ROGUE

Pietro Pezzi, Manuel Quarneti, Fabio Chiarini, Luca Tassinari

23 febbraio 2021

Indice

1 Analisi **3**
- 1.1 Requisiti . . . 3
- 1.2 Analisi e modello del dominio . . . 4

2 Design **6**
- 2.1 Architettura . . . 6
- 2.2 Design dettagliato . . . 8
  - 2.2.1 Manuel Quarneti . . . 8
  - 2.2.2 Fabio Chiarini . . . 11
  - 2.2.3 Pietro Pezzi . . . 13
  - 2.2.4 Luca Tassinari . . . 17

3 Sviluppo **23**
- 3.1 Testing automatizzato . . . 23
- 3.2 Metodologia di lavoro . . . 24
  - 3.2.1 Manuel Quarneti . . . 24
  - 3.2.2 Fabio Chiarini . . . 25
  - 3.2.3 Pietro Pezzi . . . 25
  - 3.2.4 Luca Tassinari . . . 26
- 3.3 Note di sviluppo . . . 26
  - 3.3.1 Manuel Quarneti . . . 26
  - 3.3.2 Fabio Chiarini . . . 26
  - 3.3.3 Pietro Pezzi . . . 27
  - 3.3.4 Luca Tassinari . . . 27

4 Commenti finali **28**
- 4.1 Autovalutazione e lavori futuri . . . 28
  - 4.1.1 Manuel Quarneti . . . 28
  - 4.1.2 Fabio Chiarini . . . 28
  - 4.1.3 Pietro Pezzi . . . 29
  - 4.1.4 Luca Tassinari . . . 29

A Guida Utente **30**
A.0.1 Utilizzo inventario 32
A.0.2 Movimento, attacco e raccolta oggetti 32
A.0.3 Passaggio al livello successivo 33

B Esercitazioni di laboratorio **34**
B.0.1 Luca Tassinari 34

Capitolo 1

Analisi

1.1 Requisiti

Il progetto mira alla creazione di un videogioco ispirato al celeberrimo gioco Rogue del 1980 in una moderna veste grafica. Il gioco inizia al livello più alto di un mondo in cui sono disseminati mostri e tesori. Lo scopo consiste nell'ottenere più tesori possibile, sopravvivendo agli attacchi dei mostri e alle insidie che si incontrano durante il viaggio.

Requisiti funzionali

- Il gioco e ogni sua azione è scandita dall'input dell'utente a cui corrisponde un turno.
- Il mondo è costituito da un insieme infinito di livelli.
- La mappa di ogni livello è generata casualmente, ivi compresi i suoi elementi costitutivi.
- All'interno della mappa il giocatore si muove controllato dall'utente attraverso le frecce direzionali.
- All'interno di ciascun piano vengono generati in maniera pseudocasuale vari oggetti che possono essere usati dal giocatore per agevolarsi durante l'esplorazione del sotterraneo.
- Gli oggetti raccolti dal giocatore potranno essere conservati, usati o rimossi all'interno del proprio inventario.
- Durante l'esplorazione il giocatore viene inseguito da mostri di varia natura (più o meno ostili), che vengono generati in base al livello del giocatore.

- Il giocatore entrando in contatto con un'entità nemica ingaggia un combattimento con l'avversario basato su una serie di attacchi e schivate.
- Il gioco termina non appena il giocatore ha consumato tutto il suo livello di *starvation* oppure tutti i suoi punti vita.

Requisiti non funzionali

- L'applicazione deve rispondere all'input dell'utente in maniera fluida.

1.2 Analisi e modello del dominio

Il fulcro del gioco è un **mondo** costituito da **livelli** sui quali vengono posizionate le varie **entità** che lo popolano. Queste ultime sono suddivisibili in due macro categorie: le **creature** e gli **oggetti**.

Le creature sono entità che, a differenza degli oggetti, si muovono all'interno della mappa e posseggono una **vita**. In particolare, i due sottotipi di creatura sono il **giocatore** e i **mostri**. I mostri possiedono varie caratteristiche che vengono usate durante il **combattimento** e alcuni di essi possiedono delle abilità **speciali**. Il giocatore possiede un **equipaggiamento** e l'**inventario**, un contenitore interattivo di oggetti, accomunati dal fatto che possono essere usati sul player attraverso l'inventario.

Gli elementi costitutivi del problema sono sintetizzati in Figura 1.1.

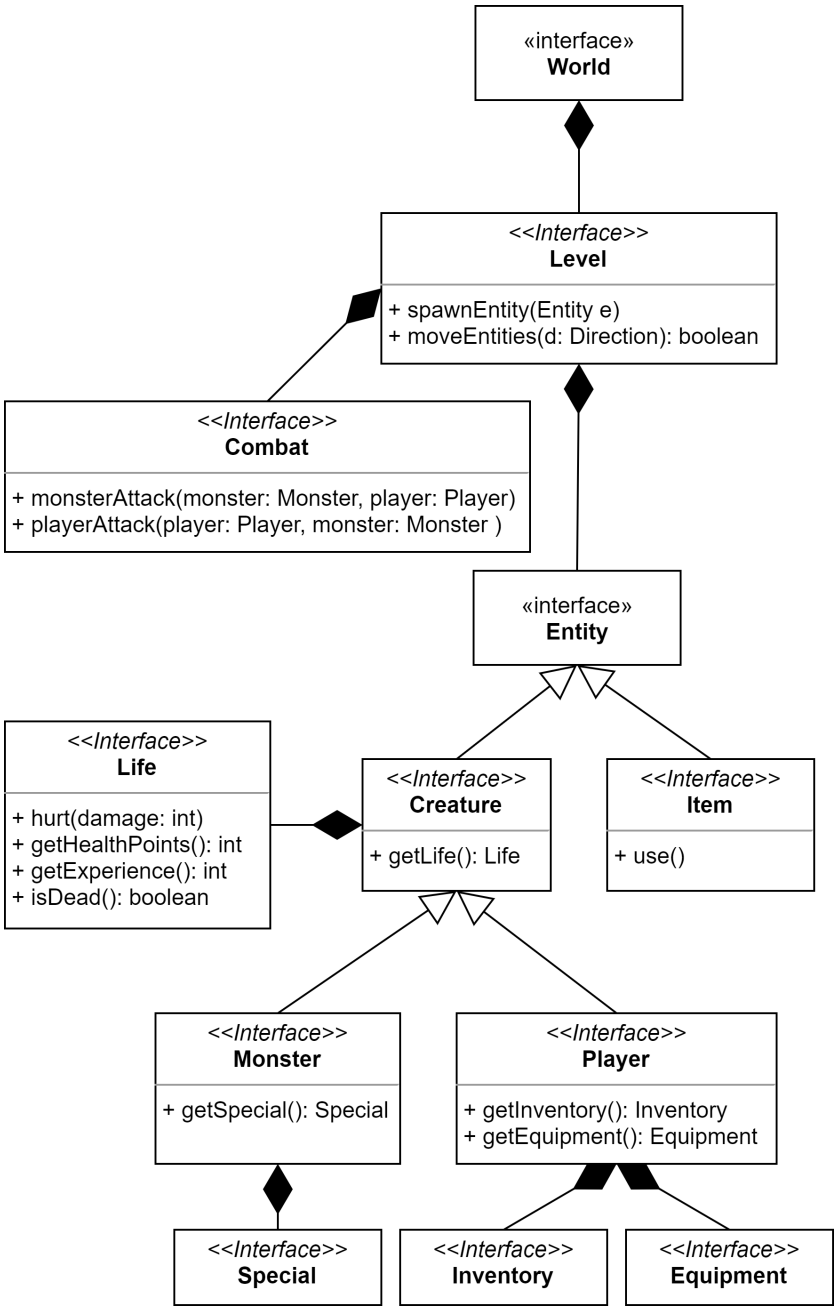


Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Il gioco è sviluppato utilizzando il pattern architetturale MVC. Il model dell'applicazione gestisce la logica delle varie entità che costituiscono il dominio applicativo e che sono già state precedentemente descritte nell'analisi e modello del dominio.

L'entry point dell'applicazione è la classe `App` che si occupa di richiamare il main della classe `Launcher`, la quale genera la View del Menu principale e il relativo `GameController`. Quest'ultimo gestisce la creazione dell'istanza del `Player` e dei tre principali controller, i quali coordinano le rispettive parti di model e view con un'unico riferimento al suddetto `Player`.

In Figura 2.1 vengono mostrate le principali interazioni tra le parti del modello e del controller.

Qui di seguito elenchiamo più specificatamente le singole responsabilità:

- `StatusBarController`: si occupa di aggiornare la relativa view non appena lo stato del player cambia;
- `InventoryController`: gestisce le operazioni eseguite dall'utente sull'inventario andando ad aggiornarne il modello;
- `WorldController`: decodifica il tasto premuto dall'utente e di conseguenza avvia un round del mondo.

La `GameView` si occupa di generare le tre View principali dell'applicazione: la `WorldBox` renderizza la mappa, la `InventoryView` visualizza i vari item presenti nell'inventario e la `StatusBarView` rappresenta la vita e le statistiche del giocatore.

In Figura 2.2 sono esemplificate le principali interazioni tra le parti del controller e relative view.

