ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Junkrisers: Rusco's Adventure

Relazione di progetto

Bandiera Luca Bevilacqua Anny Marcaccio Leonardo
Pesaresi Jacopo

giugno 2023

Sommario

Lo scopo di questa relazione è illustrare e spiegare le diverse fasi attraversate durante la progettazione dell'elaborato in oggetto, fungendo come documentazione del programma e fornendo sufficienti dettagli tecnici affinché il lettore sia in grado di comprendere la struttura del software realizzato.

Il progetto in esame è stato realizzato nell'ambito del corso di Programmazione ad Oggetti, per l'anno accademico 2022-2023. Esso consiste nello sviluppo di un videogioco 2D ispirato ai generi *dungeon crawler*, *rogue-like* e *old school rpg*, in cui il giocatore dovrà esplorare diversi livelli di difficoltà crescente.

La relazione andrà quindi definendo le metodologie usate nella modellazione, nello sviluppo e nell'implementazione del software, esponendo anche requisiti iniziali, scelte di design e dettagli tecnici inerenti all'implementazione.

Capitolo 1

Analisi

Il software da realizzare consiste in un videogioco a grafica bidimensionale, con meccaniche ed elementi tratti dai generi *dungeon crawler*, *rogue-like* e *old school rpg*. Scopo del giocatore è quello di esplorare le varie stanze nella mappa, superando livelli e sconfiggendo nemici, fino ad arrivare al 15° livello e vincere il gioco.

Con *dungeon crawler* si intendono tutti quei giochi basati su un sistema di livelli, generati in modalità procedurale, via via più difficili, che premiano l'esplorazione e spronano il giocatore ad andare sempre più avanti. Per vincere il gioco sarà infatti necessario superare tutti i 15 piani di Pattumopoli, città immaginaria in cui si svolge la trama del gioco.

Il termine *rogue-like* è usato per indicare i giochi ispirati al famosissimo videogame degli anni '80 *Rogue*[1]. In simili avventure il giocatore, interpretando i panni di uno o più avventurieri, si muove all'interno di un mondo generato in maniera randomica e affronta situazioni sempre diverse. É infatti richiesto che i livelli della città di Pattumopoli e i suoi abitanti siano sempre differenti, proponendo stili di gioco diversi ad ogni partita.

Infine, tramite *old school rpg* si fa riferimento a giochi di ruolo ispirati alle prime versioni del celebre gioco da tavolo *Dungeons & Dragons*[2], da cui prendono spesso meccaniche e concetti base, ripresentandoli in chiavi alternative. *Junkrisers: Rusco's Adventure* farà infatti uso di meccaniche tipiche di tale genere, come l'utilizzo di un sistema basato sulle statistiche del protagonista.

[1] `https://store.steampowered.com/app/1443430/Rogue/`
[2] `https://it.wikipedia.org/wiki/Dungeons_%26_Dragons`

1.1 Requisiti

Requisiti funzionali

- Il gioco dovrà incominciare mostrando una schermata d'inizio. Con "schermata d'inizio" si intende un'interfaccia comprensiva di menù che offra al giocatore la possibilità di iniziare una nuova partita o chiudere la sessione corrente.
- Il programma dovrà essere in grado di generare automaticamente i livelli di gioco.
- Ogni livello dovrà presentare al giocatore una serie di eventi che influiscano sulla sua progressione verso il livello successivo. Tali eventi dovranno comprendere affrontare nemici, scatenare trappole e raccogliere oggetti.
- Durante la partita dovranno essere disponibili dei sottomenù d'interazione che permettano al giocatore di visualizzare ed eventualmente modificare:
  - L'inventario del giocatore contenente tutti gli oggetti posseduti.
  - La salute del personaggio giocante e le sue statistiche.
- Durante una partita, il giocatore dovrà essere in grado di comandare il personaggio giocante prescelto e muoversi attraverso le diverse stanze della mappa.
- Durante un combattimento, il giocatore dovrà disporre di diverse possibilità di attacco, contemplando un attacco base, una serie di mosse speciali e l'uso di oggetti acquisiti durante il gioco.
- In caso di morte del personaggio giocante si dovranno verificare i seguenti eventi:
  - Comparsa di una schermata di game over
  - Ritorno al menù iniziale del gioco

Requisiti non funzionali

- Il programma dovrà essere portabile sui sistemi operativi più diffusi, quali Windows, Linux e MacOS.
- La finestra del programma dovrà poter essere ridimensionabile senza generare una diminuizione della risoluzione e della qualità d'immagine del gioco.

1.2 Analisi e modello del dominio

Mappa di gioco

La mappa di gioco rappresenterà strade, vie e cunicoli di Pattumopoli, città di fantasia in cui è ambientata la storia. A seguito di anni d'accumulo, Pattumopoli è diventata una vera e propria montagna d'immondizia, formando quello che in gergo è definito *dungeon*[3]: un'area racchiusa, solitamente popolata da creature ostili (nemici), nella quale è possibile trovare tesori ed oggetti rari. Il dungeon sarà strutturato su più piani, 15 in tutto, che rappresenteranno ognuno un livello del gioco. Ogni piano si estenderà orizzontalmente attraverso la montagna d'immondizia e sarà composto da più stanze interconnesse tra loro.

Le stanze, tutte di forma quadrata o rettangolare, saranno delimitate da muri e collegate tra di loro grazie a delle porte. Con *porta* si vuole indicare, più in generale, un punto di accesso che il giocatore può utilizzare per accedere a una nuova stanza. Tali porte potranno variare nell'aspetto e nel numero, andando da un minimo di una e fino ad un massimo di quattro per ogni stanza. Nelle stanze potranno inoltre essere presenti diversi oggetti, mostri e trappole con cui il giocatore potrà interagire. Per consentire il passaggio da un piano all'altro il giocatore potrà fare uso delle rampe di scale presenti sulla mappa. Una volta salite le scale il giocatore non potrà più tornare al livello precedente.

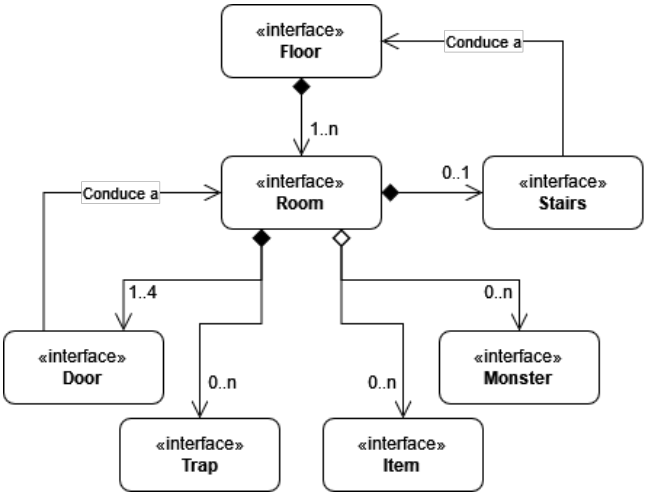

Figura 1.1: Prima modellazione del concetto di mappa

Meccaninche di gioco

Tutta la logica del gioco è basata su un meccanismo "turn-based", ciò significa che il gioco aspetterà un input del giocatore e solamente dopo averlo ricevuto il

[3] `https://www.urbandictionary.com/define.php?term=Dungeon`

turno terminerà, portando all'aggiornamento dell'ambiente circostante.
Si supponga quindi che il protagonista sia presente in una stanza con dei mostri e un forziere. Ogni volta che inizierà un turno, il giocatore dovrà decidere un'azione da far compiere al protagonista.
Tale azione potrà essere un movimento, un attacco o un'interazione.
Nel primo caso, il protagonista si sposta in una possibile posizione ortogonale ammissibile (il giocatore non può ovviamente spostarsi su un muro, o andare nella stessa posizione di un mostro).
Nel secondo caso, il giocatore sferra un attacco contro un nemico. Tale mossa può essere un attacco di base o una mossa speciale, eseguibile solo se il protagonista ha abbastanza "Energia" per farlo.
Con "Energia" si intende una barra aggiuntiva che rappresenta la capacità del giocatore di compiere azioni speciali.
Nel terzo invece, con "interazione" si intende una qualsiasi azione che comprenda il raccoglimento di un qualche oggetto da terra (e conseguente aggiunta di tale oggetto nell'inventario del protagonista), l'apertura di un forziere (che fornirà direttamente uno o più oggetti al giocatore) o attraversamento di una porta (che permette lo spostamento di una nuova stanza, il che implica che la vecchia stanza si fermi e che rinizi un nuovo turno nella nuova stanza).
Quando quindi un giocatore effettua una qualsiasi di queste azioni e viene portata a termine il suo turno si conclude.
Inizia quindi la fase di risposta. Seguendo il caso sopracitato, indipendentmente dalla mossa scelta dal giocatore, anche i mostri potranno scegliere se spostarsi o effettuare attacchi. Ovviamente questa fase viene eseguita se nella stanza sono rimasti dei mostri, altrimenti è da considerarsi essenzialmente nulla.
Il giocatore potrà fare anche uso di oggetti consumabili per ottenere vantaggi per migliorare la sua situazione nel corso della partita.
Sono implementati anche una serie di equipaggiamenti con determinate statistiche che definiscono lo stile di combattimento che il giocatore dovra attuare per poterli utilizzare al meglio. Ad esempio se possiedo una spada predidligerò uno stile più offensivo e ravvicinato di combattimeto, mentre se possiedo un arco preferirò attaccare i nemici dalla distanza evitando il più possibile di farli avvicinare
Questi oggetti possono essere raccolti durante l'esporazione delle stanze, dopo aver sconfitto i nemici o in forzieri presenti all'interno delle varie stanze.

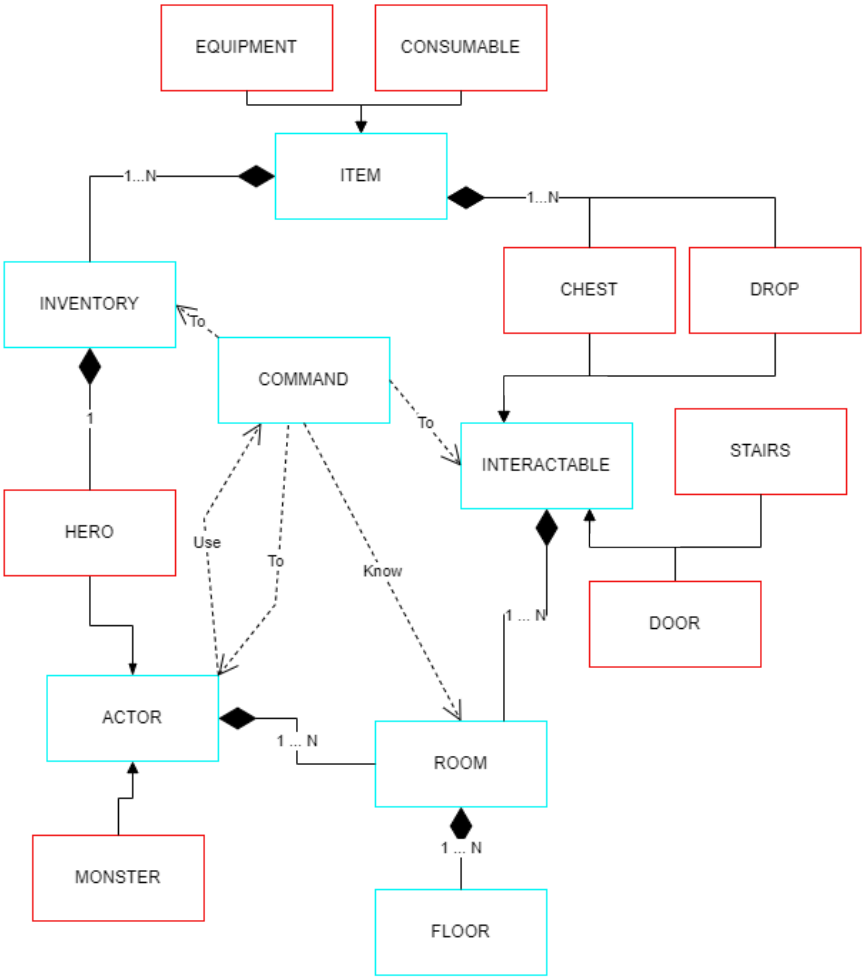

Figura 1.2: Modello ad alto livello del dominio (in azzurro i concetti più astratti, in rosso possibili specializzazioni)

Capitolo 2

Design

2.1 Architettura

Per la realizzazione di Junkrisers: Rusco's Adventure, si è optato per realizzare un'architettura basata sul pattern MVC. Tale scelta è stata fatta sulla base del desiderio di rendere il gioco modulare, permettendo così una maggiore flessibilità a fronte di modifiche, apertura alla possibilità di espansione e una decisa separazione dei concetti del gioco, ma sopratutto per tenere la logica del gioco medesima indipendentemente da quale componente view si decide poi di implementare. Il pattern è stato realizzato tramite l'implementazione di tre interfacce base che definiscono e rappresentano le componenti dell'architettura.

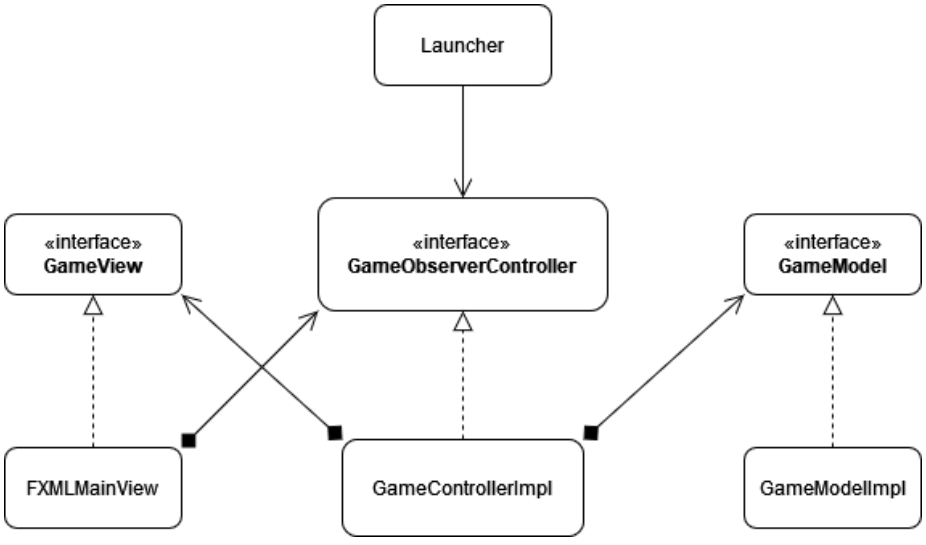

Figura 2.1: Rappresentazione UML del pattern MVC applicato al progetto

Il programma viene avviato per mezzo di una speciale classe Launcher che definisce quindi l'entry point dell'applicativo. Il metodo main, contenuto in Launcher, si occupa banalmente di istanziare il controller del gioco, richiamando successivamente le funzioni init e start, responsabili rispettivamente dell'inizializzazione tramite il settaggio della view e del suo avviamento.

