interagire con i vari controlli nei menu.
I controlli sono esclusivamente via tastiera, effettuare input con il mouse non sortirà alcun effetto.
La schermata dell'inventario è divisa in 3 colonne. La prima colonna mostra gli oggetti presenti nell'inventario. La seconda mostra gli alleati che hai raccolto. La terza mostra i dettagli del alleato selezionato. Nella schermata dell'inventario è possibile fare cose utili alla gestione del party, come ad esempio utilizzare o equipaggiare oggetti e selezionare il party attivo (primi 4 alleati in alto con bordo rosso). Selezionando 2 alleati si scambiano di posto

Shop
La navigazione è la stessa di Menu e Inventario, cambia solo l'interazione con SPACE/ENTER: l'oggetto selezionato (in giallo) verrà venduto/acquistato, in base al lato della schermata in cui si trova (sinistra: giocatore, destra: negozio)

Combattimento
Il combattimento si svolge a turni.
Con le freccette destra e sinistra è possibile cambiare il target, con le freccette in alto e in basso si seleziona l'azione che si vuole performare, con invio si esegue l'azione selezionata.
se tutti gli hp degli alleati in combattimento arrivano a 0, significa che il giocatore ha perso.
se tutti gli hp dei nemici arrivano a 0, allora il giocatore potrà continuare a giocare e la battaglia finisce.

Esplorazione
Per muoversi nel mondo si usano i tasti WASD.
Con W ci si sposta in su.
Con A ci si sposta a sinistra.
Con D ci si sposta a destra.
Con S ci si sposta in basso.
Per aprire l'inventario si clicchi il tasto I.
Per interagire con l'ambiente si deve essere abbastanza vicini all'oggetto con cui si vuole interagire.
Quando questo è vero, se si preme E ci si può interagire. Cosa le interazioni facciano, è descritto qui sotto.

Interazioni possibili

- **Ally NPC:**

  Questo NPC è un alleato, interagendo con lui, si aggiungerà al tuo party, e di conseguenza sparirà dalla mappa.

- **Item NPC:**

  Questo NPC è un benefattore, interagendo con lui ti regalerà un item, mettendolo direttamente nel tuo inventario, e in seguito sparirà dalla mappa.

- **Shop NPC:**

  Questo NPC è un mercante. Interagendo con lui è possibile acquistare e vendere oggetti. Non scopare a seguito di una interazione.

- **Fight NPC:**

  Questo NPC è un nemico. Interagendo con lui si entrerà in combattimento, a seguito dell'interazione sparisce dalla mappa.

- **Heal NPC:**

  Questo NPC è un guaritore. Interagendo con lui curerà il tuo party dalle ferite, e poi sparirà dalla room.

- **Scale:**

  Mettono in comunicazione le varie stanze del dungeon. Ricordati che possono essere monodirezionali. Interagendo con queste si può cambiare stanza del dungeon.

- **Uscita**

  Questa è la tua via di uscita dal dungeon, interagiscici per terminare il gioco.

Crediti a terzi

Per le sprite/immagini si ringrazia `https://0x72.itch.io/dungeontileset-ii`