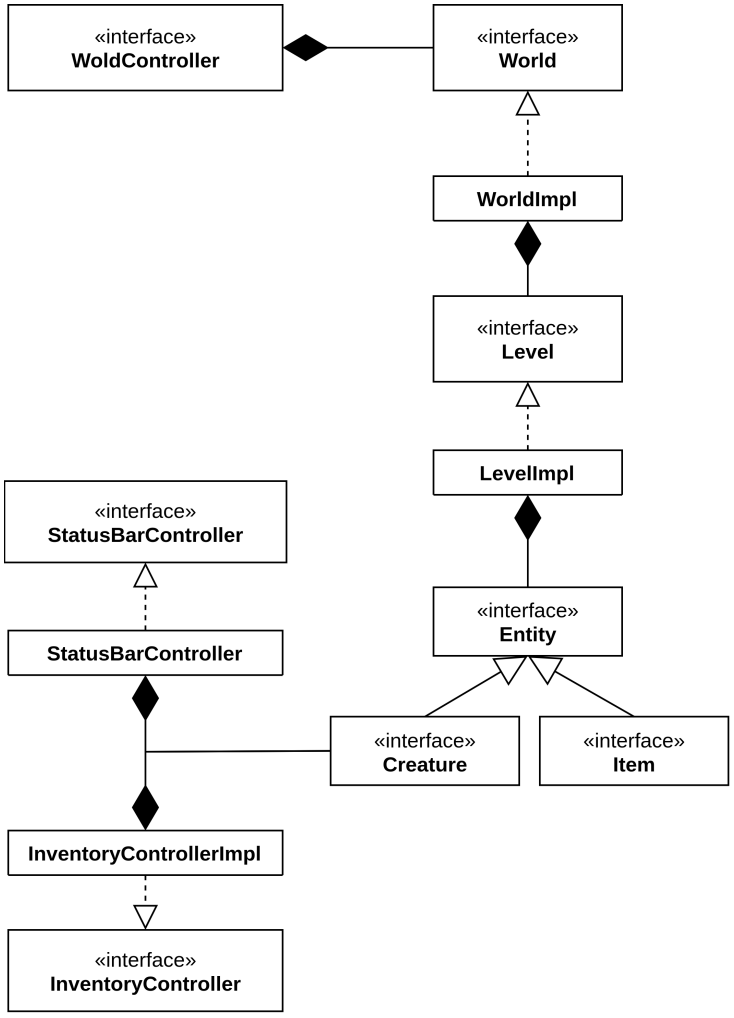


Figura 2.1: Schema UML architetturale di Rogue e delle interazioni model-controller.

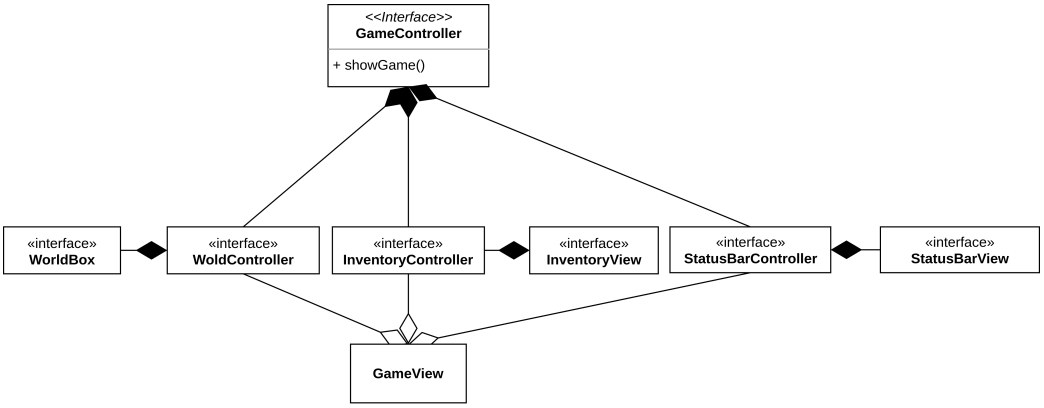


Figura 2.2: Schema UML architetturale di Rogue e delle interazioni controller-view.

2.2 Design dettagliato

2.2.1 Manuel Quarneti

Io mi sono occupato della creazione delle classi riguardanti la creazione e gestione del mondo di gioco e delle varie entità in esso.

Model

- **World** gestisce i vari **Level**, generandone di nuovi e eliminando i vecchi
- **Level** gestisce i **Tile** e le **Entity** e si occupa del movimento delle **Entity** sui **Tile**
- **CaveGenerator** si occupa di generare una matrice di booleani usata per determinare la posizione dei muri nel **Level**
- I **Tile** sono i blocchi che costituiscono la parte "fissa" del **Level**
- Le **Entity** sono qualsiasi cosa che si può muovere o rimuovere e sono sempre associate ad un **Tile**

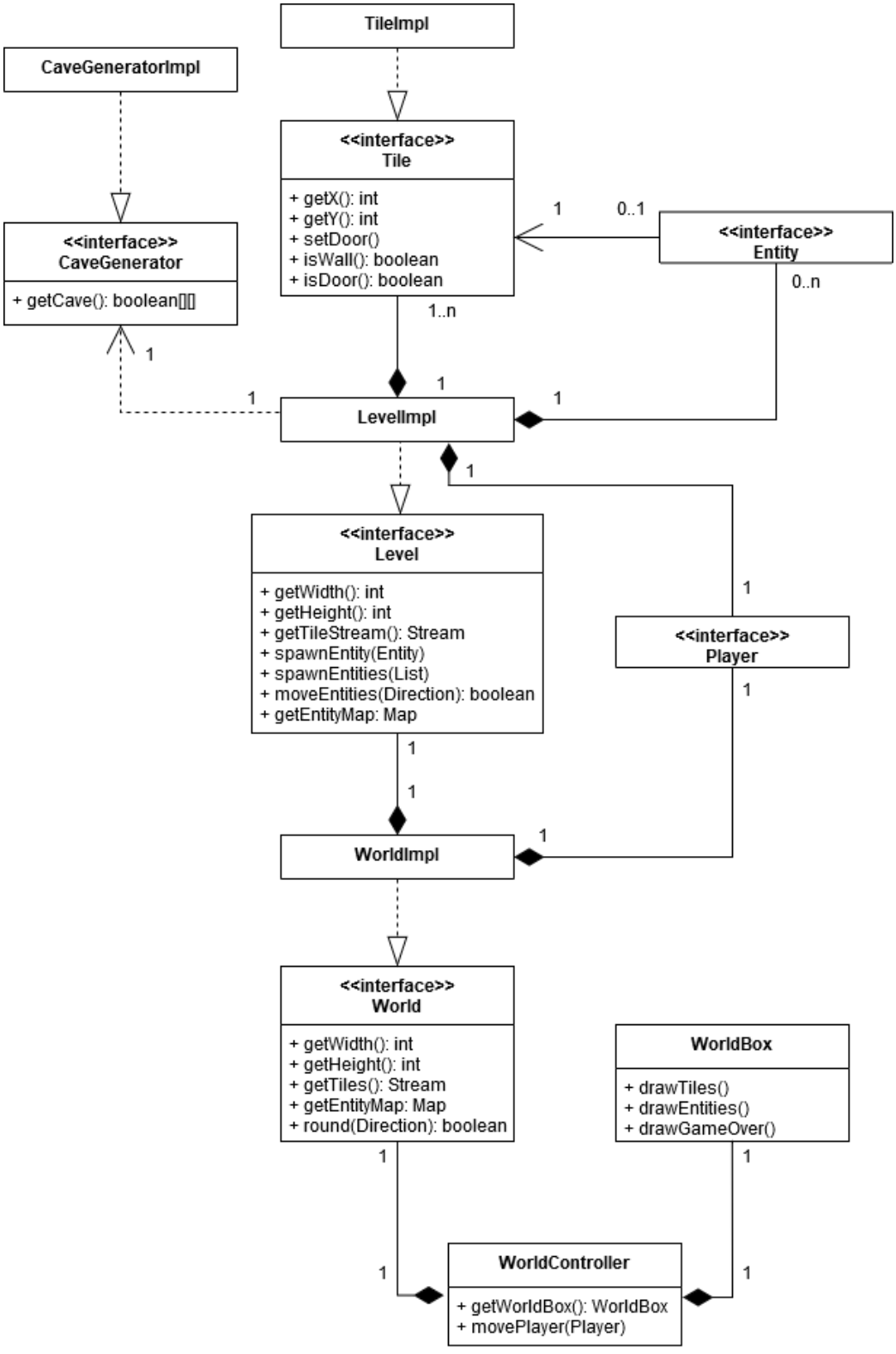
View

- **WorldBox** estende **HBox** ed implementa i metodi necessari al rendering della mappa e delle entità. In più ha anche un metodo per la visualizzazione di una schermata di "GAME OVER".

Control

- **WorldController** facilita la coordinazione fra il **World** (model) e la **WorldBox** (view).