View

La view è stata sviluppata come interfaccia grafica, in modo da rendere il gioco più accattivante e user-friendly ed è composta da due interfacce principali, più tre di supporto per gestire i menù di gioco ed il rendering delle immagini a schermo.

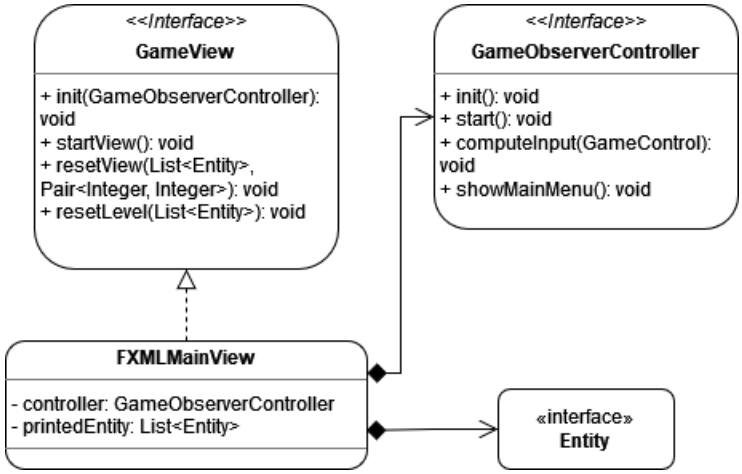

Figura 2.2: Rappresentazione UML della componente di view

Una delle classi principali, nonché quella che permette il dialogo con il controller del gioco, è FXMLMainView. MainMenuController e GameOverController si occupano, come da requisiti, di mostrare rispettivamente la schermata d'inizio gioco e quella di Game Over, che riporta al menù iniziale, mentre GameViewController si occupa di visualizzare la schermata principale di gioco. GameViewController costituisce la seconda classe principale della view in quanto responsabile del rendering di mappa, inventario e informazioni del giocatore. Essa si compone di oggetti Drawable, interfaccia che assieme alla rispettiva implementazione, rappresenta gli oggetti da renderizzare a schermo.

Controller

Il controller è stato implementato completamente nella classe GameControllerImpl, mantenendo comunque riferimenti alle classi di model e view. Il collegamento alla view, in particolare, è stato realizzato tramite una dependency injection nel metodo init di quest'ultima.

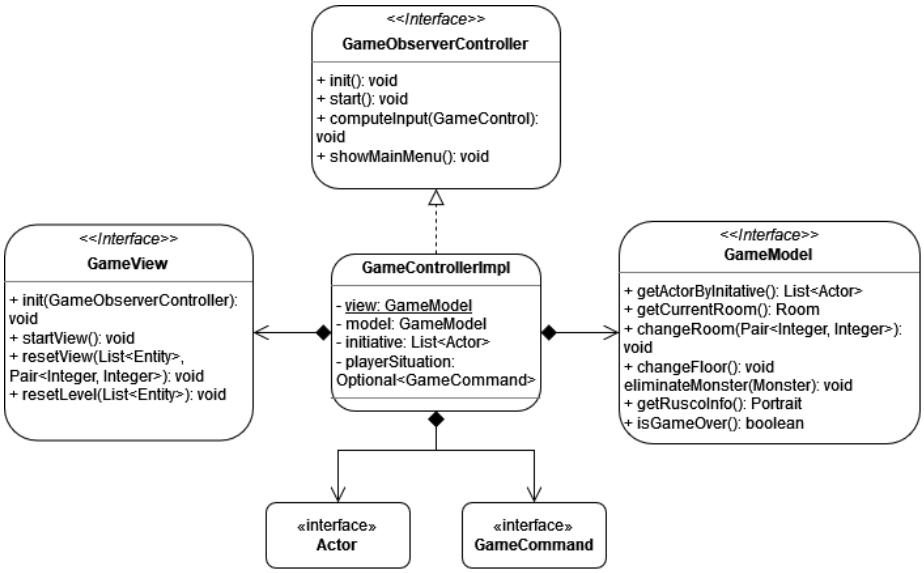

Figura 2.3: Rappresentazione UML della componente del controller

Model

Il model è costituito dall'insieme delle rimanenti classi, contenute nei package "model" e "utils". Si è deciso infatti di mettere in "utils" tutte le classi comuni a diversi package a così alto livello (è il caso delle "exception", comuni sia al model che al controller) o di eventuali classi che servono a supportare gli algoritmi eseguiti nelle classi del model, o a modellare meglio dei concetti (è il caso della classe fornita dai docenti "Pair" e di quella di algoritmi ad essa legata "Pairs", di nostra produzione) Di tutte queste classi si desidera parlare qui di quella che implementa GameModel, ovvero GameModelmpl, il cui principale compito è implementare le principali funzioni del model, ovvero:

- aggiornare il controller della "lista di iniziativa", così che sa di chi è attualmente il turno e quindi può correttamente reindirizzare a tale frangente di model l'input
- controllare se il gioco è finito (predisponendo dei metodi appositi e specifici, ovvero isGameOver e isGameWin)
- aggiornare il proprio stato interno, cambiando il riferimento al piano corrente o alla stanza corrente, e ritornare al controller la sola informazione di quale sia la stanza corrente

Si è posta particolare attenzione a disaccopiare la componente View da quella di Controller-Model, rendendola anzi quasi dipendente a 3 interfacce che necessariamente conoscono (sia direttamene che meno) tutte e tre le componenti. In particolare:

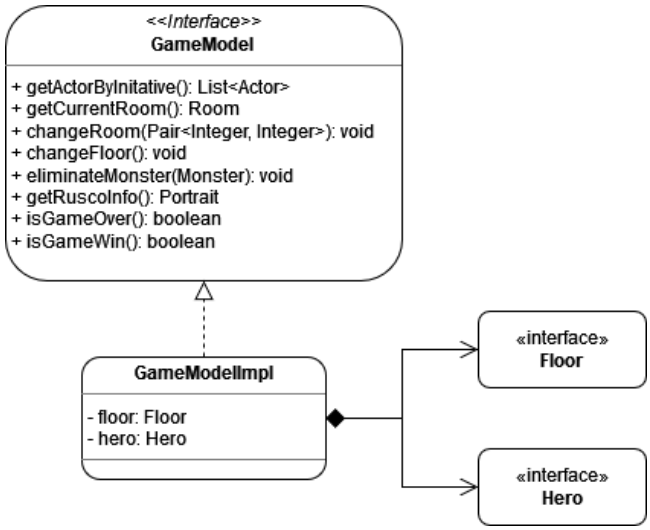

Figura 2.4: Rappresentazione UML della componente del model

- Entity: interfaccia contenente minimali informazioni utili alla view per sapere chi è dove
- InfoPayload: interfaccia contenente informazioni utili per aiutare il giocatore a comprendere particolari dinamiche di gioco (sopratutto eventuali errori)
- Portrait: interfaccia in cui riversare tutte le informazioni utili a decorare la view di informazioni non strettamente necessarie al meccanismo di base del gioco.

2.2 Design dettagliato

Mappa di gioco
A cura di Anny Bevilacqua

Come precedentemente accennato, il gioco sarà suddiviso in più livelli, ognuno dei quali rappresentante uno dei piani della città e composto da diverse stanze esplorabili. Diventa quindi necessario riuscire a realizzare una struttura di classi adatta per gestire in modo modulare e possibilmente procedurale, la creazione dei livelli, al fine di creare un'esperienza di gioco stimolante ed immersiva per il giocatore. In questa sezione verranno specificate alcune delle metodologie applicate per gestire la generazione della mappa e quali pattern di programmazione sono stati utilizzati per farlo.

Piani diversi per livelli diversi

Data la suddivisione del gioco in diversi livelli, si è deciso di approcciarsi alla generazione della mappa con un approccio top-down, partendo dalla creazione di un'interfaccia in grado di rappresentare i piani della città. Fin dai primi stadi di progettazione è tuttavia sorto un dubbio su come poter gestire piani di tipologie diverse. Essi infatti si potrebbero distinguere per difficoltà o meccaniche di gioco differenti: potremmo per esempio avere un livello formato da ampie stanze densamente popolate di mostri, o uno con una struttura labirintica, in cui il giocatore non debba prestare particolare attenzione alla presenza di mostri, ma bensì di trappole nascoste. Diventa quindi necessario riuscire a gestire i piani in modo intercambiabile. La gestione dovrebbe risultare indipendente dalle caratteristiche interne di ogni livello e potersi integrare in modo trasparente nella gestione della partita da parte del modello.

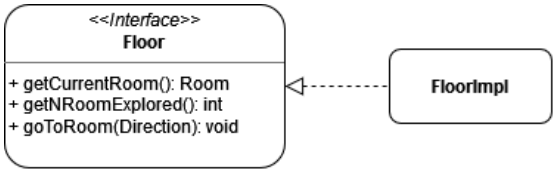

Figura 2.5: Rappresentazione UML del pattern Strategy per i piani della mappa

Problema Si desidera avere piani di tipologie e/o difficoltà diverse. I piani dovranno essere intercambiabili tra loro, permettendo di effettuare scambi anche a runtime, pur mantenendo una gestione trasparente al client.

Soluzione Per risolvere questo problema si è deciso di fare affidamento sul pattern *Strategy*. Strategy permette infatti di definire un'interfaccia comune che incapsula i diversi tipi di piano come strategie differenti, separando la logica interna e permettendo di mantenere il codice in una struttura più organizzata e semplice da comprendere, come in accordo ai principi SOLID. L'incapsulamento del comportamento dei piani, inoltre, permette l'uso di astrazioni per modellare i loro elementi interni, favorendo un approccio modulare alla loro generazione e aggiungendo flessibilità al programma. In questo modo, il client che dovrà interagire con i piani della mappa potrà fare affidamento agli stessi metodi, comuni a tutte le implementazioni di piano, senza avere la necessità di svilupparsi in funzioni specifiche per la gestione delle varie tipologie di piano e permettendo così il riuso del codice esistente e l'intercambiabilità dei piani a runtime.

Stanze intercambiabili

Come già accennato in precedenza i piani saranno costituiti da più stanze, attraverso cui il giocatore potrà muoversi ed esplorare il dungeon. Proprio come per

i precedenti e al fine di creare un'esperienza di gioco più stimolante ed immersiva, anche in questo caso potremmo avere diverse tipologie di stanze, distinte per forma, meccaniche di gioco o tipologie di mostri presenti. I piani potranno quindi risultare composti da una sola o più tipologie di stanze. Torna così il concetto di intercambiabilità, stavolta applicato alle stanze, che dovranno essere agevolmente gestite dal comportamento interno del piano, così come da altri metodi del modello di gioco.

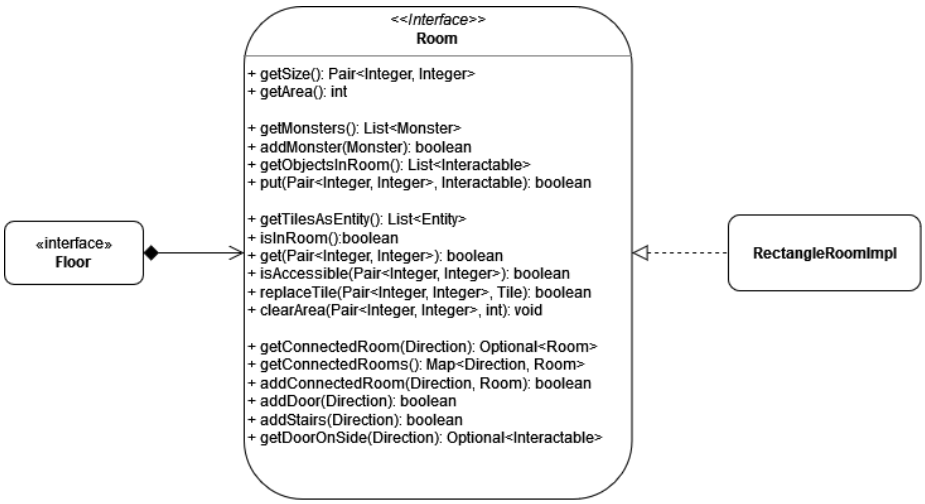

Figura 2.6: Rappresentazione UML del pattern Strategy per le stanze della mappa

Problema Si desidera avere stanze con caratteristiche diverse e che siano intercambiabili tra loro. Le stanze dovranno mantenere le stesse funzionalità senza intaccare l'esecuzione del gioco.

Soluzione Decidiamo nuovamente di applicare il pattern *Strategy*. La presenza di un'interfaccia con metodi comuni alle varie classi permette anche qui di incapsulare il comportamento delle diverse tipologie di stanze, mantenendo il codice organizzato e semplificando la manutenzione dello stesso. Sarà quindi possibile astrarre il concetto di stanza, definendo più implementazioni tra loro intercambiabili e donando così flessibilità al programma. Ciò permette inoltre un cambio di stanza agevole, che potrà avvenire basandosi su specifiche dinamiche del piano o sulle interazioni eseguite dal giocatore.

Generazione delle stanze

Durante la fase di analisi dei requisiti è stato evidenziato il desiderio di avere livelli generati automaticamente, in modo da non avere mai partite ripetitive con configurazioni della mappa già incontrate.

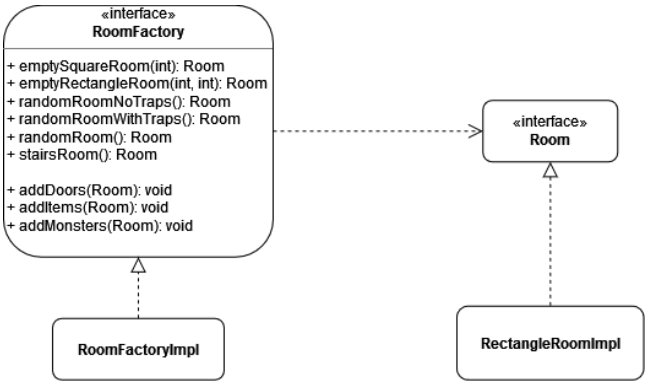

Figura 2.7: Rappresentazione UML del pattern Strategy applicato alla Factory di stanze

Problema Si desidera creare numerose stanze con caratteristiche diverse, nascondendo la logica di generazione.

Soluzione Il pattern *Factory* permette di centralizzare la creazione delle stanze, fornendo metodi per creare varie tipologie di stanze e permettendo di delegare la responsabilità del compito a una classe apposita. Inoltre, l'uso di questo pattern permette un maggior riutilizzo di codice e una maggiore flessibilità a fronte dell'aggiunta di nuove tipologie di stanze. Va notato che è stato anche qui riproposto l'utilizzo del pattern *Strategy*, in vista di una eventuale espansione della generazione della mappa.

Tiles alla base delle stanze

Al fine di rendere la creazione delle stanze della mappa il più modulare possibile, si è deciso di rappresentare lo spazio presente come suddiviso in Tile: unità individuali di forma quadrata contenenti terreno omogeneo. L'uso di Tile di tipo diverso diventa quindi essenziale per creare un'esperienza più immersiva e definire le caratteristiche della stanza.

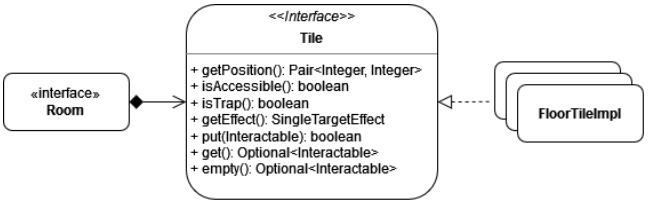

Figura 2.8: Rappresentazione UML del pattern Strategy applicato alla Factory di stanze

Problema Si desidera gestire diverse tipologie di Tile, ognuna con caratteristiche e meccaniche differenti. La gestione dovrà essere trasparente con eventualmente la possibilità di modificare dinamicamente il tipo di Tile durante il runtime.

Soluzione Il pattern *Strategy* permette di incapsulare il comportamento delle Tile, permettendo di selezionare la strategia più adatta all'esigenza corrente. Questo pattern si rende nuovamente utile nello sviluppo di un design modulare ed estendibile, che consenta l'estensione tramite l'aggiunta di nuove Tile e dinamiche di gioco.

Creazione ripetuta delle Tile

Durante il processo di generazione di una stanza, appare ovvia la necessità di dover creare numerose Tile, spesso di tipologie diverse, per definire quelli che saranno gli ambienti di gioco. Questa operazione, soprattutto nel caso della creazione dei muri, potrebbe richiedere ulteriori passaggi per andare a definire quelle che saranno le caratteristiche della Tile, aggiungendo diverse istruzioni che andrebbero a complicare la comprensione del codice originale.

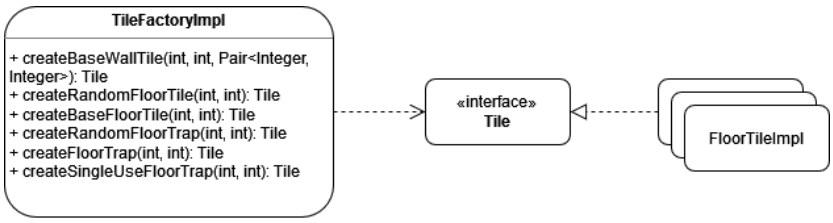

Figura 2.9: Rappresentazione UML del pattern Strategy applicato alla Factory di stanze

Problema Si desidera separare la logica di creazione delle Tile dalla generazione complessiva della stanza per evitare una over complicazione del codice.

Soluzione Il pattern *Factory* permette di incapsulare il processo di creazione e configurazione delle Tile, fornendo un accesso centralizzato e consistente durante tutti gli stadi del gioco e rendendo più semplice gestire o modificare la creazione di una specifica tipologia di Tile. Il client che farà uso della Factory potrà interagire direttamente con i metodi di suo interesse per ottenere oggetti aderenti all'interfaccia astratta delle Tile. Questa caratteristica permette l'astrazione e disaccoppiazione dei componenti, aumentando la modularità del programma e permettendo agevolmente di sostituire diversi tipi di Tile tra loro.

Jacopo Pesaresi

Una partita, tante forme di interazione: i GameCommand

Una volta che la stanza è stata creata, il giocatore deve poter iniziare a interagire con la partita.

Problemi: si è subito interessati a come poter gestire il loop che sta alla base del gioco, e quindi:

- Il gioco deve poter gestire autonomamente il turno dei mostri mentre deve aspettare l'interazione dell'utente quando è il turno degli eroi
- Il gioco non deve (o quantomento, per mantenere il gioco un minimo fluido non dovrebbe) aspettare un input di conferma per eseguire comandi "semplici", o meglio "veloci" (esempio: se si desidera spostare un eroe, non ha senso dover richiedere sempre una conferma da parte del giocatore... risulterebbe una gestione dell'input inutilmente pesante)
- Il gioco deve invece gestire altre situazioni che richiedono maggiore interazione da parte del giocatore, e che quindi necessitano sostanzialmente di un continuo ciclo di presa di input dalla view, rielaborazione dal model e aggiornamento di quanto rielaborato nella view, finchè il giocatore non è soddisfatto della sua interazione e quindi può confermare la sua esecuzione

Un primo problema è sicuramente stato: come poter gestire allora generalmente le varie forme di input?
Soluzione: al di là della prima intelaiatura (e poi successiva evoluzione) del Controller, per poter gestire ogni forma di interazione si è da sempre decisi di predisporre una serie di classi che implementassero una interfaccia comune: "GameCommand".

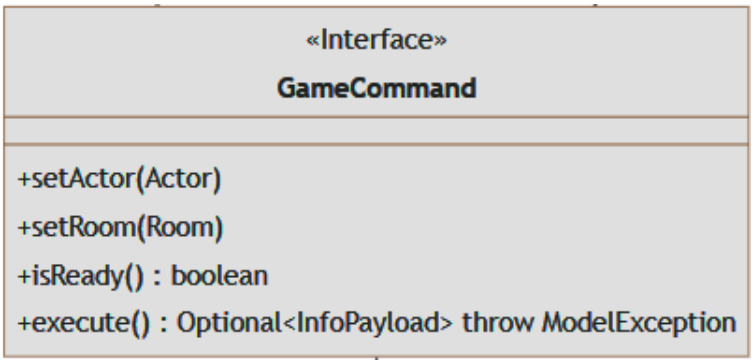

Figura 2.10: Interfaccia dei comandi

Si è considerato opportuno risolvere ogni possibile esigenza di interazione del gioco in una specifica classe (che implementa gli algoritmi necessari a gestirlo), secondo l'idea che vige dietro al "Command Pattern".

Questa interfaccia tuttavia non contiene solo il metodo "execute" (approfondito in un apposito paragrafo). I due setter servono per finire di determinare lo stato interno del comando (tipicamente tutti i comandi devono sapere quale "Actor" ha lanciato l'interazione e in quale "Room"), mentre "isReady" serve per capire, principalmente, se l'utente è soddisfatto di come ha impostato il comando, determinando praticamente una eseguibilità positiva del comando

È possibile riassumere i vantaggi di questa soluzione:

1. Per implementare qualsiasi futura nuova situazione che si vuole gestire è sufficente creare una nuova classe che implementi questa interfaccia

2. Notevole riduzione di dipendenze dirette tra le varie componenti del model *(grazie ai setter è possibile creare quasi ovunque un comando, specie perchè tipicamente queste classi vengono create all'interno della classe "Skill" di "Actor", una classe che però non conosce nel suo stato interno le informazioni "chi" e "dove", e da qui la necessità di predisporre i due setter)*

3. Facile manuntezione delle situazioni che si sono già gestite (se un giorno si vuole modificare il "drop" di oggetti lasciato da un mostro quando muore, basta mettere mano solo alla classe che gestisce l'interazione dell'attacco)

Approndimento del desing utilizzato dietro al problema In primis, bisogna subito aggiungere che in realtà GameCommand implementa altre due interfacce. Entrambe servono a gestire comandi che gestiscono interazioni "complesse" (o meglio, non "veloci", non "semplici"), ma si differienzano in quanto:

- HandableGameCommand: contiene tutti i metodi utili a gestire le interazioni "complesse" giocatore-gioco

- IAGameCommand: contiene tutti i metodi utili a gestire le interazioni "complesse" gioco-gioco

Data la vastissima necessità che questo problema richiede, non si è riusciti a implementare un vero e proprio pattern noto (o quantomeno, non uno che fosse noto al team di sviluppo)

Inizialmente si è tentato di ispirarsi al "Builder pattern", data la necessità di costruire oggetti indubbiamente complessi. Tuttavia, dopo giorni di tentivi, si è infine decretato che era difficilmente applicabile un tale pattern perchè:

- difficile prevedere una interfaccia comune a quelle che sono le "tre classi di necessità" per cui è possibile classificare i comandi (comandi veloci (o comandi "quick"), comandi complessi a cura del giocatore, comandi complessi a cura del gioco (detti anche comandi "complex"))

- avrebbe reso l'implementazione inultilmente contorta. Per poter fare un bel lavoro, si era pensato innanzi tutto di non permettere a GameCommand di implementare HandableGameCommand e IAGameCommand, e di lasciare in questa interfaccia solo "isReady" e "execute". Si reputava

infatti interessante l'idea di spostare i due setter in un apposito builder di questi comandi (totalmente normali per questo pattern). Poi provvedere per questo builder due implementazioni, che estendevano da GameCommand: una per i comandi quick e una per i comandi complex, distinguibili in base al metodo "isReady". Nel caso dei quick, "isReady" resituiva true e tutte le classi che estendevano da questa classe erano i comandi concreti quick. Nel caso dei complex, questo metodo restituiva false, e si aveva allora necessità di costruire un comando complesso. Tuttavia si paventavano problematiche più o meno rilevanti:

  - Per permettere alle classi costruite dal builder di conoscere le informazioni fondamentali "chi" e "dove", era necessario o aggiungerle nel costruttore di ogni comando complesso concreto, o quantomeno predisporre nuovamente dei setter. La seconda soluzione era spiacevole (già fatto nei builder) mentre la prima è praticamente impossibile, in quanto, quando nel codice si desidera creare un oggetto complesso, queste informazioni sono praticamente ignote *(tipicamente questi comandi vengono originati dentro alle "Skill" degli attori, oggetto di cui si aggrega ma al cui interno non si ha nessun rifermento, nè dell'attore stesso nè della stanza)*. Si era allora pensati di aggiungere una nuova interfaccia oltre alle interfacce HandlableGameCommand e IAGameCommand (quella che nel UML deprecato sottostante è chiamata "ComplexObserver"), settabile poi nei builder dei comandi complessi attraverso un metodo setter, in modo che questi potessero pescarle all'occorrenza. Ma sembrava una soluzione terribile e che rendeva più difficile manuntenere il codice
  - Inoltre, ogni comando complesso desidera certe informazioni per essere eseguito *(l'apertura dell'inventario non ha bisogno dei concetti di area degli attacchi, quello di attacco invece ovviamente sì)*, quindi si doveva prevedere un insieme di costruttori per ComplexActionBuilder e un insieme di metodi che permettevano di costruire il giusto comando
  - Se in un futuro si voleva aggiungere una terza categoria di comandi, che non fosse ne Handlable ne IA, il builder complex doveva aggiungere un'altra famiglia di metodi
  - Risultava veramente complesso all'interno di Skill (con cui i comandi devono necessariamente collaborare, seppur non di mia responsabilità ma di Leonardo Marcaccio) capire cosa mettere come parametro di ritorno al metodo getAction.

- Inoltre, non si sfruttava una delle principali peculiarità del "Builder pattern": istanziare una volta sola la classe builder che genera più oggetti, ma anzi perdere il suo riferimento una volta che il comando andava a buon fine (qui infatti si è sempre ragionato di caratterizzare ogni interazione col gioco in un comando generato sul momento).

- E infine, un builder command doveva costruire un oggetto che una volta costruito andava praticamente subito perso (addirittura non si desiderava neanche fargli costruire veramente un oggetto data la sua volatilità)

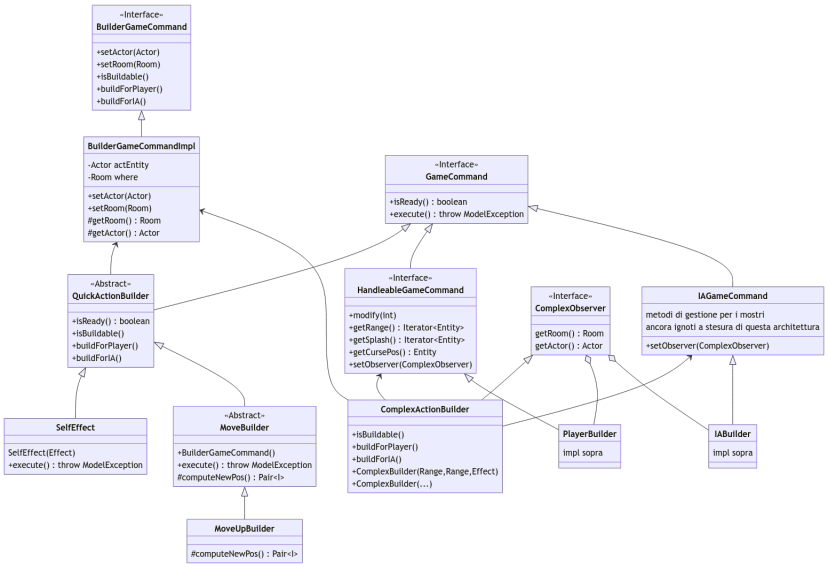

Figura 2.11: Idea di architettura dietro a quanto spiegato sopra (notare che è una architettura abbastanza antecendente alle ultime soluzioni, e onestamente anche un pò trascurata, ma si desiderava inserirla per provare a dare un minimo di riferimento alla spiegazione precedente)

E allora insomma, tanto valeva implementare classi che fossero direttamente dei comandi

Ulteriori dettagli implementativi di questa soluzione è possibile trovarlo nell'apposito paragrafo "Metodologia di lavoro". Qui sotto si limita a rappresentare l'UML del design effettivamente implementato, limitato allo stretto necessario:

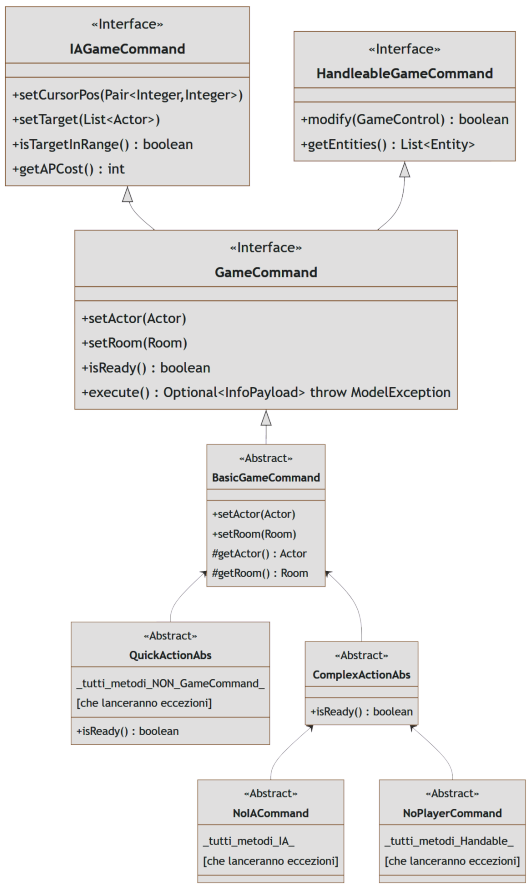

Figura 2.12: "Albero delle implementazioni": nelle interfacce i metodi comuni a tutti i comandi, nelle classi astratte vengono riportati i metodi implementati in quella classe. I comandi effettivi estendono da almeno una delle ultime tre classi astratte, le uniche "foglie" di questo albero

Concludo pertanto ammettendo che il punto debole di questa soluzione è che potrebbe risultare semplice invocare un metodo sbagliato, o meglio inappropriato per il tipo di comando nascosto sotto l'interfaccia comune GameCommand, in quanto, indipendentemente dal contesto, su un GameCommand è possibile invocare qualsiasi metodo. (insomma, non ha senso e non si deve chiamare il metodo "modify" su un comando veloce, eppure l'interfacca lo permette!)