Non è menzionato nel seguente UML, ma esiste un enum contenente le direzioni cardinali (**Direction**)

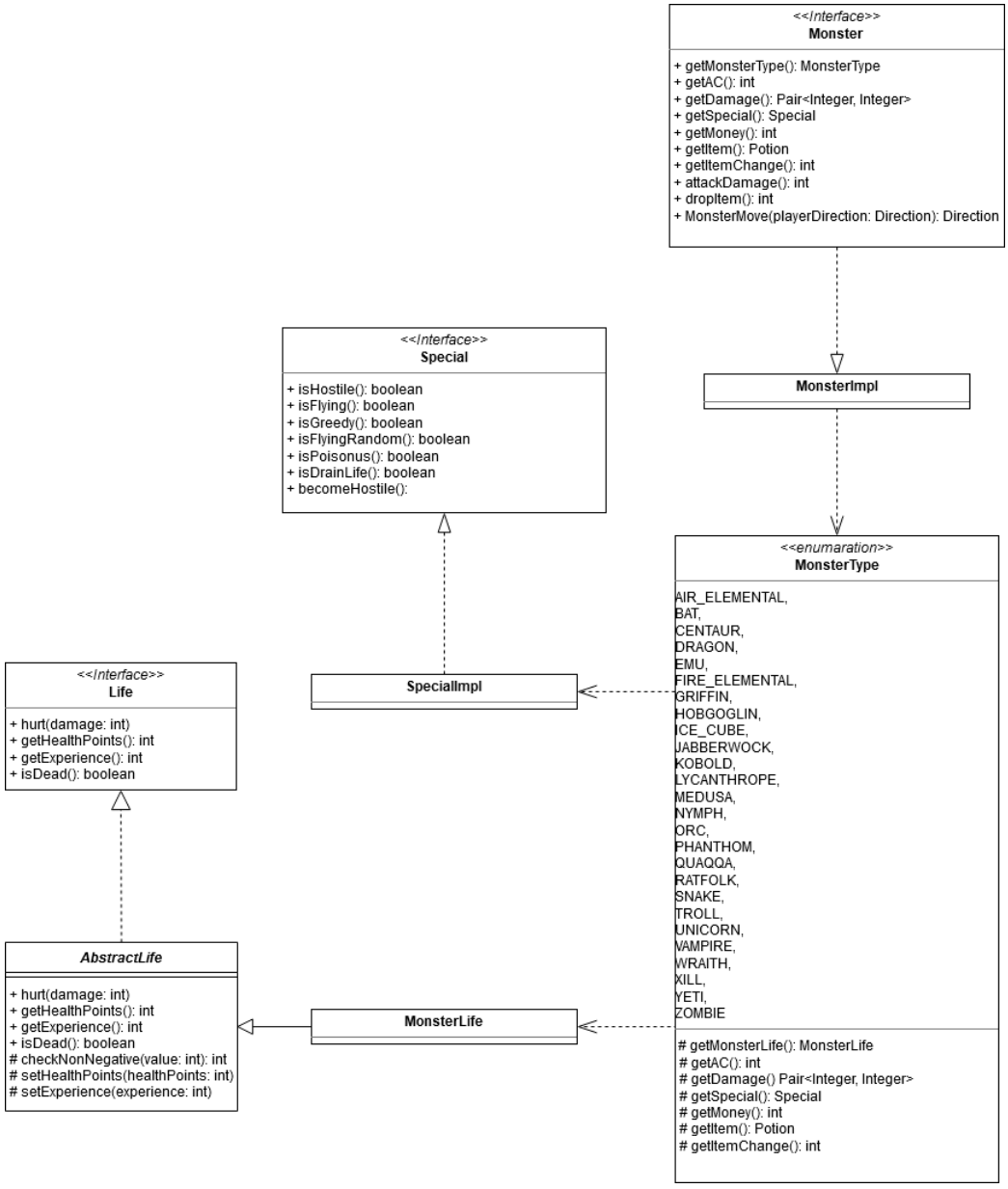

TileImpl
CaveGeneratorImpl
<<interface>>
Tile
+ getX(): int
+ getY(): int
+ setDoor()
+ isWall(): boolean
+ isDoor(): boolean
<<interface>>
Entity
<<interface>>
CaveGenerator
+ getCave(): boolean[][]
LevelImpl
<<interface>>
Level
+ getWidth(): int
+ getHeight(): int
+ getTileStream(): Stream
+ spawnEntity(Entity)
+ spawnEntities(List)
+ moveEntities(Direction): boolean
+ getEntityMap: Map
<<interface>>
Player
WorldImpl
<<interface>>
World
+ getWidth(): int
+ getHeight(): int
+ getTiles(): Stream
+ getEntityMap: Map
+ round(Direction): boolean
WorldBox
+ drawTiles()
+ drawEntities()
+ drawGameOver()
WorldController
+ getWorldBox(): WorldBox
+ movePlayer(Player)
1
0..1
0..n
1..n

2.2.2 Fabio Chiarini

Il mio obiettivo era quello di creare i vari mostri che avrebbero dovuto successivamente popolare il sotterraneo e di implementare il combattimento tra il giocatore e i vari nemici.

Monster

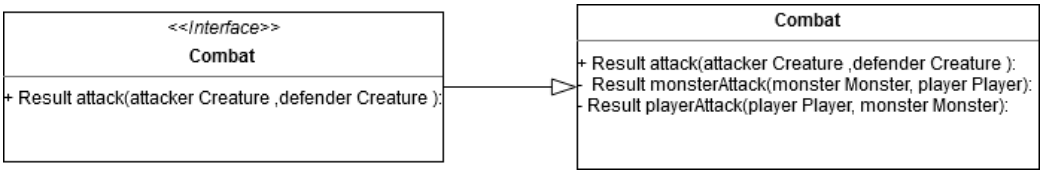
Sia i mostri che il giocatore hanno una **Life** (creata da Luca Tassinari) che poi nel caso del mostro si espande in **MonsterLife**. Esistono diversi nemici all'interno del gioco ogni creatura viene definita in una enumerazione che ne definisce le varie caratteristiche. I vari attributi comportano sia il movimento all'interno della mappa sia la loro pericolosità durante il combattimento. Il mostro possiede anche una serie di oggetti che rilascia quando viene sconfitto, quello di maggior importanza è l'item, ovvero una **Potion** che in base ad una percentuale, viene rilasciata o no una volta finito il combattimento. Ogni volta che viene creato un nuovo **Level** viene richiamato **MonsterFactory** che in base al livello del protagonista genera una lista di mostri variabile. Inoltre mi sono anche occupato delle realizzazione della **MonsterImageGenerator** che si occupa di associare ad ogni nemico una relativa immagine.



Combat

Il combattimento si basa sul far scontrare due **Creature** una prenderà il ruolo di attaccante mentre l'altra sarà il difensore. Il combattimento si basa sul prendere le varie informazione da entrambe le entità in gioco e di metterle a confronto, facendo anche in modo che il duello sia in parte casuale considerando il fatto che esiste una percentuale per colpire e un range per il danno. Per facilitarne l'uso **Combat** comprende solo un metodo pubblico

che in base a chi è l'attaccante e chi è il difensore si occupa lui di richiamare i metodi privati interni per far attaccare il mostro o il giocatore.



2.2.3 Pietro Pezzi

Questa sezione si concentrerà sugli aspetti degli Oggetti consumabili, Contenitore oggetti e sulla GUI Principale.

Item

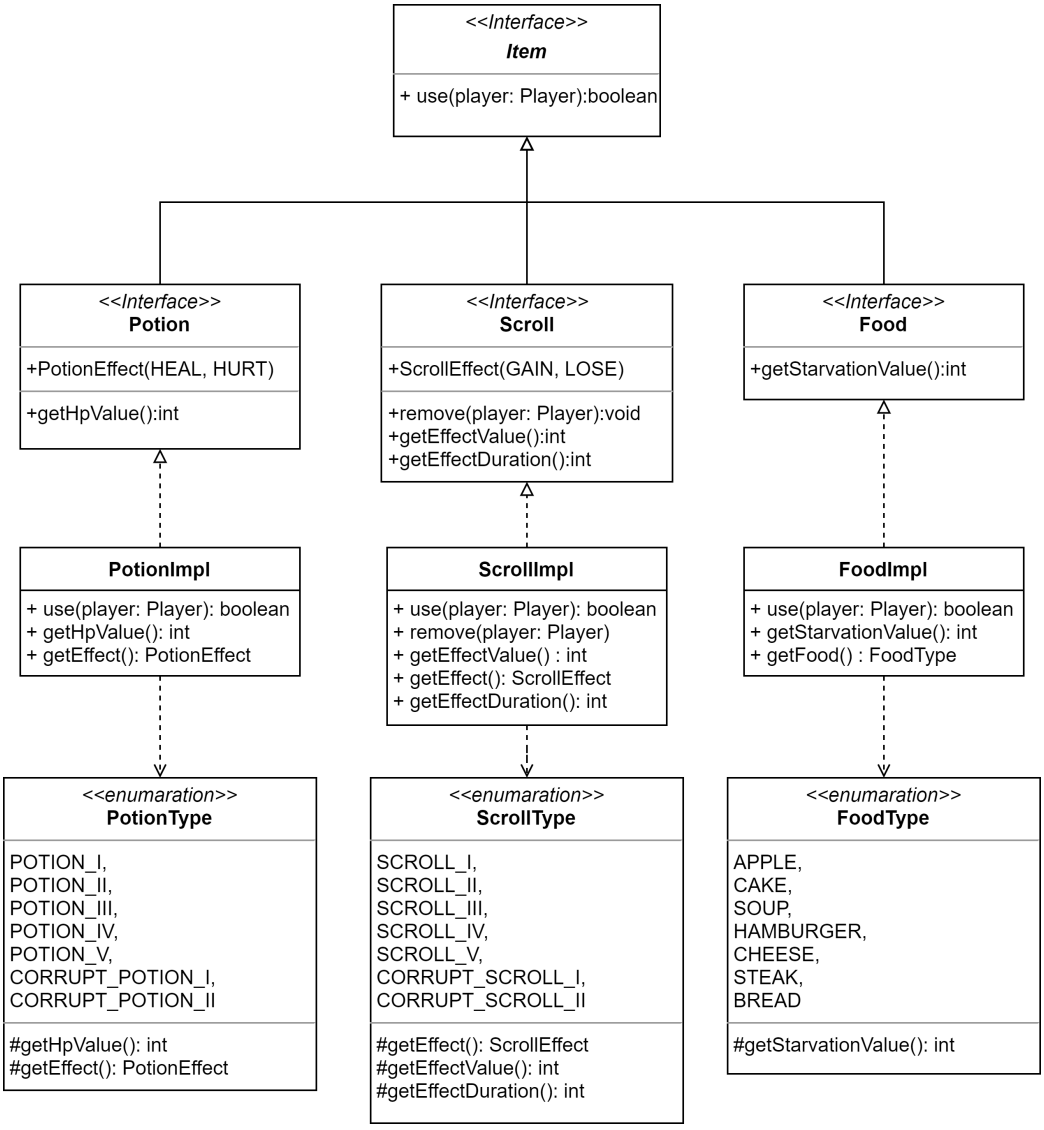
Per quanto riguarda gli oggetti consumabili, dato che sia io che Luca Tassinari avevamo il compito di creare degli oggetti, abbiamo creato una interfaccia **Item** che raggruppa tutti gli oggetti all'interno del gioco assegnando ad ognuno un metodo **use** che sarà poi richiamato dall'inventario.

Gli oggetti Consumabili all'interno del gioco sono le pozioni, le pergamene e il cibo. Ad ognuno di questi oggetti è assegnata un'interfaccia, nel caso delle pergamene e delle pozioni,queste sono accompagnate da una enumerazione che ne definisce l'effetto.

Ogni implementazione degli Oggetti consumabili offre il metodo `use` e diversi metodi per ottenere le varie caratteristiche di ogni oggetto consumabile, ad esempio nel caso delle pozioni abbiamo il metodo **getHpValue** che ritorna la quantità di vita da aggiungere o togliere al giocatore.

Ad ogni implementazione degli oggetti consumabili è assegnata una enumerazione che elenca tutti gli oggetti, questa enumerazione si occuperà, nel caso delle pozioni e delle pergamene, anche di generare un numero casuale di vita o forza da aggiungere o togliere al giocatore.

Nella figura seguente è possibile vedere l'implementazione degli oggetti consumabili più nel dettaglio.

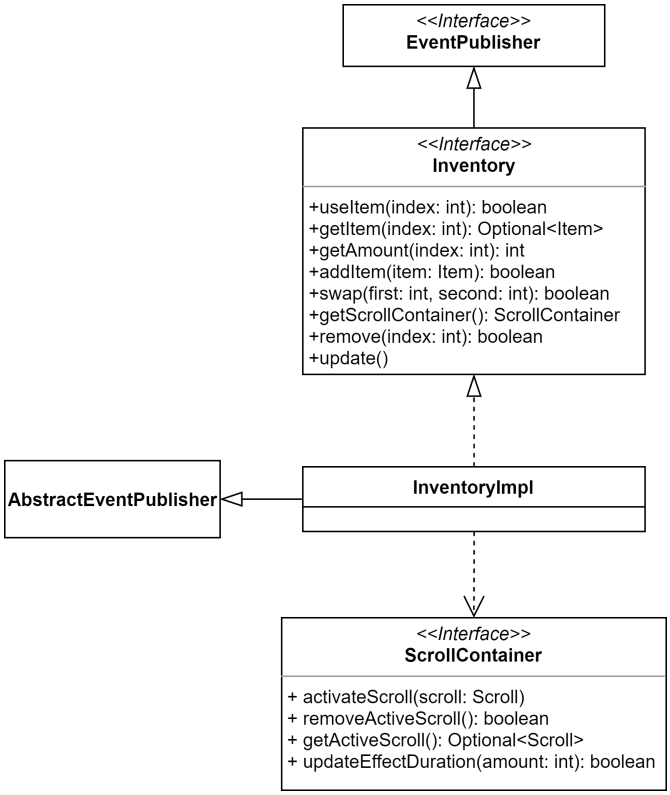


Mi sono anche occupato, collaborando con Luca Tassinari, alla realizzazione della **ItemFactory**, che si occupa di generare una lista contenente un determinato numero di Item casuali, ed alla realizzazione , collaborando con Fabio Chiarini, della **ItemImageGenerator**, che si occupa invece di restituire l'immagine corrispondente di un determinato Item.

Inventario

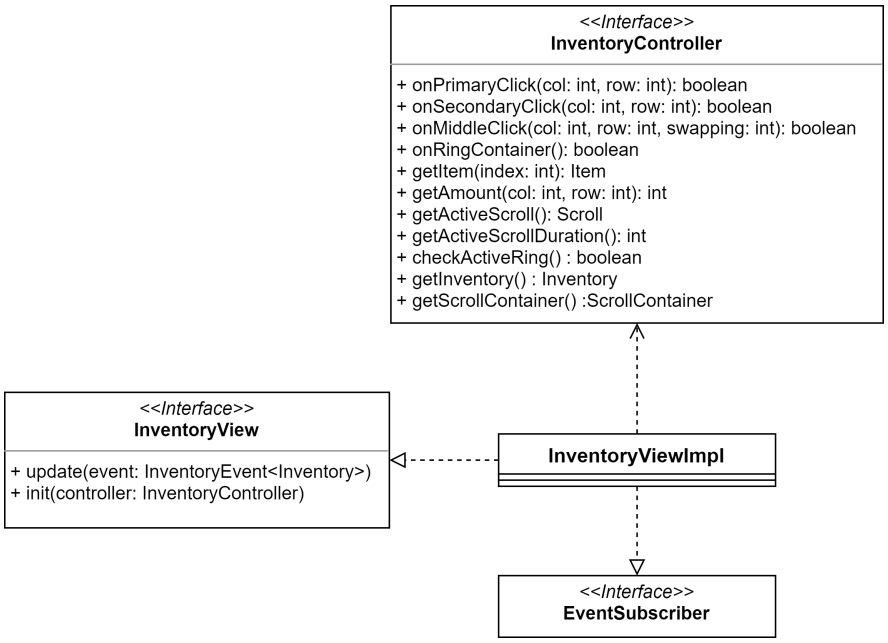
Per gestire i contenitori di oggetti all'interno del gioco, dopo aver discusso con il mio gruppo, ho deciso di implementare un unico contenitore, ovvero l'`inventario` del giocatore. Dato che nel progetto abbiamo implementato

il pattern MVC, anche l'inventario è stato suddiviso in una parte di model, view e di control. Il model è implementato con un contenitore interagibile di item, in modo da consentire l'eventuale aggiunta di nuovi oggetti al gioco. Questo contenitore consente al giocatore di collezionare,utilizzare e rimuovere oggetti, inoltre permette di gestire l'attivazione e disattivazione delle pergamene. L'inventario è anche organizzabile attraverso lo spostamento dei vari oggetti. Vengono esposti in modo più dettagliato i compiti dell'inventario nel seguente UML.



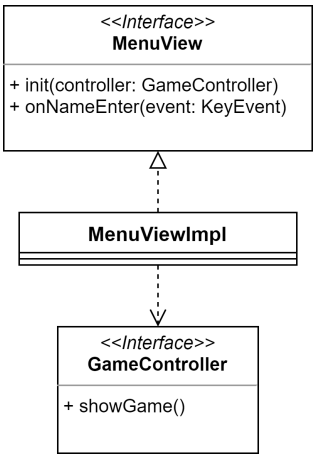
Per quanto riguarda la view dell'inventario, questa si presenta all'utente come una griglia di contenitori per oggetti, più due contenitori rispettivamente per gli anelli e pergamene attive. Ad ogni azione eseguita sull'inventario, la view andrà a richiamare il controller, il quale aggiorna lo stato del contenitore e in corrispondenza dell'utilizzo di un oggetto, lo andrà ad applicare sul giocatore. L'aggiornamento della view del contenitore è stata gestita utilizzando la libreria **EventBus**, in particolare con un evento dedicato all'inventario chiamato **InventoryEvent**. Ad ogni operazione eseguita dall'utente sull'inventario il controller andrà a rinnovare il contenitore e la corrispondente view richiamando il metodo `update` contenuto nell'`EventHandler` per l'inventario, ovvero la **InventoryView**. Il controller si occupa anche di fornire alla view

gli eventuali dati da presentare al giocatore sul display. Nella seguente figura viene mostrato il funzionamento della parte View-Control dell'inventario.