Focus: i comandi di movimento

I comandi che hanno incapsulano l'interazione "muovi l'attore qui" sono essenzialmente 4: su, giù, destra, sinistra. Come deducibile da spiegazione precedente, sono tutti comandi di tipo "quick", ma che hanno tutti l'implementazione di un algoritmo comune:

1. Calcola la nuova posizione ortogonalmente adiacente a quella dove l'attore è in questo momento
2. Se questa posizione è effettivamente accessibile (insomma: non si sta spiaccicando l'attore contro un muro della stanza o contro un altro attore), allora consenti il comando, altrimenti... (si rimanda alla lettura del prossimo paragrafo)

È palese l'aspetto comune di tutti questi spostamenti, distinti solo per la direzione. Volendo attuare il "DRY Principle", sono le premesse perfette per poter applicare il "Template Pattern", implementando l'algoritmo caratterizzante la generale interazione "muovi" nella classe MoveCommand, mentre delegare il "muovi qui" in apposite classi, che quindi implementano il metodo astratto "computeNewPos" dichiarato in MoveCommand e usato nell'execute di questa classe

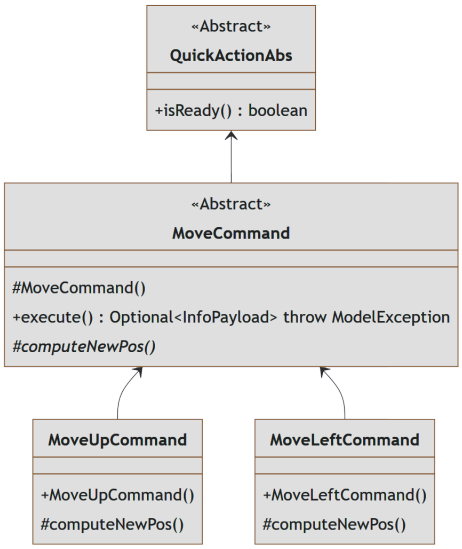

Figura 2.13: MoveCommand è la classe dei movimenti che chiama il suo metodo astratto "computeNewPos" durante execute, e che le altri classi implementano al fine di concretizzarlo

L'estrema comodità di questa soluzione è che se in un futuro si desidera muovere gli attori in modi diversi (esempio in diagonale) è sufficente creare una

nuova classe che calcola solo la nuova posizione, farla estendere da questa classe, quindi aggiungere questo nuovo comando nelle Skill di quel attore che si desidera aggiornare

Anche i comandi devono essere controllati

Si desidera aprire una piccola parentesi che spiega le modifiche relative al metodo "execute" del GameCommand, che lo distingue dall'execute tipico del pattern da cui tutta l'architettura prende nome. (ovvero un metodo che tipicamente returna void e che non lancia eccezioni) Finora si è dato abbastanza scontato che ogni comando sia normalmente eseguibile quando il giocatore lo desidera (implicitamente subito per i comandi quick, attesa di un suo input per quelli complex).
Ma può succedere che in realtà un comando non sia effettivamente eseguibile, anche se desiderato dal giocatore, per svariati motivi. Per fare un esempio diverso da quello fatto nei punti precedenti, se il giocatore punta il cursore di attacco fuori dalla portata di range del comando di attacco, allora l'attacco non può andare a buon fine, anche se il giocatore ha dato conferma dell'eseguibilità del comando! Può anche succedere che il comando vada effettivamente a buon fine, ma che la sua esecuzione comporti anche una alterazione del normale flusso di gioco. Un semplice esempio di questa situazione è quando il giocatore attraversa porte o scale per cambiare stanza o piano: il cambio del contesto deve forzare un spopolamento della lista dell'iniziativa, che poi ripopolata (seguendo il meccanismo implementato nel controller). Oppure il giocatore decide, invece di attaccare un determinato mostro, di muoversi (magari per indietreggiare), e quindi può inviare al controller l'input di annullamento per annullare il comando d'attacco

Essendo tutte situazioni particolari, si è deciso di affidare la loro gestione al Controller attraverso il meccanismo del ritorno al chiamante offerto dalle Exception. Per aumentare la chiarezza di questa classe di eccezzioni, e anzi effettivamente gestirle a livello implementativo, si è deciso di creare una classe principale chiamata "ModelException" e estendere questa classe in nuovi classi che aiutano a dedurre la natura dell'eccezzione e a garantire, eventualmente, una corretta gestione all'interno del Controller (predisponendolo di appositi metodi privati). A posteriori, per capire meglio la natura di qualche errore, si è deciso di appaiare alle semplici eccezzioni anche il concetto di "InfoPayload" prima descritto. Ottenere dall'esecuzione di un comando questo oggetto equivale a considerare l'esecuzione di quel comando come fallita

A meno di alterazioni drastiche del flusso di gioco (determinate principalmente dal cambio di stanza o di livello), una esecuzione "fallita" di un gioco corrisponde a mantenere il turno per quel attore (se il giocatore ha premuto per sbaglio "invio" per confermare un comando di attacco che non può essere eseguito, ha diritto ancora a sistemarlo o a fare un'altra interazione)

Gestire attacchi: Range

Problema: dotare i comandi di attacco di concetti aggiuntivi che non servono necessariamente ad altri comandi, ovvero i concetti di "area". Dotare insomma gli attacchi di quelle informazioni che permettono di distinguere un attacco base, da uno invece speciale. L'area, sia di range o di splash, deve essere componibile a piacere, deve aiutare il comando di attacco a definire se è effettivamente eseguibile o meno e, nel caso, a specificare chi sono i bersagli dell'attacco. Ogni tipo di area deve poi ovviamente dipendere dagli elementi presenti nella stanza (o comunque, in ogni caso, non ha senso avere un raggio di attacco che esce dalla stanza!)
Soluzione: data la necessità di svincolare il comando in sè da questi controlli, data la necessità di far dipendere la forma in base a quale area concettualizzare, e data l'eventuale necessità di definire forme più complesse, si è deciso di comporre il comando d'attacco di due oggetti che implementassero l'interfaccia "Range", e che eseguissero le operazioni prima richieste al loro interno, secondo la logica dello "Strategy pattern" e di pemettere la combinazione di varie forme usando lo "Decorator Pattern"

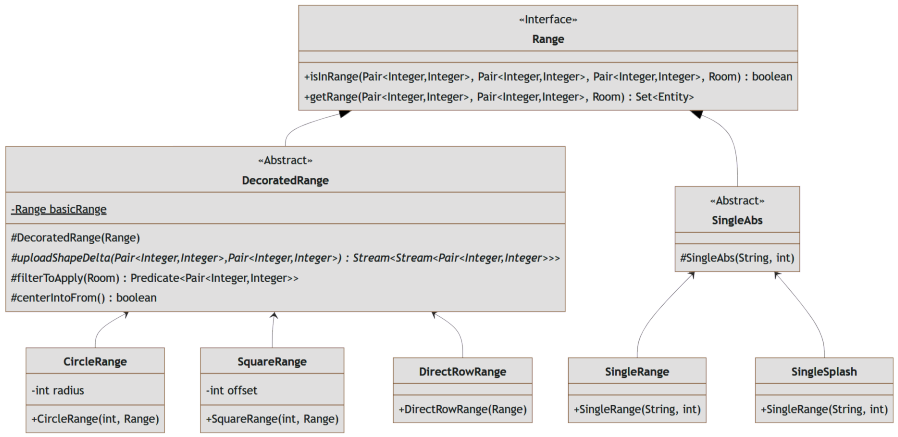

Figura 2.14: Architettura di Range (esistono altre classi che estendono DecoratedRange)

L'interfaccia definisce quali operazioni deve poter svolgere un range, e rappresentano i metodi che seguono la logica del pattern Strategy. Le classi astratte servono per definire i comportamenti che comunemente avranno le classi che li estendono, semplicemente il sottoalbero implementativo la cui radice è "SingleAbs" rappresenta l'insieme delle classi concrete del Decorator Pattern, mentre quello la cui radice è "DecoratedRange" rappresenta l'insieme delle classi "ingredienti", che decorano un range di partenza. Di conseguenza, in questo contesto, con "decorazione" si intende l'estensione di una forma di partenza (nonchè l'esecuzione ricorsiva delle operazioni-strategy offerta da interfaccia).

DecoratedRange offre il metodo astratto "uploadShapeDelta", che serve a molti tipi di Range per definire qual'è la loro effettiva forma (che sarà poi manipolata dai metodi privati citati nello schema UML), per poi essere opportunamente filtrata dai rimanenti metodi implementati da questa interfaccia. Notare quindi come il pattern Decorator si serve per certi versi anche di una sorta di "Template pattern", delegando alle classi concrete del DecoratedRange il compito di definire la propria forma.

In questo modo:

1. risulta particolarmente comodo creare sia un nuovo tipo di area sia una nuova forma, qualora vi fosse la necessità (basta estendere la giusta classe astratta)

2. è possibile cambiare e personalizzare gli algorimti per ogni area, se lo si reputa necessario

3. se in un futuro si desidera cambiare in runtime il range di un attacco, la logica dello Strategy offre tranquillamente questa possibilità, sarà sufficente cambiare quella componente prima di costruire il comando di attacco nuovo *(ahimè funzionalità non implementata nel progetto, ma pronta se in un futuro si vuole implementare "gli effetti di stato")*, o anche se si volesse arricchire il range di un attacco già esistente, attribuendogli una nuova forma, perchè l'eroe che può eseguire quel attacco possiede un particolare status

Gestire la morte di un mostro: Drop

Problema: il mostro quando muore può rilasciare a terra un drop
Soluzione: Il possibile drop che può rilasciare un mostro è stato concettualizzato in una apposita classe che implementa l'interfaccia "DropManager". I metodi servono per selezionare solo certi elementi piuttosto che altri dalla lista di partenza (si rimanda a documentazione per capire meglio i vari filtri che ogni metodo offre, nel caso non siano autoesplicativi da UML). L'unica cosa che cambia tra un DropManager e un altro è la collezione iniziale di elementi che lo costituiscono, e in particolare generarla in funzione di determinate informazioni. Si è allora deciso di creare una classe generante i "DropManager" secondo l'idea dietro al "Factory pattern" incapsulata sotto il nome di "DropFactory", essendo la creazione di questa lista comune per ogni forma di drop possibile ma differente solo per i parametri di partenza (insomma: generare un drop sul mostro appena sconfitto piuttosto che sulle dimensioni della stanza, affinchè anche il generatore della stanza potesse sfruttare un concetto simile, cambia solo la fonte di generazione, ma per il resto il meccanismo è lo medesimo)

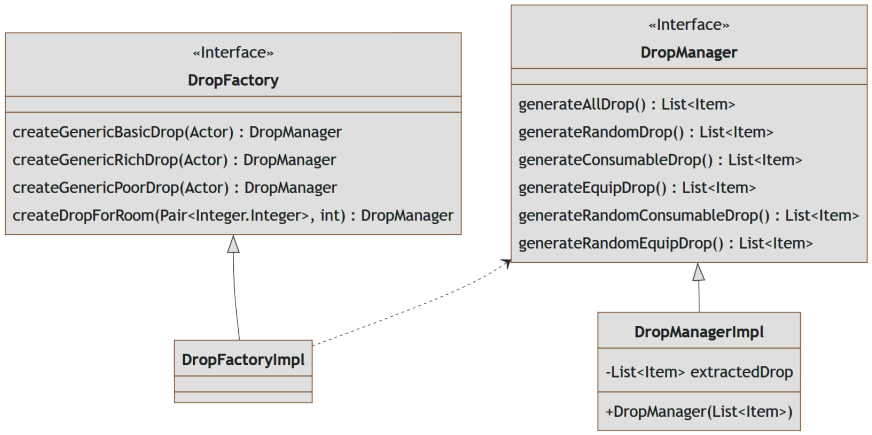

Figura 2.15: Architettura dei Drop

Luca Bandiera

View del gioco

Una volta che tutti i componenti del gioco sono pronti non rimane che stamparli. La view si occuperà infatti di stampare tutti i componenti del gioco quando vengono creati e ogni volta che succede qualcosa per cui la vecchia stampa non è più adeguata (per esempio quando l'eroe si muove, viene ucciso un mostro, si apre un forzie o si raccoglie un oggeto).
Una caratteristica importante della view è quella che deve essere responsiva, ossia si deve adattare dinamicamente a varie dimensioni dello schermo in modo

coerente, ovvero senza subire grosse modifiche.
Problema: garantire al giocatore un'interfaccia di gioco costantemente aggiornata a ciò che succede nel dominio salvato nel gioco, ma allo stesso tempo sufficientemente distaccata per poterla adattare a quanti più dispositivi diversi.
Soluzione: per sviluppare un tale sistema si è serviti del pattern MVC ampliamente descritto soprastante. Mi limito allora a sottolinare come la view riesca a tenere aggiornato il gioco attraverso la comunicazione al controller, possibile grazie al "Observer pattern", mentre riesce a tenere sempre aggiornato il giocatore offrendo, in metodi diversi, la possibilità di stampare informazioni diverse raggruppate sotto una delle tre interfaccie prima descritte.