GUI Principale

Riguardo alle GUI, mi sono occupato della realizzazione del menu principale e del suo relativo controller. Il **Menu** principale si occupa solamente di chiedere all'utente di inserire un nome, una volta che un nome valido sarà fornito il Menu richiamerà il **GameController** che si occuperà di costruire i principali elementi che andranno a formare il gioco. Collaborando con il gruppo, abbiamo costruito la **GameView**, cioè l'insieme delle varie view per completare il gioco.



2.2.4 Luca Tassinari

In questa sezione verrà trattato il design lato model del player e delle sue correlate caratteristiche, quali vita ed equipaggiamento, nonché il controller e la view della barra di stato in cui vengono rappresentate le principali informazioni sul player.

Player e Life

Per quanto concerne la struttura del player e i suoi caratteri distintivi (in particolare la sua vita, trattata successivamente), vista la forte comunanza e tuttavia non totale coincidenza di elementi con i mostri, si è optato per un design che permettesse il massimo riuso degli elementi comuni, aderendo ad uno schema interfaccia - classe astratta - classe concreta. In Figura 2.3 viene mostrato uno schema esemplificativo del design comune delle creature e del player.

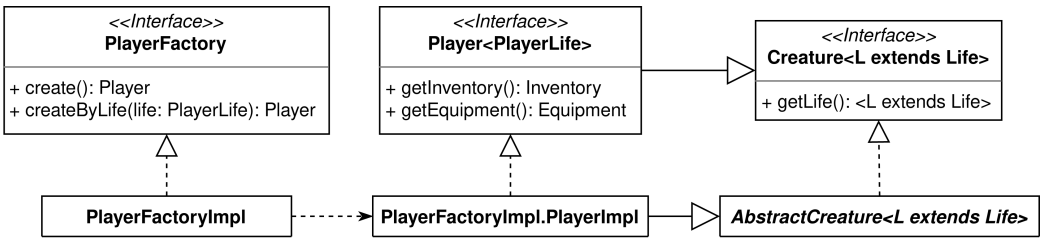


Figura 2.3: Schema UML del design del player e del pattern Factory Method per la sua creazione.

Il pattern `Factory Method` è stato qui utilizzato per permettere che all'atto della creazione del `Player` possa essergli fornita anche un'istanza di `PlayerLife` con la quale inizializzare la sua vita, aspetto chiaramente importante per l'ipotetico sviluppo del salvataggio della partita. Qualora invece si voglia creare un'istanza del player con i parametri di vita predefiniti, la creazione della stessa viene effettuata dal metodo della factory in modo del tutto trasparente al client.

Come si può notare, il principale punto di contatto tra le due creature (mostri e player) è sicuramente la vita, reificata attraverso un'interfaccia `Life` e una sua implementazione astratta, la quale consolida il codice riusabile tra le varie implementazioni concrete, `PlayerLife` e `MonsterLife`, che la estendono per specializzarla opportunamente.

Come si può evincere dallo schema UML presentato in Figura 2.4, si è separata la costruzione dell'oggetto `PlayerLife` dalla sua effettiva rappresentazione attraverso il pattern **Builder** principalmente per due motivi:

- Vista la modesta quantità di parametri presenti nella classe, e tenuto contro che la maggior parte di essi hanno lo stesso tipo, se fosse stato fornito un singolo costruttore, il codice sarebbe risultato opaco;
- Ogni parametro ha un valore di *default* e, anche in prospettiva dell'aggiunta del salvataggio della partita di gioco, è importante che si abbia la possibilità di impostare in maniera non predefinita le statistiche della vita del player con cui avviare il game play.

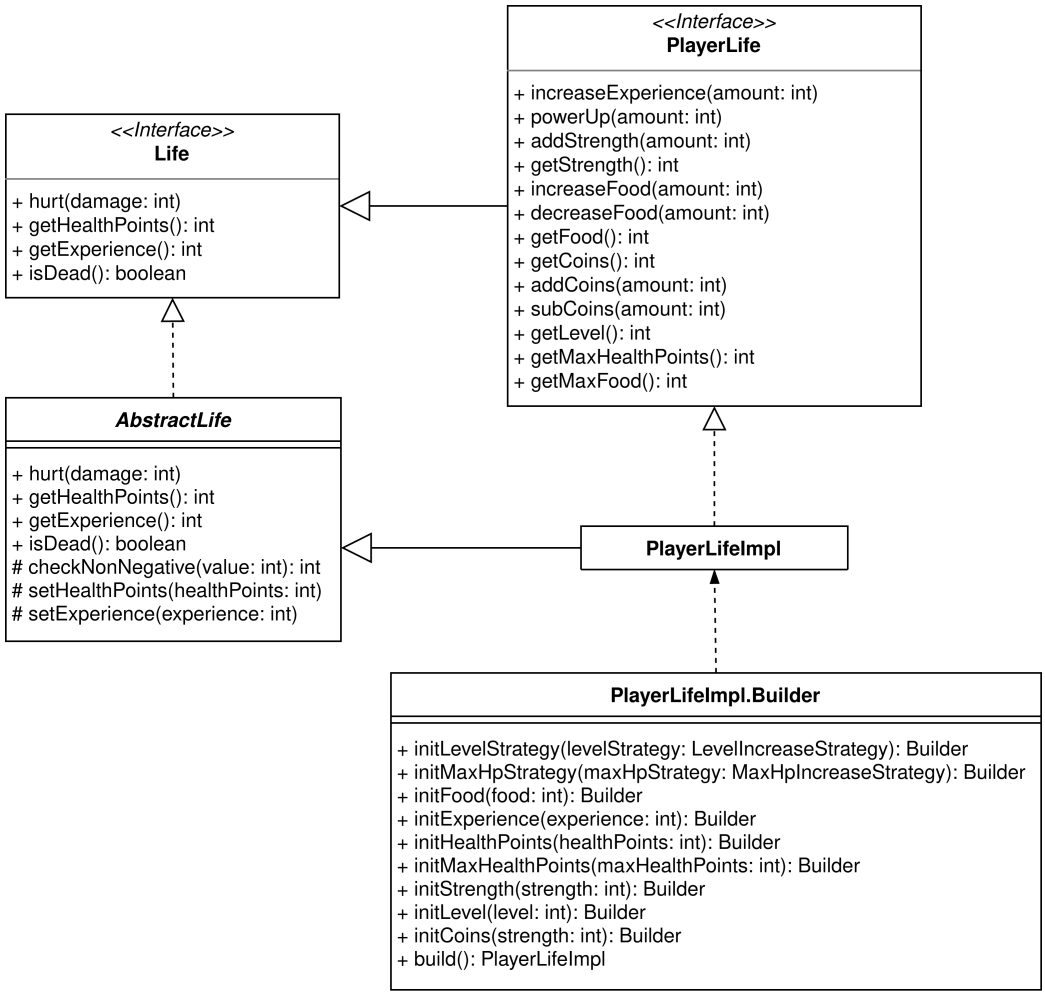


Figura 2.4: Schema UML del pattern Builder per `PlayerLife`.

Per quanto riguarda la gestione e l'incremento del livello e dei punti vita massimi accumulabili in funzione, rispettivamente, dell'esperienza accumulata e del livello stesso del player si è usato il pattern **Strategy** in cui le due strategie sono rappresentate, come mostrato in Figura 2.5, dalle interfacce `LevelIncreaseStrategy` e `MaxHpIncreaseStrategy`. In questo modo la classe `PlayerLifeImpl` ha un'unica responsabilità, quella di gestire i vari parametri della vita del player, demandando la concreta strategia di incremento del livello o della massima vita accumulabile alle rispettive `StandardLevelIncreaseStrategy` e `StandardMaxHpIncreaseStrategy`, aderendo al *Single Responsability Principle*. Inoltre questo approccio permette, laddove ve ne fosse l'esigenza, di aggiungere una diversa strategia senza dover modificare la classe, ma semplicemente implementandone una nuova, in accordo con l'*Open Closed Principle*.

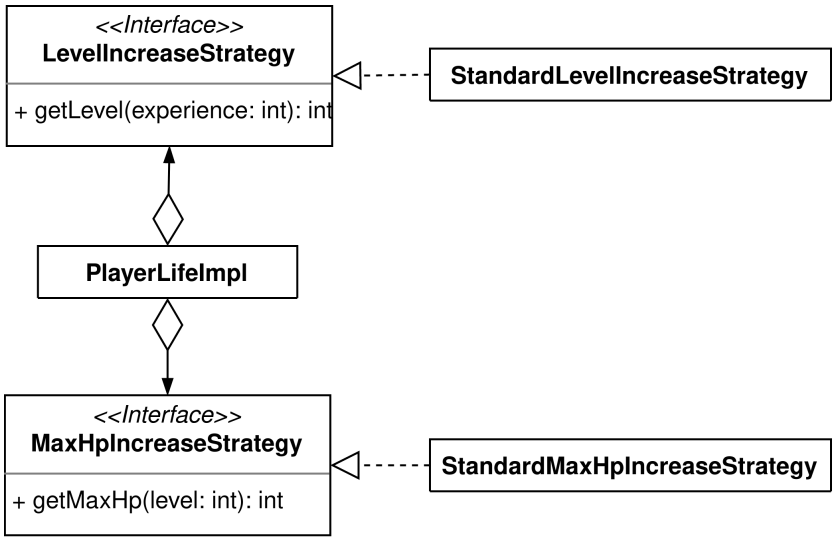


Figura 2.5: Schema UML del pattern Strategy nella classe `PlayerLifeImpl`.

Item e Equipaggiamento

In Figura 2.6 viene mostrato un UML complessivo che illustra la struttura generale degli item che sono alla base dell'equipaggiamento del player. Tutte e tre le tipologie, `Ring`, `Armor` e `Weapon`, reificate in oppurtune interfacce, estendono `Item` che definisce il contratto d'uso per tutti gli item del gioco. In particolare, come mostrato in Figura 2.7, si è usato il pattern **Decorator** per consentire di migliorare la precisione o l'accuratezza dell'arma dinamicamente durante il gioco. Nel caso specifico la classe astratta che funge da decoratore è `WeaponDecorator` e le sue concrete implementazioni sono la

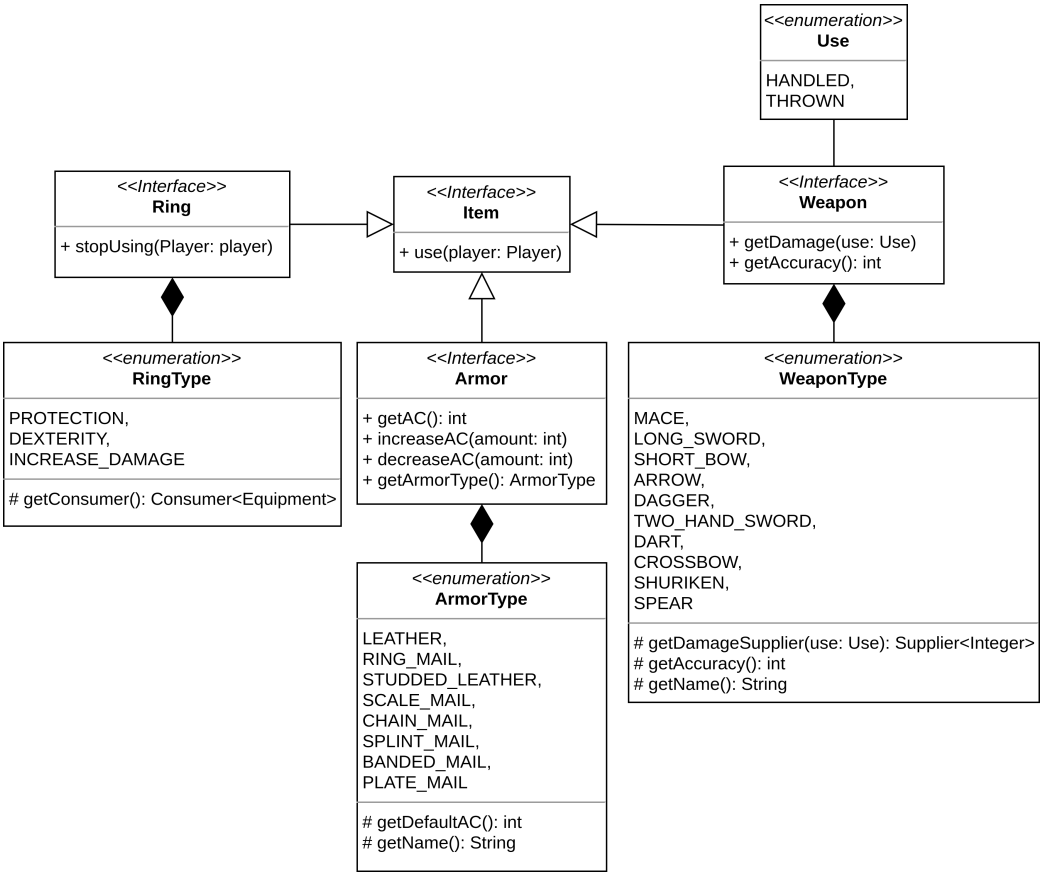


Figura 2.6: Schema UML della struttura generale degli item che compongono l'equipaggiamento del player.

`IncreaseDamage` e `IncreaseAccuracy`. Attraverso questo design, laddove necessario (si veda, ad esempio, la sezione relativa ai `Ring`), è possibile cambiare "pelle" all'arma, in modo tale che le sue caratteristiche base (danno e accuratezza) vengano opportunamente modificate a seconda del decoratore impiegato. Questa soluzione dunque permette, non solo di organizzare in modo selettivo le varie componenti, ma anche di poter aggiungere nuove decorazioni o varianti delle armi con facilità.

In merito all'equipaggiamento e ai ring è stato impiegato il pattern **Memento**. Infatti, i ring sono oggetti che il player, durante la partita, può decidere o meno, una volta raccolti, di indossarli e questi ultimi producono un miglioramento alla propria armatura o arma che permane finché non vengono tolti. Allorché il ring attualmente in uso viene "staccato" dal player, è necessario che l'oggetto su cui operava ritorni al suo stato originale, quello "non migliorato". Inoltre, è necessario che laddove l'equipaggiamento cam-

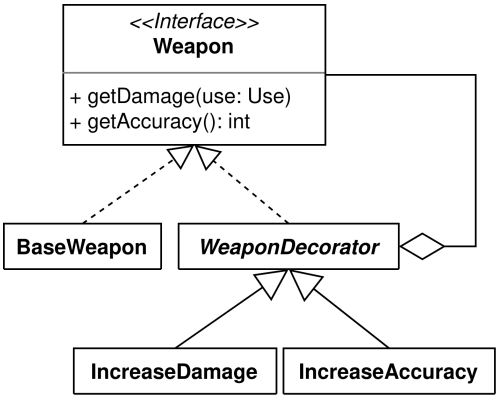


Figura 2.7: Schema UML dell'utilizzo del pattern Decorator a `Weapon`.

biasse durante l'uso di un ring, l'effetto di quest'ultimo transiti al nuovo equipaggiamento. Come mostrato in Figura 2.8 la classe `EquipmentImpl` può sia catturare il proprio stato che ripristinarlo. Il `Memento` è l'oggetto che agisce come istantanea dello stato dell'equipaggiamento, mentre `RingImpl` è responsabile di memorizzare lo stato prima di applicare il suo effetto e di chiedere il suo ripristino quando viene tolto al player. La classe `Memento` è innestata all'interno di `EquipmentImpl`: in questo modo `EquipmentImpl` riesce ad accedere ai campi del `Memento`, mentre `RingImpl` può solo passarlo ad altri oggetti ma non riesce ad accedere al suo stato.

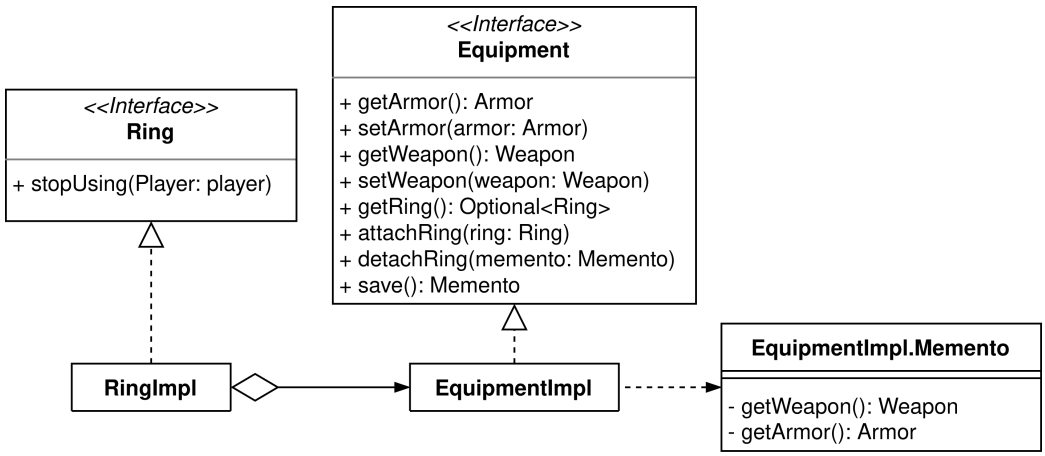


Figura 2.8: Schema UML dell'utilizzo del pattern Memento a `Equipment`.

StatusBarController e StatusBarView

Per quanto attiene la parte di controller e view è stata utilizzata, in collaborazione con Pietro Pezzi, la libreria EventBus di Google Guava che permette l'invio e la ricezione di eventi senza richiedere che i vari componenti si registrino l'un con l'altro. Infatti, le componenti responsabili di dover inviare un evento, nel caso in esame a seguito di un cambiamento di stato nella vita del giocatore o nel suo equipaggiamento, estendendo la classe astratta `EventPublisher`, effettuano il `post` dell'evento, mentre i controller semplicemente si registrano agli eventi interessati. Questo metodo consente un buon disaccoppiamento tra le componenti del dominio che inviano l'evento e i controller che lo ricevono e applicano le conseguenti modifiche sulla relativa View. Entrambe le parti in causa, all'atto dell'invio o ricezione di eventi, non sono consapevoli di chi li abbia generati o ricevuti (se li ha ricevuti). Inoltre, l'utilizzo di questa libreria offre il non sottovalutabile vantaggio di snellire il codice, non preoccupandosi di gestire la notifica dell'evento. In Figura 2.9 uno schema che esemplifica le interazioni tra le varie componenti.