Interactable

Gli oggetti interagibili riguardano tutti quegli oggetti nella mappa con cui l'eroe può interagire e vengono generati randomicamente durante la creazione delle stanze. Nel nostro gioco gli oggetti interagibili sono di quattro tipi e sono:

- Door: permette il passaggio dell'eroe da una stanza all'altra, situate sullo stesso piano;
- Stair: permette il passaggio dell'eroe da un piano a quello successivo;
- Chest: permette all'eore di raccogliere gli oggetti che si trovano al suo interno e di metterli nell'inventario;
- Drop: permette all'eroe di raccogliere gli oggetti posseduti da un mostro e aggiungerli nel suo inventario, una volta che il mostro è stato ucciso;

Problema: l'eroe deve poter attraversare porte e scale nel gioco per andare avanti e interagire con eventuali forzieri e drop dei mostri per collezionarli in un inventario.
Soluzione: Per poter gestire al meglio l'attraversamento di porte e scale da parte dell'eore si è deciso di fare un apposito metodo all'interno delle rispettive classi che registra l'interazione con l'oggetto e di conseguenza richiama il corrispondente metodo, ChangeRoom per le porte e ChangeFloor per le scale. Per quanto riguarda drop e forzieri, invece, bisogna vedere che oggetti generano e successivamente controllare che siano presenti nell'inventario una volta avvenuta l'interazione con essi da parte dell'eroe.

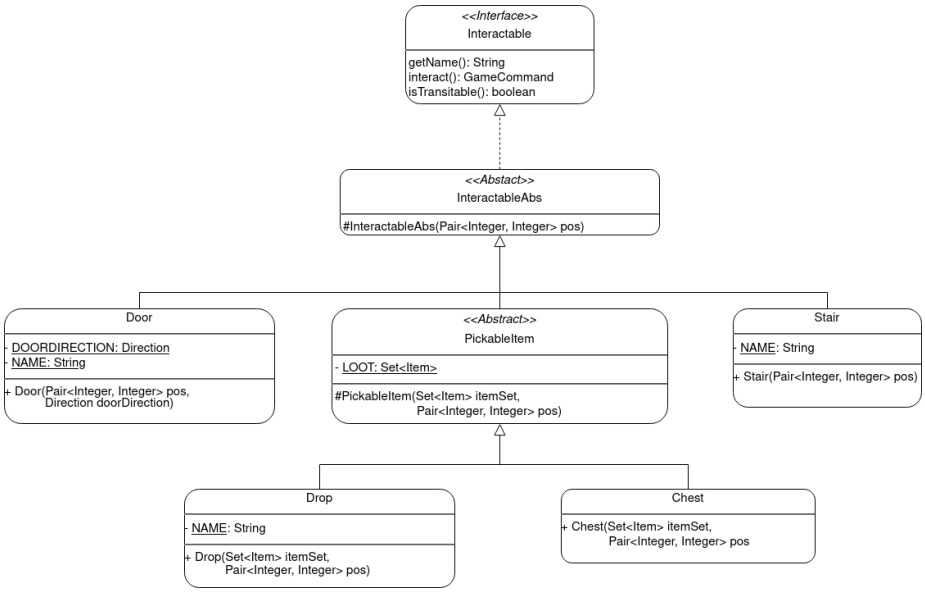

Figura 2.16: Uml degli oggetti interagibili

Leonardo Marcaccio

Prima di poter parlare dei pro

Entità, chi e dove?

Ogni stanza è di per se un contenitore, ma di cosa? Per poter essere popolata abbiamo biogno di un qualcosa che ci permetta di dire in ogni momento chi sia e dove si trova, questo qualcosa nel nostro progetto prende il nome di Entità. Il concetto di Entità dunque rappresenta tutte le cose che necessitano di tale caratteristiche, ovvero:

- Gli Attori, tutti coloro che si devono muovere all'interno delle stanze.
- Gli Interagibili, tutti quegli elementi che permettono l'interazione all'interno delle stanze.
- Le Tile, le varie celle che compongono la mappa di gioco

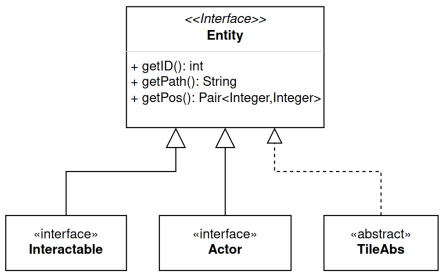

Figura 2.17: Uml dell'Entità

Attori, i protagonisti della scena

Il primo problema che mi sono trovato ad affrontare è: come creare una struttura solida che mi permetta di creare tutte quelle Entità che interagiscono con il mondo circostante?

La mia soluzione è stata creare un concetto che racchiuda questa caratteristica, ovvero gli Attori.

Essi rappresentano tutti quegli elementi che "Recitano" all'interno della mappa di gioco, ovvero che compiono diverse azioni come muoversi, attaccare o interagire con il mondo di gioco.

Gli attori possono essere di due tipi che si differenziano in base al modo in cui scelgono e eseguono tali azioni, essi sono l'Eroe e i Mostri.

Ogni Attore per compiere le sue azioni ha bisogno di due elementi fondamentali

- Le Skill, l'insieme di tutte le azioni possibili che un Attore può compiere
- Le Stat, l'insieme di tutte le statistiche dell'Attore.

Qui sorge però sorge un problema, l'Eroe rappresenta il personaggio controllato dal giocatore mentre i Mostri sono Attori che agiscono indipendentemente dalla volonta del giocatore. Quindi come si possono creare due oggetti con tanti elementi simili ma con due modi di agire diversi?

All'inizio avevo pensato di sviluppare il tutto attraverso il Template Pattern, ma alla fine ho optato per l'implementazione più semplice astraendo il concetto di Attore in un elemento a parte ActorAbs il quale viene poi gestito indipendentemente dall'Eroe e dai Mostri. in quanto applicare tale Pattern portava a delle forzature nel codice.

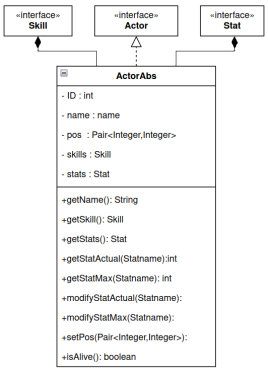

Figura 2.18: Uml degli Attori

L'Eroe

Come precedentemente annunciato nello scorso punto l'Attore che permette al giocatore di interagire con l'ambiente di gioco è l'Eroe.

Esso dunque deve agire di conseguenza e venir comadato dagli input dati dal giocatore.

Quindi il suo agire consiste in nient'altro che nell'interpretare i comandi del giocatore e trasformarli in azioni che dovrà eseguire.

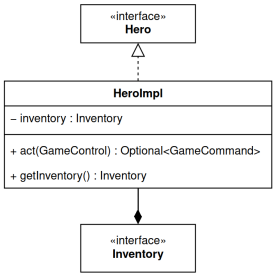

Figura 2.19: Uml degli Eroi

Altro punto da tener conto è che il giocatore deve poter far utilizzo di ciò che ha ottenuto durante il gioco e necessita dunque di un Inventario che gestisca tutti gli Oggetti che l'Eroe ha raccolto, equipaggiato o vuole usare.

L'Inventario è composto sostanzialmente da 2 parti: gli Slot dell'Equipaggiamento e una Borsa per tutti gli Oggetti che il giocatore ha raccolto.

Gli Oggetti si suddividono in 2 tipi, gli Equipaggiamenti, ovvero oggetti particolari che l'Eroe può equipaggiare per alterare le sue Statistiche e ottenere nuove abilità, e i Consumabili, oggetti per recuperare o potenziare le Statistiche dell'Eroe.

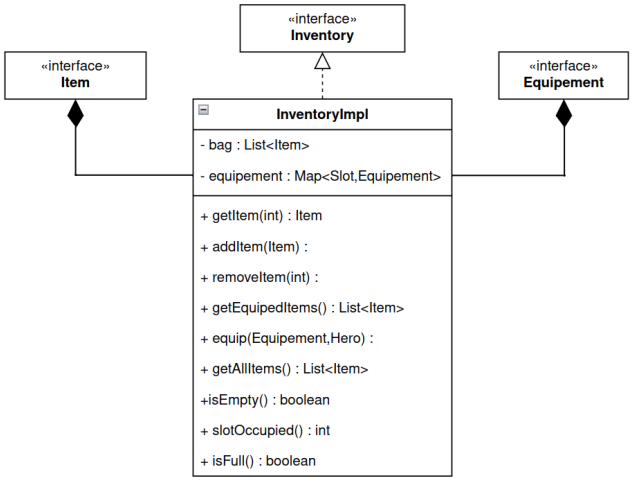

Figura 2.20: Uml dell'Inventario

Gli Oggetti

Un'altro punto che ho dovuto affrontare è il come vengano creati i vari tipi di oggetti.

Per la realizzazione di tale scopo ho deciso di utilizzare due differnti istanze del Factory Pattern, una per gli Oggetti di tipo Consumabile e una per quelli di tipo Equipaggiamento.

Tale scelta mi ha permesso di ridurre notevolmente la dimensione del codice e la eliminandone la sua duplicazione soprattutto nella Factory dedicata agli Equipaggiamenti che fa utilizzo di un'altra Factory di supporto dedita alla creazione di Abilità fornite dagli Equipaggiamenti, in quanto alcuni essi modificano particolari attacchi del Eroe (Ad esempio una spada non può fare attacchi a distanza mentre un arco permetterebbe al giocatore di farlo).

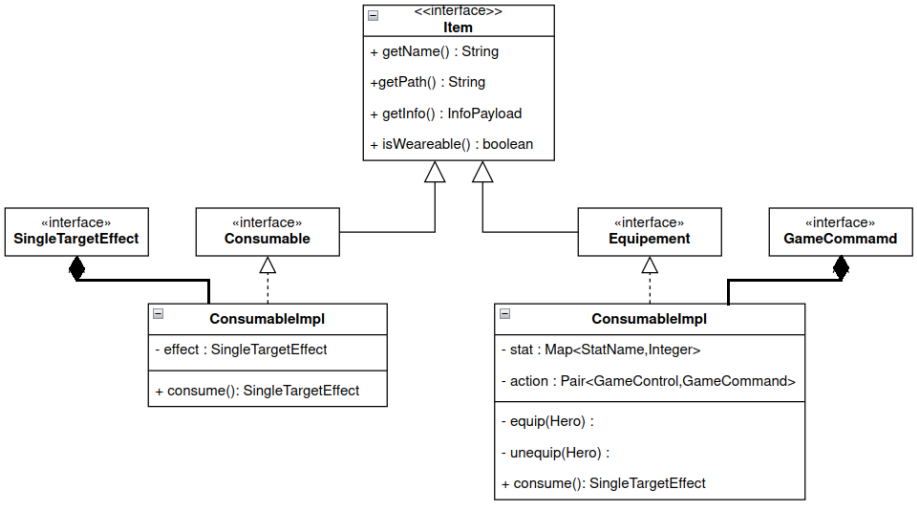

Figura 2.21: Uml degli Item

I Mostri

Se per l'Eroe il problema della scelta dell'azione non si pone, per i Mostri non vale lo stesso discorso.

Infatti i Mostri necessitano di un sistema che gli permetta di scegliere autonomamente tali Azioni, la mia soluzione è stata quella di creare un Comportamento composto da due elementi un Comportamento di Movimento e uno di Combattimento.

Il Compoortamento di un Mostro sostanzialmente controllerà per il Mostro se può attaccarequalcosa, altrimenti cerca di muoversi e alla fine, se non può fare nulla, resta immobile.

Questo Comportamento come le Statistiche e le Abilità dei Mostri vengo creati attraverso un sistema modulare di Factory ogniuna dedita alla creazione di una componente specifica.

La scelta di usare questo sistema di Factory è stata presa per permette una facile estendibilità, manutenzione e facilità di utilizzo che permette con pochi passi di generare qualsiasi mostro del gioco.

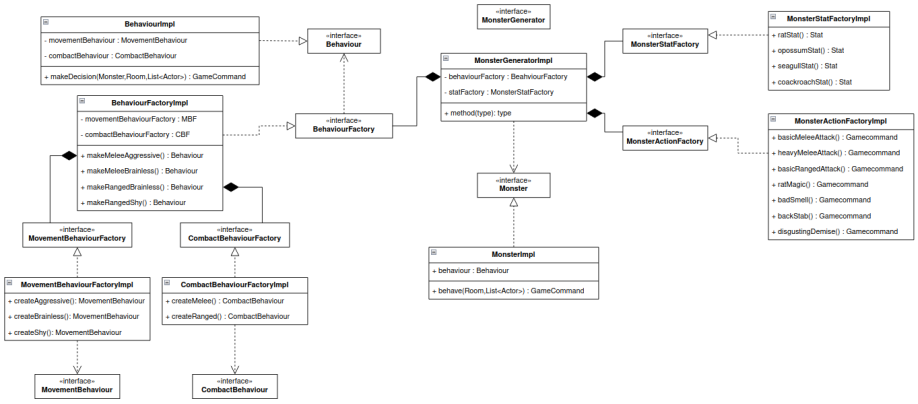

Figura 2.22: Uml dei Mostri

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Ogni componente del team ha realizzato almeno un paio di test completamente automatizzati per le proprie classi, giusto il necessario per dimostrare che ogni componente ha capito il necessario che la libreria JUnit offre ai programmatori per realizzare dei test. Di seguito ogni componente descrive sommariamente i propri test:

Mappa di gioco
A cura di Anny Bevilacqua

Essendo la creazione dei piani e delle stanze una delle meccaniche cardinali del gioco, si è deciso di porre sotto esame le principali implementazioni che contribuiscono al processo di generazione della mappa. Per strutturare le classi di test, al contrario che nella fase di design, è stato attuato un approccio bottom-up, partendo quindi dal collaudo delle meccaniche legate alle Tile per arrivare fino all'intero piano.