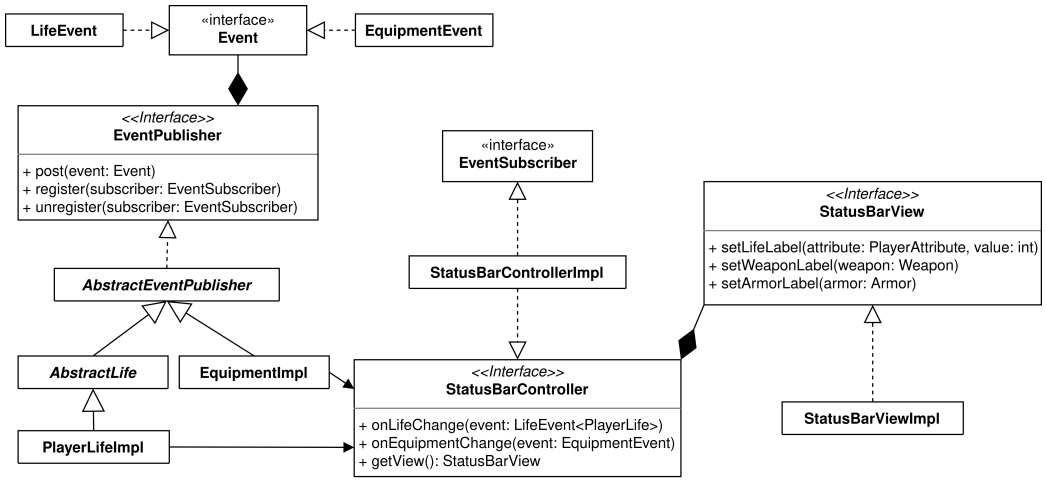


Figura 2.9: Schema del controller e view e loro interazioni.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Gradle è stato configurato in modo da effettuare test con JUnit. Di seguito un elenco dei test realizzati con una breve descrizione:

- `MonsterTypeTest` testa il corretto funzionamento dei mostri e la corretta assegnazione dei vari parametri;
- `PlayerImplTest` testa il corretto funzionamento del personaggio e della sua vita;
- `StandardLevelIncreaseStrategyTest` testa il corretto funzionamento della strategia di incremento del livello al crescere dell'esperienza;
- `FoodImplTest` testa il corretto utilizzo di un'oggetto Food applicato su un giocatore.
- `InventoryImplTest` testa il corretto funzionamento delle principali operazioni dell'inventario, ad esempio: l'utilizzo, la rimozione, lo spostamento e l'aggiunta di un oggetto.
- `ScrollContainerTest` testa la corretta attivazione e rimozione di una pergamena sul giocatore.
- `PotionImplTest` testa il corretto funzionamento di una pozione sul giocatore.
- `PotionTypeTest` testa la correttezza del numero generato in modo casuale dal metodo getHpValue.

- `RingsTest` testa il corretto funzionamento dei ring e l'effettivo ripristino dello stato iniziale nel momento in cui viene staccato dal player;
- `ScrollImplTest` testa il corretto funzionamento di una pergamena sul giocatore.
- `ScrollTypeTest` testa la correttezza del numero generato in modo casuale dal metodo getEffectValue.
- `ItemFactoryTest` testa che il numero di item generati sia coerente.
- `WorldTest` testa che la creazione del mondo avvenga correttamente.

La maggior parte della View e l'interazione con l'utente è stata testata manualmente o attraverso logging su console. L'applicazione è stata provata sia su Linux, che su Windows.

Inoltre (sempre tramite Gradle), abbiamo utilizzato SpotBugs con il relativo plugin per un'analisi del codice in cerca di possibili bug e incorrettezze nella programmazione.

La build di gradle (su Ubuntu, Windows e MacOS) è stata automatizzata tramite un workflow su GitHub.

3.2 Metodologia di lavoro

La parte di analisi e modello del dominio è stata svolta insieme, cercando di massimizzare il riuso tra le varie componenti. Durante lo sviluppo sono state aggiunti alcuni componenti, ma la struttura generale è sempre rimasta invariata. Per quanto concerne l'architettura abbiamo cooperato per la struttura generale, mentre le singole componenti sono state sviluppate in autonomia.

Abbiamo utilizzato il DVCS GIT cercando di lavorare su `feature-branch` separate dal branch `develop` per sviluppare le principali feature dell'applicazione. Inoltre, nella fase finale, si è lavorato su opportuni `fix-branch` per fixare i bug e `refactor-branch` per piccole modifiche. A volte è capitato di fare alcuni commit sul branch `develop`. I merge hanno sempre prodotto conflitti di piccole entità, anche grazie all'indipendenza tra le varie parti.

Di seguito la suddivisione del lavoro:

3.2.1 Manuel Quarneti

- model/world/*
- view/WorldBox

- controller/WorldController

Per l'interazione delle varie entità in seguito al movimento delle stesse ho utilizzato metodi delle classi relative all'azione da eseguire. In particolare ho fatto uso delle classi **Combat** (di Fabio Chiarini), **Player/Inventory** (di Luca Tassinari/Pietro Pezzi) e **Monster** (di Fabio Chiarini).

WorldBox fa uso delle classi **ItemImageGeneratorImpl** (di Pietro Pezzi) e **MonsterImageGeneratorImpl** (di Fabio Chiarini) per la creazione delle **Image** delle varie entità.

3.2.2 Fabio Chiarini

- Mi sono occupato dei nemici quindi nel package `rogue.model.creature` ho creato tutte le classi e le interfacce riguardanti i mostri.
- All'iterno del package `rogue.model.creature` mi sono anche occupato dello sviluppo della interfaccia `Combat` e della sua relativa implementazione.
- Mi sono anche occupato della realizzazione di `ItemImageGenerator` collaborando con Pietro Pezzi e di `MonsterImageGenerator` che si trovano dentro il package `model.view`.

3.2.3 Pietro Pezzi

- All'interno del package `rogue.model.items` mi sono occupato dei package: `food` , `potion`, `scroll`. Inoltre, con la collaborazione di Luca Tassinari, ho sviluppato le interfaccie: `Item`, `ItemFactory` e la sua rispettiva implementazione `ItemFactoryImpl`.
- All'interno del package `rogue.model.events` mi sono occupato di `InventoryEvent`.
- All'interno del package `rogue.view` mi sono occupato dei package: `inventory` e `menu`, mi sono anche occupato, con la collaborazione di Fabio Chiarini, del `ItemImageGenerator`. Ho inoltre collaborato con Luca Tassinari nell'implementazione della `GameView`.
- All'interno del package `rogue.controller` mi sono occupato del package `inventory` e ,con la collaborazione di Luca Tassinari, del `GameController`.

3.2.4 Luca Tassinari

- Per quanto riguarda il package `rogue.model.creature` mi sono occupato delle interfacce comuni `Creature`, `Life` e relative implementazioni astratte. Inoltre ho sviluppato tutte le classi e interfacce che riguardano la gestione del player e la sua vita, comprese le strategie per l'aumento di livello o massimi punti vita;
- All'interno del package `rogue.model.items` sono state da me curati l'equipaggiamento e solo i seguenti package: `armor`, `rings`, `weapons`;
- Il controller e la view della status bar e le semplici interfacce `EventPublisher`, `EventSubscriber` e relative implementazioni astratte.

Ho collaborato con Pietro Pezzi all'implementazione della `GameView` e del relativo `GameController`.

3.3 Note di sviluppo

3.3.1 Manuel Quarneti

- Lambda expressions
- Stream
- JavaDocs
- JavaFX
- Guava
- SLF4J Logger
- Gradle
- SpotBugs

L'algoritmo usato per la generazione delle caverne è stato adattato da questa guida.

3.3.2 Fabio Chiarini

- Utilizzo della libreria JavaFX.
- Uso di SLF4J Logger.

3.3.3 Pietro Pezzi

- Utilizzo della libreria `JavaFX` per il Menu e la view dell'inventario.
- Utilizzo dei file `fxml` per l'interfaccia utente.
- Utilizzo del `SceneBuilder` per facilitare la costruzione dei file fxml.
- Utilizzo della libreria `EventBus` per la gestione degli eventi.
- Utilizzo degli `Optional` per i contenitori.
- Uso di SLF4J Logger.