Tiles e TileFactory

Le Tiles sono di per sé oggetti relativamente semplici, le cui meccaniche possono tuttavia avere un impatto notevole sul gameplay. I test sviluppati riguardano:

Accessibilità della Tile Dovendo le Tile rappresentare porzioni di spazio all'interno di una stanza, ci interessa,per prima cosa sapere se gli attori potranno muoversi su di esse o, se queste rappresentino porzioni inaccessibili della stanza. Per esempio, una Tile rappresentante un muro dovrà ostacolare il movimento del giocatore e non potrà essere attraversata in alcun modo, risultando quindi inaccessibile. Una Tile rappresentante una porzione di pavimento potrà invece essere transitabile a meno che non sia occupata da un oggetto invalicabile.

Posizionamento di oggetti Come accennato sopra, alcune Tile potranno contenere oggetti. Dovremo quindi testare il corretto posizionamento di un oggetto sopra a una Tile e i diversi metodi per recuperare un suo riferimento. Nello specifico vogliamo assicurarci che:

- in nessuna occasione sia permesso posizionare oggetti sopra a una Tile di tipo muro o trappola
- se su una Tile vi sia già un oggetto presente, qualsiasi tentativo di "sovrascrivere" il contenuto senza prima averlo rimosso abbia esito negativo
- in nessuna occasione sia permesso di posizionare una Tile all'interno di un'altra Tile
- usando il metodo per prendere il riferimento al contenuto della Tile, questo non venga eliminato
- dopo aver svuotato la Tile sia nuovamente possibile posizionarvici oggetti sopra

Effetti speciali Alcune Tile, come le trappole o le pozzanghere, potranno infine avere degli effetti collaterali sugli attori che vi passano sopra. Andranno pertanto testate la corretta applicazione degli effetti e l'esecuzione delle operazioni post attivazione, in caso fossero presenti. In questo caso, poiché le trappole hanno un comportamento più complesso rispetto alle pozzanghere, si è deciso di concentrare i test su di esse, ponendo attenzione anche ad alcune loro meccaniche, come l'interazione, la modifica del danno inflitto o la definizione delle operazioni da eseguire dopo l'azionamento.

Generazione delle Tile Per testare la generazione delle tile, oltre a verificare il corretto funzionamento del costruttore, si è deciso di effettuare alcuni accertamenti ai metodi contenuti nella classe TileFactory. I controlli eseguiti sono di carattere generale, interessando quelle che sono le proprietà principali delle tile create.

Rooms e RoomFactory

Una volta testate le Tile, viene da sé che dovremmo essere in grado di generare una stanza vuota senza temere comportamenti inaspettati. Il prossimo passo nella generazione della mappa è quindi la costruzione delle singole stanze ed il loro popolamento.

Dimensione delle stanze Al fine di fornire maggior coesione a livello grafico durante il gameplay e più in particolare al passaggio da una stanza all'altra, si è deciso di definire una dimensione minima e una massima per la grandezza delle camere generate. Andranno quindi testate l'impossibilità di creare stanze che non rispettino tali misure e i controlli eseguiti per assicurare il contenimento di una posizione nelle stesse.

Aggiunta di oggetti Sono stati successivamente implementati una serie di metodi legati alla possibilità di aggiungere oggetti nella camera, secondo le logiche già viste nelle Tiles. Questa categoria di test include anche l'aggiunta di porte e scale lungo i muri della mappa. Le funzioni che se ne occupano, infatti, dovranno prima rimpiazzare la Tile muro con delle Tile di pavimento e solo successivamente potranno posizionare la porta o rampa di scale in questione. Inoltre, essendo queste ultime aggiunte in modo casuale lungo il muro della stanza, è bene assicurarsi che non siano posizionate in Tiles già occupate o negli angoli della camera.

Generazione di stanze Essendo la creazione di stanze un evento frequente nel gioco, è sembrato più che dovuto effettuare dei test sulla loro generazione. Sono quindi stati creati dei test che ricalcano i metodi presenti in RoomFactory. I test si assicureranno la corretta creazione di stanze con e senza trappole, così come il loro popolamento con porte, oggetti e mostri. Va notato che, per queste ultime due categorie, i test non verificano l'effettiva aggiunta dei singoli alla stanza, ma piuttosto l'upper bound indicante la quantità massima consentita, in base alla grandezza della stanza stessa. Infatti, seppur la creazione e il popolamento avvengono in maniera casuale, si desidera, soprattutto ai livelli più bassi, mantenere un numero contenuto di nemici, in modo da non sopraffare il giocatore e rendere l'esperienza di gameplay più piacevole.

Floors

Infine per i piani si è deciso di testare il comportamento al cambiamento di stanza, assicurando la creazione dinamica delle stanze e l'impossibilità di cambiare stanza in caso di porta non presente.

Bandiera Luca

Per quanto riguarda il package degli Interactable si è testato per prima cosa l'interazione con le Chest andando a verificare che gli oggetti contenuti in essa, una volta avvenuta l'interazione da parte dell'eroe, venissero tutti trasferiti nell'inventario dell'eroe stesso.
Oltre a questo test, si è voluto controllare anche il fatto che un determinato Interactable fosse attraversabile o meno assicurandosi che Stair e Door lo fossero, mentre Chest e Drop no.
Infine nella classe di testing QuickCommandTest di Pesaresi Jacopo è stato fatto un test per controllare che quando l'inventario è pieno l'eroe non può più raccogliere oggetti, mentre quando non è pieno l'eroe può continuare a raccogliere oggetti.

Pesaresi Jacopo

Sono state realizzate essenzialmente tante classi di testing quanti package di classi di cui sono il diretto responsabile. In particolare, per il package:

- Range: si è testato se i vari range contavano correttamente quanti nemici vi fossero nella loro area (il Circle e il Cone non hanno l'area estesa come il Global, che infatti, se si determina la posizione di un mostro molto lontana da quelli considerati, Global riesce comunque a contarlo, mentre gli altri dure no, Infine, Global non si estende sui muri, quindi se un mostro si trova un muro (che sarebbe però un errore di programmazione) allora conterà solo i mostri vicini)

- Drop: essendo che si sono attuati delle estrazioni prevalentemente random, si è controllato se l'estrazione randomica estraesse solo un sottoinsieme (proprio o improprio) dell'insieme degli elementi salvati all'interno dello stesso DropManager.
  Inoltre, si è testato se la generazione del loot di una stanza (riposto poi nelle chest) è diverso almeno da quella precedente

- Command: si sono testati tutte le caratteristiche di architettura, e una buona parte del funzionamento dei comandi. Per comodità si è deciso, in realtà, di dividere l'insieme dei test in due classi, in base se si dovesse controllare un comando rapido o uno complesso. Si è incluso il controllo della architettura nella prima delle due classi. Si è lasciato sviluppare il test su un comando quick dal team. Si rimanda a documentazione dove sono specificati i credits

Marcaccio Leonardo

Sono state realizzati test nelle classi create da me che hanno creato problemi durante lo sviluppo ed usate per migliorare il codice attraverso un processo Test Driven. Le classi testate sono:

- Inventory: si è testato i vari metodi della classe InventoryImpl come l'aggiunta e la rimozione di Oggetti dall'Inventario e l'equipaggiamento e il disequipaggiamento di Equipaggiamenti creati dalla rispettiva Factory.

- MonsterGenerator: si è testato la generazione di ogni Mostro presente attualmente nel gioco, comparandone le Statistche e Abilità con quelle delle Factory addette alla generazione delle Statistiche e Abilità dei Mostri.

- MovementBehavioru: si sono testati tutti i possibili movimenti di ogni tipo di Comportamento di Movimento presente al momento nel gioco(Aggressivo, SenzaCervello/Randomico e Timido).

3.2 Metodologia di lavoro

Eseguita la fase di analisi, si è tentato di instaurare un UML che fosse in grado di risolvere quante più richieste finali espresse in analisi, ma alla fine si è deciso di farne una ridotta per poi espanderla mano a mano.
Si può trovare conferma di ciò osservando il work-flow del progetto: inizialmente

sono nati due branch principali, quello di "release", in cui mettere le versioni deployabili del programma, mentre in quello di "develop" riversare tutti i nuovi sviluppi. Da quello di develop è stata poi una continua creazione e interruzione di branch, necessari sia se si reputava di fare aggiunte di nuovi componenti al progetto sia se era necessario apporre drastiche modifiche a quanto già precedente. Altrimenti per piccole modifiche, o per le ultime prima della consegna, il team si è concentrato a rimanere sul develop
Per lasciare nel develop un minimo di stampe di debug, si è infine deciso, prima di aggiornare il release con l'effettiva versione finale, di creare un branch apposito di `"pre_release"` in cui togliere tutto quello che potesse aiutare il debug e quindi affinare la versione finale

Anny Bevilacqua

Approccio Generale Per la mappa di gioco, la metodologia di lavoro utilizzata ha seguito un andamento a spirale, alternando una fase di raccolta dei requisiti a una di implementazione e di raffinamento. Si è partiti esaminando i requisiti introdotti già nel capitolo di analisi, individuando le entità astratte in gioco ed abbozzando una prima versione delle interfacce. Successivamente, assieme agli altri componenti del gruppo, sono stati accordati alcuni metodi pubblici in grado di definire l'interazione tra le classi della mappa e le altre componenti del gioco. I metodi in questione fanno prevalentemente parte di una serie di getter e setter usati per accedere alle Tiles e al loro contenuto.

Si è quindi deciso di creare un package all'interno del model per contenere i file inerenti alla mappa e alla sua generazione, permettendo di mantenere una struttura del filesystem organizzata e facilitando l'identificazione di classi e interfacce coinvolte. Allo stesso modo, si è cercato di mantenere gli sviluppi su un branch separato del DVCS, in modo da non ostacolare il lavoro dei compagni con eventuali refusi e bozze incomplete e mantenere minime le dipendenze da altri package.

Dopo aver definito in linea generale quali campi e metodi dovesse contenere ciascuna classe, si è deciso di partire con l'implementazione delle Tiles. Delineati i comportamenti base e più comuni e racchiusi gli stessi in una classe astratta, sono state create due implementazioni distinte, rappresentanti rispettivamente pavimento e muri: due concetti fondamentali di quella che sarà la composizione della mappa. Come accennato in precedenza, si è cercato di prestare particolare attenzione al concetto di modularità e riusabilità, assicurandosi un'implementazione abbastanza flessibile da accomodare agevolmente future modifiche e aggiunte. Passando all'implementazione della stanza, si è optato per un modello rettangolare, come specificato dai requisiti del gioco, non mancando di lasciare spazio a prossimi sviluppi. Sono quindi state definite le regole per il posizionamento delle Tiles e che struttura avrebbe dovuto contenere oggetti e mostri racchiusi nella camera. Fatto ciò, sono stati creati i primi metodi nella factory per agevolare la creazione delle Room e permettere lo sviluppo dinamico della mappa.

Definita la modalità di creazione basilare, si è pensato a come i vari attori in gioco avrebbero potuto interagire con le mattonelle e come il giocatore avrebbe potuto passare da una stanza all'altra. Si è quindi pensato di implementare delle speciali Tiles rappresentanti porte e passaggi delle stanze. Successivamente questa idea è stata abbandonata in favore del posizionamento di oggetti interagibili di tipo Door sulle mattonelle di pavimento già presenti. Il concetto di Tiles interagibili è tuttavia tornato in auge quando, rileggendo i requisiti, si è notata la necessità di aggiungere delle rappresentazioni per il concetto di trappole. Per realizzare le nuove entità si è quindi deciso di estendere l'implementazione delle mattonelle di pavimento, implementando, oltre che l'interfaccia Tile, quella Interactable, già sviluppata dal collega Luca Bandiera e aggiungendo la possibilità di generare effetti collaterali sugli attori del gioco. Per le normali unità di pavimento questo effetto non apporterà alcun cambiamento all'attore, tuttavia le trappole potranno andare ad agire sulla salute del giocatore e dei mostri, causando dei danni su chi vi dovesse passare sopra. Gli effetti causati dalle trappole vengono registrati ed applicati grazie all'integrazione dei comandi sviluppati da Jacopo Pesaresi nella gestione dei movimenti.

Il comportamento inverso si ha invece per il cambio di stanza: una volta registrato l'evento nel model, questo viene passato al piano corrente della mappa, che si occupa di gestire la sostituzione della stanza corrente con quella nuova.

Infine, anche il popolamento delle stanze è stato fatto integrando due diverse factory, sviluppate rispettivamente da Leonardo Marcaccio e Jacopo Pesaresi, la prima per la creazione di mostri e la seconda per quella degli oggetti. In particolare per il posizionamento degli oggetti è stata utilizzata anche la classe Chest. Tale classe, sviluppata da Luca Bandiera per la categoria degli Interactable, essendo in grado di contenere più items contemporaneamente e di passarli al giocatore su interazione, è stata selezionata per rappresentare i forzieri presenti nelle stanze. Sia per i forzieri che per i mostri è stato definita una soglia massima di istanze generabili in una singola stanza. Tale numero è stato generato a partire dal livello corrente e dalla grandezza della stanza stessa, permettendo di aumentare la difficoltà del gioco con il superamento dei piani, senza occludere esageratamente lo spazio disponibile nelle stanze.

Si accenna infine ad alcuni contributi apportati alle classe GameModelImpl, sempre legati alla gestione della mappa e all'implementazione della View.

Luca Bandiera

Interactable

Per quanto riguarda gli oggetti Interctable, come specificato in precedenza, ovvero quelli oggetti che hanno bisogno di un'interazione, è stato deciso di creare un package apposito nel model.
Sono partito creando un'interfaccia comune per tutti e quattro i tipi di Interactable (door, stair, drop e chest). Successivamente ho diviso gli Interactable in due parti in base a caratteristiche comuni.
Da una parte ci sono Door e Stair in quanto entrambi servono per portarmi in

una stanza nuova (stanza adiacente nel caso di Door e prima stanza del piano nel caso di Stair).
Dall'altra parte ci sono Chest e Drop in quanto entrambi servono per mettere nell'inventario dell'eroe degli oggetti (nel primo caso gli oggetti che sono contenuti all'interno della Chest, nel secondo caso gli oggetti posseduti da un mostro una volta morto).
Door e Stair estendono direttamente da InteractableAbs (la classe astratta dell'interfaccia Interactable) e ciascuno implementa il proprio metodo (rispettivamente ChangeRoom() e ChangeFloor()). Drop e Chest invece estendono PickableItem, che a sua volta esente Interactable, in quanto sono due oggetti diversi che però devono fare la stessa cosa, riempire l'inventario dell'eore con gli oggetti situati all'interno di un forziere nel caso delle Chest, quelli posseduti dal mostro nel momento in cui muore nel caso dei Drop. Il loro metodo per gestire l'interazione sarà quindi lo stesso in entrambi i casi.