3.3.4 Luca Tassinari

Ho fatto uso delle seguenti feature avanzate di Java:

- Generici *bounded* per l'interfaccia `Creature` e `LifeEvent`;
- `Optional` al posto di `null`;
- Uso di stream e interfacce funzionali;
- Utilizzo di librerie di terze parti quali JavaFX (con relativo SceneBuilder e fxml), SL4J Logger per il logging su console e EventBus per la gestione degli eventi.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Manuel Quarneti

- - La gestione del turno poteva essere gestita meglio, magari fuori da LevelImpl
- - La gestione delle entità e delle tiles poteva essere ottimizzata meglio, magari utilizzando meglio tecniche di caching
- - La view poteva essere eseguita in parte usando FXML
- + Uso estensivo di JavaDocs
- + Separazione di interfacce e implementazioni
- + Rispetto del paradigma MVC e dipendenze fra pacchetti diversi ridotte al minimo

4.1.2 Fabio Chiarini

Sono abbastanza contento di come si presenta alla fine il nostro gioco, anche per quanto riguarda la grafica che per quanto semplice per me è molto gradevole. Sicuramente potevo gestire meglio la creazione dei mostri tramite enumerazione magari usando un Builder, un'altra cosa che avrei voluto implementare è un logger interno al gioco per mostrare a video tramite delle scritte il risultato del combattimento. Sono contento che come primo grande progetto di gruppo abbiamo deciso di dedicarci alla creazione di un videogioco, probabilmente in futuro mi dedicherò all'aggiornamento di questo progetto aggiungendo il logger e probabilmente anche degli effetti sonori.

4.1.3 Pietro Pezzi

Tutto sommato il lavoro finale mi soddisfa, sono particolarmente contento della implementazione per quanto riguarda gli `Item` consumabili e della parte di model per l'`Inventario`. Detto questo le parti che mi convincono meno sono il controller dell'inventario e l'aggiornamento della view. In conclusione ci tengo a dire che ho vissuto questo progetto come una importante esperienza formativa, rendendomi consapevole di alcune mie lacune a livello organizzativo. E ho apprezzato anche il confronto di un lavoro eseguito in team.

4.1.4 Luca Tassinari

Personalmente sono abbastanza soddisfatto di come sono riuscito a progettare la parte che mi competeva, anche se sono consapevole che alcune parti sarebbero da rifattorizzare. Ad esempio, attualmente quando la vita cambia e genera un evento, memorizza solo il parametro cambiato, ma poiché tutti i parametri che riguardano la vita del player sono degli interi, ho dovuto creare l'enum `PlayerAttribute` per distinguerli (e che ho usato nella View per associargli il nome della relativa label). Questo soluzione è sicuramente subottima e poco "pratica" nel momento in cui si volesse aggiungere un nuovo attributo al personaggio, visto che non esiste alcuna correlazione tra i campi della `PlayerLife` e gli elementi dell'enum (il programmatore dovrebbe apportare modifiche in due punti diversi per agire sullo stesso concetto).

Per quanto riguarda il controller avrei preferito delegare a due ipotetici `PlayerController` e `MonsterController` la gestione del loro movimento. Inevitabilmente questa scelta avrebbe complicato le cose ma, dal mio punto di vista, avrebbe migliorato il disaccoppiamento. Sicuramente avremmo dovuto dedicare più tempo all'architettura e alla gestione del coordinamento tra controller e model.

Peccato che la parte di view relativa al mondo non sia stata concepita *resizable*. Questo è sicuramente un punto negativo, in quanto su schermi di piccole dimensione la griglia di gioco viene inevitabilmente tagliata. Inoltre, altra funzionalità di cui si sente la mancanza è la presenza di uno spazio di view dedicato ad indicare le interazione che intercorrono tra il player e il mostro durante il combattimento.

Appendice A

Guida Utente

Al lancio dell'applicazione, ci si troverà nel Menu principale. Per avviare il gioco sarà necessario inserire nella `TextBox` un nome valido, ovvero con un massimo di 15 caratteri e che non contenga spazi.

Una volta inserito un nome valido verrà aperta una nuova finestra contenente il gioco (vedi fig. A.1).

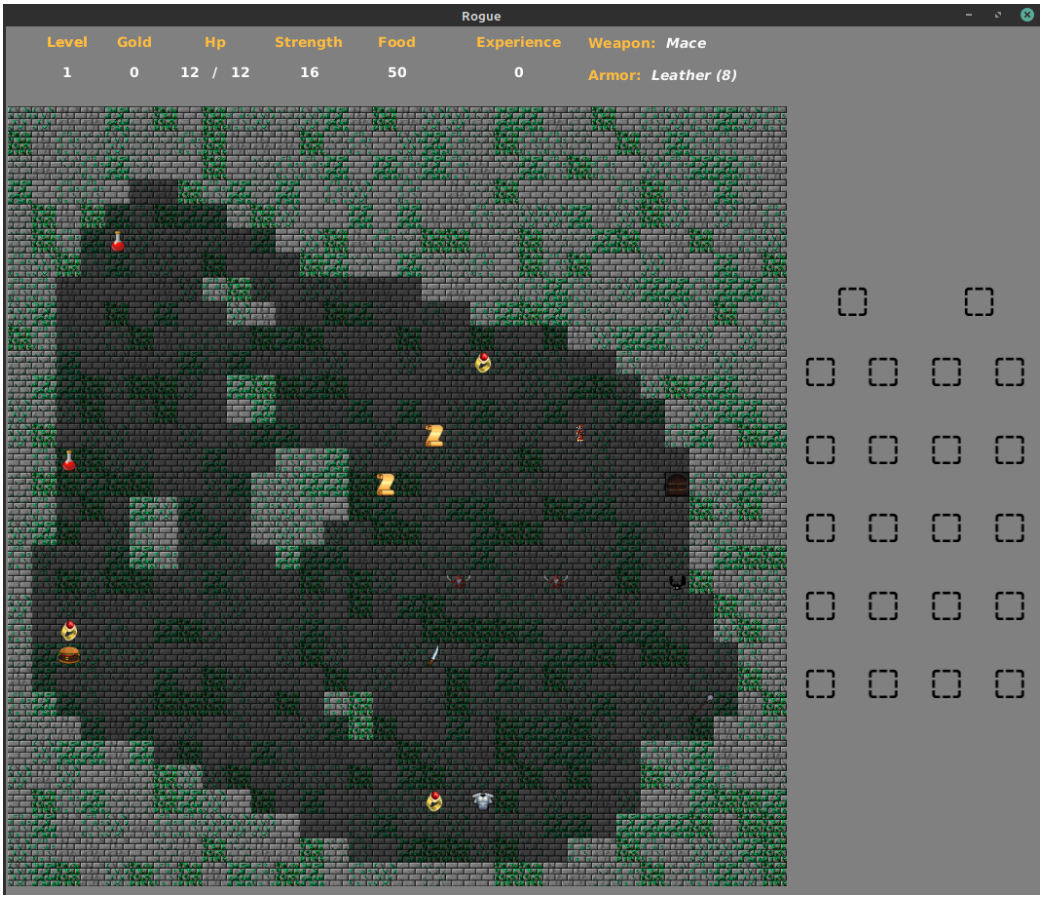
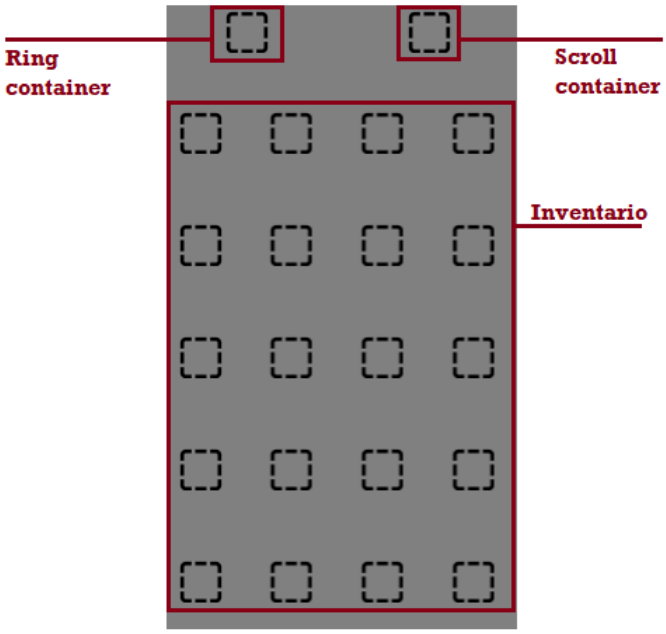


Figura A.1: Schermata di gioco.

A.0.1 Utilizzo inventario



All'interno dell'area segnata come **Inventario** sarà possibile:

- Utilizzare un oggetto cliccandolo con il pulsante `sinistro` del mouse.
- Rimuovere un oggetto cliccandolo con il puslante `destro` del mouse.
- Scambiare due oggetti di posizione cliccandoli consecutivamende con il pulsante `centrale` del mouse.

All'interno dell'area segnata come **Ring container** sarà possibile, nel caso fosse attivo un anello, rimuoverlo cliccandolo con il pulsante `sinistro` del mouse. Infine, All'interno dell'area segnata come **Scroll container** sarà possibile visualizzare il numero di turni rimanenti per l'effetto di una pergamena.

A.0.2 Movimento, attacco e raccolta oggetti

Il giocatore si può muovere con le key LEFT, DOWN, UP, RIGHT oppure utilizzando HJKL con comandi equivalenti a quelli di vi(m).

La raccolta degli oggetti/attacco dei mostri viene gestita automaticamente quando il giocatore si muove su una casella occupata da un'altra entità.

A.0.3 Passaggio al livello successivo

In ogni livello è presente una porta, quando il giocatore si muove su di essa, viene immediatamente trasportato al livello successivo.

Appendice B

Esercitazioni di laboratorio

B.0.1 Luca Tassinari

- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p101278
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p104064