View

Si è deciso di sviluppare la view in FXML, in quanto offre vari vantaggi tra cui la separazione, in file diversi, tra la logica della view e la sua rappresentazione grafica.
Ho deciso di creare due package appositi per la sua gestione. Un package è situato nelle risosrse del progetto e contiene i file fxml della view (game-view, game-over, menu-iniziale), ovvero la loro rappresentazione grafica. L'altro si trova nel package java insieme agli altri package (model, controller..). In questo package ho creato 3 classi (MainMenuController, GameViewController e GameOverController) che si occupano di controllare e gestire i rispettivi file fxml (gestire le stampe del gioco, layout delle finestre...).
Il file FXMLMainView è il controller di view e si occupa di "controllare" appunto il tutto (per esempio il fatto che quando nel menu princiaple clicco sul pulsante "Nuova partita" devo caricare la view del gioco e inizializzare una nuova partita, oppure il fatto che quando si vince o perde si deve caricare la rispettiva schermata di vittoria o perdita). L'interfaccia GameView, estesa dalla classe FXMLMainView, contiene tutti i metodi principali di gestione del gioco come per esempio l'inizializzazione di una nuova partita, stampare le informazioni a video e l'apertura dell'inventario. L'interfaccia Drawable e la sua rispettiva implementazione servono per facilitare la view con la gestione delle stampe.

Sviluppo condiviso

Parti che ho sviluppato insieme ai miei colleghi:

- Ho collaborato con Jacopo Pesaresi per lo sviluppo della classi FillInventory e ChangeSituation all'interno del package dei comandi.
- Progettazione e sviluppo del MVC.

Jacopo Pesaresi

Volendo fare un commento generale sulla metodologia, ho cercato sin dai primi istanti di progetto di implementare, solo dopo aver concretizzato un UML, tutte le classi. Quando poi ne ho potuto effettivamente verificare il funzionamento, sono ricorso a una serie di aggiustamenti che permettesso di tenere intatto lo schema presupposto. Di seguito specifico quali classi sono state sviluppate da me direttamente, cogliendo l'occasione per approfondire l'implementazione dietro ai desing prima descritti, per poi citare in quali classi ho dato contributo.

Package gamecommand

Dopo aver analizzato l'architettura nel desing, in questa sezione mi limito ad affermare che le classi che non compaiono nello schema UML sono principalmente di mia responsabilità (ovviamente lo sono anche quelle che lo costituiscono). In realtà, avendo praticamente centralizzato l'interazione utente-model, ho lasciato implementare basilarmente qualche comando anche al team. Classi quindi come "ChangeSituation" e classi che la estendono, "FillInventory" e "OpenInventory" sono sì di mia responsabilità ma solo in parte da me sviluppate (le ho principalmente adattate e mantenute, poco più).

La caratteristica importante che si vuole sottolineare è che le classi di gamecommand che non compaiono in UML estendono da *solo e solamente* una delle classi astratte (oltre a essere un vincolo di Java è in realtà un vincolo concettuale progettuale a tutti gli effetti, non ha senso che un comando sia "complesso e quick", "handlable e IA"), ma per evitare di risultare verboso, si rimanda a documentazione per capire bene la sfumatura che ogni classe aggiunge al comando finale

Tuttavia, si desidere esplicitare veloci riflessioni (lasciate intendere nello schema UML ma qui specificate per emergere alcune delle classi da me implementate):

- l'implementazione dei metodi HandableGameCommand e IAGameCommand in QuickActionAbs prevede lanciare eccezzioni che non sono gestite in alcuna parte del gioco (servono infatti allo sviluppatore per aiutare a sviluppare dei nuovi comandi veloci: essendo appunto "veloci" su questi oggetti non si deve lanciare metodi che permettono invece la loro configurazione!)

- in modo abbastanza analogo, comandi che gestiscono situazioni complesse estendono da una delle due classi:

  - NoIACommand: classe astratta che implementa però i metodi di IAGameCommand, facendoli lanciare delle eccezioni
  - NoPlayerCommand: classe astratta che implementa però i metodi di PlayerGameCommand, facendoli lanciare delle eccezioni

(l'utilità delle eccezzioni è la stessa vista nel punto precedente: comandi che richiedono l'interazione con l'utente non devono essere gestiti dal gioco e viceversa)

- per quanto riguarda l'instaurazione dell'interfaccia IAGameCommand, è affermabile che la sua definizione è stata effettuata in collaborazione con il team (in particolare con Leonardo Marcaccio) al fine di migliorare la sincronia di lavoro

Durante le prime prove del programma, avevo però sin da subito capito che in realtà fatto un *"new comando()"* questo oggetto rimane salvato nella mappa, quando invece ero ancora rimasto con la mentalità dei builder, e quindi ero convinto venisse ricostruito da zero. Pertanto, per aggiustare i comandi, si sono dovute aggiungere delle flag che permettono di impostare lo stato interno del comando (quantomeno ciò per i comandi complessi)

Package range

Le classi in questo package sono tutte del sottoscritto, sia a livello di responsabilità che implementativo. Le classi che implementano DecoratedRange, quali CircleRange, ConeRange, ecc, si servono per calcolare l'area di algoritmi implementati nella classe apposita Pairs, sviluppata interamente dal sottoscritto secondo proprio diletto matematico.
Questa classe di utility in realtà serve anche ogni qualvolta che si desidera calcolare un nuovo Pair adiacente ortogonalmente al precedente: offre infatti delle funzioni che permettono di centralizzare questo calcolo. Desidero sottolineare questo aspetto perchè è stato di vitale importanza in fase di collaudo in quanto ci ha permesso di sincronizzare subito una piccola imcomprensione emersa nel team (per aggiustarci è stato sufficente negare una condizione che invertisse a quale coordinata applicare l'offset)
In ogni caso, se un giorno si volesse calcolare le varie aree basandosi su nuove idee o magari su nuove classi (basta che mantengano il concetto di calcolo dell'area partendo da chi ha lanciato l'attacco verso eventuali punti di termine), è sufficiente modificare solo un paio di elementi all'interno della sola classe Range.

L'unico Range che si discosta da tutta questa logica è il "GlobalRange". È la classe che concettualizza un range avente la stessa forma della stanza (utile se per esempio esiste un attacco che ha appunto range globale, svincolato da barriere di ogni genere). Allora "Global Range" sarà sicuramente una classe che estende da "DecoratedRange" (prende nel costruttore uno delle classi-mattoncino base del pattern e la estende a tutta la dimensione della stanza) e che è costretta a sovrascrivere i metodi previsti da DecoratedRange", essendo che NON deve incapsulare un concetto di "radialità"

Package drop

Per questo package (anche questo di mia totale implementazione) si desidera solo far emergere che l'inizializzazione delle strutture dati che contengono ogni

possibile item viene effettuato una sola volta, in quanto altrimenti ogni volta che viene creata una factory le liste verrebbero popolate continuamente e in queste strutture dati ci sarebbero allora inutili doppioni

Package outputinfo: comunicare con la view

In questo package ci sono le classi utili a comunicare con la view, e quindi principalmente i concetti di InfoPayload e Portrait, che non hanno particolari necessità e che quindi si è semplicemente prevista la coppia implementazione-interfaccia. Notare come siano praticamente assenti di particolari design (ed ecco perchè assenti nel relativo paragrafo)

Altre classi varie ma di mia sola responsabilità

Per aiutarmi a implementare certi desing, o certi algoritmi, mi affermo responsabile anche di:

- tutte le classi-eccezioni di tipo "ModelException"
- la classe MyIterator, che altro non è un iterator che offre due metodi in più: un reset (reimposta l'iteratore alla partenza) e un "getActual", al fine di non doversi salvare in una variabile apposta l'eventuale next chiamato su un Iterator semplice

Sviluppo condiviso

Parti che ho sviluppato insieme ai miei colleghi:

- La progettazione e l'implementazione di Effect e di EffectAbs
- "L'aumento di prestazioni" di rendering: progettato e implementato qualche metodo che permettesse di stampare un pò più velocemente le entità da aggiornare. Per farlo mi sono servito di una convesione, da Entity a Drawable (sono allora anche responsabile di Drawable e di DrawableImpl).
- La progettazione e l'implementazione di interfaccia-implementazione del MVC

Leonardo Marcaccio

Package actors

Il package actors è uno dei due "macro-package" che ho sviluppato e come definito precentemente nella sezione di design è il package che contiene tutti gli elementi legati agli Attori del gioco.

Questo package è stato subito creato dopo la creazione dell'interfaccia Entity e da qui sono partito con lo sviluppo delle sue due classi: Actor e la sua classe astratta ActorAbs che servono da nucleo centrale per tutto quello che verrà poi

sviluppato nei suoi sotto-package.

Il primo sotto-pakage che ho sviluppato è stato quello di hero all'interno del quale ho sviluppato la prima implementazione dell'Eroe del nostro gioco, questa implementazione va ad estendere quello che è stato implementato precedentemente in ActorAbs andando ad aggiungere i metodi act() e successivamente getInventory().

Dopo hero sono subito passato allo sviluppo di skill e stat entrambi usati da hero e successivamente da monster.

Fatta questa fase di sviluppo ho deciso poi di concentrarmi sul package item e i suoi sotto-package per dare spazio di lavoro ai miei colleghi di usare quello che avevo già sviluppato.

L'ultimo package sviluppato di questo blocco è stato monster, il package più carico di elementi sviluppati. Qui ho sviluppato tutta la struttura modulare della generazione dei mostri attraverso una serie di Factory gerarchiche che costruiscono le singole componenti del Mostro per culminare nel MonsterGenerator.

Le factory di questo sistema gerarchico comprendono la MonsterActionFactory, la MonsterStatFactory e la BehaviourFactory, la quale a sua volta è composta da due factory la MovementBehaviourFactory e la CombactBehaviourFactory.

Tutto quello che riguarda il behaviour è stato messo nell'omonimo package all'interno del quale appunto viene raccolto il processo di creazione del Comportamento di un Mostro partendo dai singoli comportamenti, quello di Movimento e quello di Combattimento, terminando nella loro unione attraverso la BehaviourFactory che restitusce un Behaviour completo usato poi nel MonsterGenerator.

Package Item

Il secondo macro-package che ho creato è stato Item il package dedicato agli Oggetti del gioco e all'Inventario che utilizzerà l'Erore.

Per prima cosa ho creato l'Inventory che non è altro che una classe che si occupa appunto di gestire tutti gli Oggetti che il giocatore ha ottenuto durante la partita.

Dall'implementazione dell'Inventario sono poi passato subito allo sviluppo degli Item, con un approccio simile a quello che ho avuto per gli attori. Infatti l'interfaccia item serve solo ad identificare cosa sia un oggetto indicandone i principali elementi che deve rappresentare, di cui il più importate è il meto-

do isWearable() che definisce se un oggetto è Indossabile o meno, ovvero se è equipaggiabile o no.

Dalla creazione di Item ho poi creato due package Consumable e Equipement che contengono tutto ciò che definisce e serve a creare rispettivamente i Consumabili e gli Equipaggiamenti.

Sia per Consumable che per Equipement ho riusato delle Factory per la creazione di tutti gli oggetti di gioco implementando in Equipement anche una Factory dedita alla creazione delle Abilità fornite dai vari Equipaggiamenti

Sviluppo condiviso

Parti che ho sviluppato insieme ai miei colleghi:

- Durante lo sviluppo dei Consumable sono stato a stretto contatto con Jacopo pesaresi e nel suo sviluppo di Effect e di EffectAbs, utilizzato poi in Consumable e Skill
- Durante lo sviluppo del Behaviour dei Mostri ho collaborato con Jacopo Pesaresi, il quale attraverso lo sviluppo degli IAGameCommand ha permesso uno sviluppo ottimale per le Skill dei Mostri.
- Supporto allo sviluppo del sistema di MVC insieme ai vari membri del gruppo per la rimozione di errori e implementazione delle varie componenti

3.3 Note di sviluppo

Anny Bevilacqua

Optionals

- lambda expressions - `https://github.com/ZhiAn-ny/OOP22-rusco-dc/blob/34222d787734cc759532118ce42845a4d9b68539/src/main/java/it/unibo/ruscodc/model/gamemap/RectangleRoomImpl.java#L123`

Streams

- stream - `https://github.com/ZhiAn-ny/OOP22-rusco-dc/blob/34222d787734cc759532118ce42845a4 src/main/java/it/unibo/ruscodc/model/gamemap/RoomFactoryImpl.java#LL58C1-L64C24`

Lamba expressions

- optional - `https://github.com/ZhiAn-ny/OOP22-rusco-dc/blob/34222d787734cc759532118ce42845 src/main/java/it/unibo/ruscodc/model/gamemap/FloorTrapTileImpl.java#L80`

Luca Bandiera

FXML

Grazie a questo progetto ho imparato ad usare il linguaggio FXML per l'implementazione della view.
Permalink: https://github.com/ZhiAn-ny/OOP22-rusco-dc/blob/34222d787734cc759532118ce42845a4d9
src/main/resources/it/unibo/ruscodc/view/game-view.fxml#LL1C1-L1C39

JavaFX

Sempre per l'implementazione della view ho utilizzato anche JavaFX.
Permalink a una feature particolre di JavaFX: https://github.com/ZhiAn-ny/OOP22-rusco-dc/blob/34222d787734cc759532118ce42845a4d9b68539/src/main/java/it/unibo/ruscodc/view/GameViewController.java#L162

Jacopo Pesaresi

Uso degli Stream

Sono state sfruttati molto ampliamente in tutte le classi da me direttamente sviluppate. Affermabile che il massimo sfruttamento di questa utility di Java è stato fatto nella classe di utility "Pairs.java" (il perchè di questo ""abuso"" nella classe l'ho spiegato nella sua documentazione).
Permalink (alla documentazione della classe, quindi sostanzialmente alla classe stessa): https://github.com/ZhiAn-ny/OOP22-rusco-dc/blob/34222d787734cc759532118ce42845a4d9b6
src/main/java/it/unibo/ruscodc/utils/Pairs.java#LL12C4-L18C9

Uso della reflection e di lambda

Per aiutare il meccanismo di generazione dei drop, e per rendelo dinamico all'aggiunta a posteriori di nuovi oggetti, si è serviti dei meccanismi offerti dalla reflection per popolare, a caricamento di gioco, le liste contenenti ogni possibile drop. Essendo che erano necessari tutti i metodi delle relative factory men quelli di "Object", per poter generare le strutture dati contenenti gli oggetti si sono considerati i metodi della classe factory, poi quelli di object, quindi si è eseguita una differenza insiemistica al fine di avere solo quelli in interesse, da invocare su una classe costruita al volo e quindi popolare le liste
Permalink (reflection): https://github.com/ZhiAn-ny/OOP22-rusco-dc/blob/34222d787734cc759532118ce42845a4d9b68539/src/main/java/it/unibo/ruscodc/model/actors/monster/drop/DropFactoryImpl.java#LL54C24-L54C58
(si è consapevoli che il meccanismo funziona finchè le classi-factory degli item mantengono i propri metodi con parametroRitorno-signature "*Item* generateItem(*void*)")
A posteriori le liste dei drop sono diventate liste di supplier. Propongo qui sotto allora un chiaro esempio di uso delle lambda nel mio codice (per altri esempi basta vedere a tutte le lambda create in Pairs)
Permalink (lambda relativa): https://github.com/ZhiAn-ny/OOP22-rusco-dc/

blob/34222d787734cc759532118ce42845a4d9b68539/src/main/java/it/unibo/ruscodc/model/actors/monster/drop/DropFactoryImpl.java#L52

Uso dei generici

Per alleggerire il meccanismo di stampa, si è aggiunta una interfaccia Drawable con tipo generico X per astrarre il tipo di immagine che la view implementa nella stampa a video (quella che per JavaFX può essere la ImageView, per un'altra tipo di view può essere un altro tipo), che alla fine non fa altro che associare a una possibile posizione una immagine
Permalink https://github.com/ZhiAn-ny/OOP22-rusco-dc/blob/34222d787734cc759532118ce42845a4d9
src/main/java/it/unibo/ruscodc/view/GameViewController.java#LL51C19-L51C19
(si noti che il permalink è riferito a dove è usato il generico per questa implementazione, mentre risulta palese dove risulta l'implementazione del generico)

Uso dei Optional

Si è serviti degli Optional per aiutare la comprensibilità del codice all'interno dei comandi (altrimenti, per l'uso che ne ho fatto, andava bene anche confrontrare i risultati eventualmente anche con null). Esempio principe è la serie di controlli effettuata nell'execute del comando Interact
Permalink: https://github.com/ZhiAn-ny/OOP22-rusco-dc/blob/34222d787734cc759532118ce42845a4d9
src/main/java/it/unibo/ruscodc/model/gamecommand/playercommand/Interact.java#LL107C17-L107C17

Leonardo Marcaccio

Optionals

- lambda expressions - https://github.com/ZhiAn-ny/OOP22-rusco-dc/blob/34222d787734cc759532118ce42845a4d9b68539/src/main/java/it/unibo/ruscodc/model/actors/monster/behaviour/MovementBehaviourFactoryImpl.java#LL109C17-L116C31

Streams

- stream - https://github.com/ZhiAn-ny/OOP22-rusco-dc/blob/34222d787734cc759532118ce42845a
src/main/java/it/unibo/ruscodc/model/actors/monster/behaviour/BehaviourImpl.java#LL63C9-L73C23

Lamba expressions

- optional - https://github.com/ZhiAn-ny/OOP22-rusco-dc/blob/34222d787734cc759532118ce42845
src/main/java/it/unibo/ruscodc/model/actors/monster/behaviour/CombactBehaviourFactoryImpl.java#LL21C5-L35C23

3.3.1 Risorse esterne usate nel progetto

Per il progetto ci siamo serviti delle seguenti risorse:

- Le immagini degli oggetti sono state realizzate prendendo risorse dal sito `https://opengameart.org/`. Sono state scelte dal team e poi evenutalmente rimodellate a piacere da Pesaresi Jacopo
- Per quanto riguarda la classe Pair, ci siamo presi la libertà di copiarla dalle simulazioni per l'esame pratico e di aggiustarla per farla passare alle compilazioni di gradle

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Anny Bevilacqua

Lo sviluppo di questo progetto è stata senz'altro un'esperienza formante sia dal lato pratico, chiedendo di creare un software con un'architettura strutturata partendo da zero, sia da quello relazionale, permettendo di allenare le nostre capacità di lavorare in team. Certo, ci sono stati alcuni piccoli disguidi e incomprensioni durante la progettazione ma, tutto sommato, penso che il risultato ottenuto non sia male. Mi ritengo abbastanza soddisfatta dell'uso che ho fatto di Optionals e Streams. Penso di essere riuscita a ridurre notevolmente le porzioni di codice duplicato o ripetitivo, tuttavia, con il senno del poi, mi sarebbe piaciuto scorporare maggiormente gli Items nella stanza, togliendoli dalle Tiles e mantenendoli in una struttura separata. Chissà che non possa essere un punto d'inizio per sviluppi futuri.

Luca Bandiera

Nonostante alle superiori avessi già fatto progetti di gruppo simili a questo in cui dovevamo inventare, progettare e implementare dei software, grazie a questo progetto ho appreso ancora meglio quali sono tutte le fasi necessarie al fine di sviluppare un software completo partendo da zero.
All'inizio del progetto ho avuto alcune difficoltà su come strutturare e implementare la view del gioco, ma in seguito grazie anche al supporto del mio team sono riuscito a capire la metodologia più adatta e implementarla di conseguenza.
Sicuramente questo progetto mi ha aiutato molto a crescere sia dal punto di vista informatico, insegnandomi a gestire meglio tutte le varie fasi del progetto, sia dal punto di vista relazionale facendomi capire l'imporatanza di una buona comunicazione con il proprio team per cercare di evitare qualsiasi tipo di incomprensioni.

Concludo dicendo che sono soddisfatto di questo progetto e mi sono trovato molto bene a lavorare con il mio team.

Leonardo Marcaccio

Essendo stata questa la mia prima esperienza nello sviluppo di software, e soprattutto di sviluppo in team, l'ho trovata altamente formante e impegniativa. Una sfida che mi ha permesso di capire ancora di più ciò che ho appreso lo scorso semestre riguardo alla programmazione ad oggetti. Non nego che durante lo sviluppo io e il gruppo abbiamo dovuto affrontare diverse difficolta, ma con un po di tentativi siamo sempre riusciti a risolvere tutte leproblematiche che ci sono venute incontro. Alla fine del progetto mi sono reso conto effettivamente di quanto fosse soddisfacente vedere ciò che si è realizzato prendere forma e funzionare, eanche vedere come partendo da un timido sviluppo si arrivi a utilizzare strutture e meccanismi del linguaggio più complessi cercando di sfruttarli al proprio vantaggio. Sicuramente farò tesoro dellàesperienza ottenuta da questo progetto e cercherò di migliorare le carenze che ho riscontrato durante lo sviluppo per migliorare i miei progetti fututri.

Jacopo Pesaresi

Ho considerato questo progetto un fondamentale punto di partenza per intraprendere eventualmente una carriera come software engineer. Grazie a questa esperienza, ora mi è chiaro quali fasi seguire per sviluppare un applicativo non banale, e sopratutto quali soft skill sviluppare per garantire un ottimale sviluppo (verso fine del progetto ho dovuto rimettere mano a certe parti del codice, rovinando un pò l'architettura in partenza, in quanto non facevano effettivamente quanto pensato, problema derivante da una non sempre completa comunicazione nel team. Tuttavia, è stato particolarmente facile grazie all'uso dei desing pattern) Farò tesoro di tutti gli errori fatti durante progettazione e implementazione per un prossimo progetto, che alla luce di questa esperienza non vedo l'ora di intraprendere.
Inoltre, oltre ad aver sviluppato la mia parte di codice, essendo il membro del team con più tempo libero, sono tornato particolarmente utile per gli altri membri al fine di sincronizzare tutto il lavoro e di evolvere certe parti del loro codice, al fine di assicurare l'armonia tra le varie parti del codice. In primis, sono colui che ha proposto come strutturare il "game-flow" del gioco.
Infine concludo che, nonostante le difficoltà incontrate e le modifiche fatte all'ultimo che, in mia opinione, hanno lievemente rovinato la progettazione delle mie classi, sono complessivamente soddisfatto di quanto sviluppato.

4.2 Difficoltà incontrate e commenti per i docenti

È stata svolta la fase di analisi in tempo utile e correttamente, non sono state incontrate particolari difficoltà, tuttavia è risultato particolarmente complesso definire subito una architettura compiuta di tutto il gioco. Fattori che indiscutibilmente hanno generato questo problema sono stati l'inesperienza globale nel team (principalmente per quanto riguarda quelli che hanno concluso il corso a dicembre 2022) e la difficoltà di trovare un periodo che potesse andare bene a tutti i quattro i membri del team (riuscivamo solo ad esserci quasi sicuramente 2 pause pranzo universitarie su 5).
Quindi si è provato, subito dopo aver finito la fase di analisi, a crearla senza definire esattamente tutte le relazioni tra le classi. Ma senza un accordamento robusto di una serie di interfacce condivise dopo un pò sono state incontrate delle difficoltà che abbiamo mano a mano affrontato, pensato, concluso una soluzione che portava a una riprogettazione di certe classi e che generava altre problematiche, creando un circolo virtuoso, finchè non si è raggiunta una situazione di stasi

Appendice A

Guida utente

Il gioco inizia subito con una schermata di inizio, molto autoesplicativa, generalmente gestita con il mouse (ma è sufficente premere invio). Passando quindi subito al gioco in sè, questo è invece gestito prevalmentemente da tastiera. In particolare:

- WASD : classico movimento dei giochi da tastiera (quindi W per andare in alto, S in basso, A per sinistra e D per destra)
- ESC : serve per annullare una interazione giocatore-gioco complessa (sia una interazione di attacco, di inventario, ecc)
- F : comando di interazione. Crea un'area grigia attorno all'eroe su cui posizionare il cursore. In questa situazione, premere C permette di stamapre a video delle informazioni relative a quella cella (o un messaggio di errore se il cursore è oltre all'area di interazione o se si chiede l'informazione di una cella vuota). Ha senso interagire con porte, scale, chest (scatoloni), drop (sacchi spazzatura) e con i mostri (Rusco non ha la vista aguzza, quindi per capire come sta il suo avversario deve aspettare che si avvicini!)
- 1 2 3 4 5 : comandi che creano una situazione di attacco (in particolare 1 è l'attacco base, mentre gli altri sono abilità speciali). Quando premuti generano l'area range, l'area splash e il cursore. Affinchè un comando di attacco vada a buon termine bisogna premere C (Conferma) quando il cursore si trova sopra a una cella grigia (che è l'area di range), mentre durante la fase di preparazione quella arancione mostra chi è compreso nell'attacco. Notare che se il cursore esce dall'area grigia, quella arancione non viene più aggiornata.
  Di base, è da considerare che gli attacchi sono perforanti (se due mostri sono in linea ma entrambi sono compresi nell'area arancione allora entrambi subiranno lo stesso effetto), e non è compreso il "fuoco amico" (Rusco è immune ai suoi attacchi, poi ovviamente se si hanno dei mostri che lo possono bersagliare e il comando viene confermato con il mirino sopra Rusco,

i danni ricevuti non sono quelli del suo attacco ma quelli dei mostri!)
Notare come il comando 4 in realtà è un comando di "recupera salute", quindi non fa altro che consumare AP per guadagnare HP Per più informazioni aprire il menù di inventario

- I : apertura dell'inventario. La griglia di gioco viene rimpiazziata da una serie di oggetti che permettono di modificare le statistiche, che vengono riportate in una label a fianco.
  Per sapere più informazioni su un oggetto è sufficente posizionare il cursore sopra quell'oggetto e quindi ripremere I - Informazioni: verranno stampate a video delle informazioni.
  Per poterne usufruire è sufficente premere C e, se l'oggetto era consumabile, questo sparisce dall'inventario e modifica la stat secondo quanto descritto, mentre se era equipaggiabile allora questo viene equipaggiato. Se si equipaggia un pezzo di armatura A dopo aver già equipaggiato un pezzo di armatura B, e sono entrambi dello stesso tipo (es due elmetti), allora Rusco equipaggia A, ma dis-equipaggia B, rimettendolo nell'inventario e modificando le stats di conseguenza.
  Nel caso non si desideri avere l'inventario affollato di oggetti, è possibile scartarli con lo BACKSPACE.
  Nel caso in cui l'inventario sia vuoto, viene comunque stampato l'inventario con a fianco le stat di Rusco, ma alla prossima pressione di pulsante verrà chiuso. Se si desidera chiuderlo anticipatamente, è sufficente usare ESC

- qualsiasi altro pulsante di tastiera non citato : se Rusco stava aspettando una nuova interazione, verrà intepretato come comando di "passa turno", mentre se stava elaborando un attacco, una interazione o stava consultando l'inventario allora semplicemente la pressione del pulsante viene ignorata

Interazioni non prettamente di tastiera sono:

- il bottone di ritorno al menù principale

- un bottone flag che assicura di non premere accidentalmente il bottone precedente se non desiderato

In caso di game over (o di game win), si apre invece una schermata di game over che permette di tornare al menù principale.

Appendice B

Esercitazioni di laboratorio

B.0.1 luca.bandiera3@studio.unibo.it

- Laboratorio 03: https://virtuale.unibo.it/mod/forum/discuss.php?d=112846#p168217
- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p170047
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=117044#p173235
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=117852#p174191
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=118995#p175371
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=119938#p175872

B.0.2 Leonardo Marcaccio

- Laboratorio 03: https://virtuale.unibo.it/mod/forum/discuss.php?d=112846#p168157
- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p169596
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=115548#p171217
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=117044#p172673

- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=117852#p173910
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=118995#p174736
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=119938#p176182
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=121130#p177388
- Laboratorio 12: https://virtuale.unibo.it/mod/forum/discuss.php?d=121885#p178161

B.0.3 jacopo.pesaresi2@studio.unibo.i

- Laboratorio 03: https://virtuale.unibo.it/mod/forum/discuss.php?d=112846#p168212
- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p169900
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=115548#p171903
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=117044#p172961
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=117852#p173878
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=118995#p175042
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=119938#p175884