Relazione del progetto di programmazione ad oggetti

PlatformCraft

Realizzato da:

Alessandro Ronchi
Simone Carpi
Daniele Zecchini
Gabriele Abeni

Indice

Capitolo 1
Analisi

1.1 Requisiti

Il software si pone l'obiettivo di realizzare un videogioco appartenente al genere platform 2D che dà all'utente la possibilità di creare, modificare e giocare i propri livelli. Un gioco platform è un gioco in cui il giocatore controlla un personaggio che può muoversi, saltare e interagire con un livello composto da diverse piattaforme. L'obiettivo del giocatore è quello di giungere alla fine del livello evitando trappole, nemici e ostacoli.

Requisiti funzionali

- Il software si dovrà comporre di due sezioni principali: una sezione Editor, che fornisce al giocatore tutti gli strumenti necessari per creare un livello e modificarlo, e una sezione di gioco che consente al giocatore di giocare i livelli creati tramite l'Editor.
- Nella sezione Editor, al giocatore dovrà essere data la possibilità di aggiungere e rimuovere vari elementi dal livello, salvare il proprio livello e caricare un livello esistente per modificarlo. Inoltre, l'editor dovrà avere una funzione che permette di "azzerare" il livello in creazione, rimuovendo tutti gli elementi al suo interno.
- Tra gli elementi a disposizione del giocatore ci saranno:
    - Il personaggio giocabile
    - Vari tipi di nemici
    - Delle trappole
    - Delle piattaforme su cui il giocatore potrà camminare
    - Un traguardo che il giocatore dovrà raggiungere
- All'interno del livello in creazione, non si potrà aggiungere più di un personaggio giocabile e più di un traguardo. Questi due elementi

sono comunque considerati obbligatori per la creazione di un livello. Si dovrà quindi realizzare un sistema che dà un feedback all'utente in caso di una configurazione errata del livello o in caso il giocatore tenti di aggiungere più di un personaggio o più di un traguardo.
- I livelli creati verranno salvati su file, in modo che possano essere esportati e condivisi con altri utenti.
- Nella sezione di gioco, il giocatore partirà dalla posizione specificata all'interno del livello e dovrà evitare nemici e trappole per riuscire ad arrivare al traguardo illeso. La partita è considerata persa se il giocatore viene toccato da un nemico (eccetto un caso specifico illustrato di seguito) o da una trappola oppure se il giocatore cade nel bordo inferiore della mappa.
- Il giocatore potrà eliminare i nemici del livello saltando sopra la loro testa. In questo caso, il giocatore non perderà la partita ma ucciderà il nemico.
- Al giocatore si dovrà fornire un esempio di livello, che mostra le potenzialità dell'editor, e che il giocatore potrà giocare e modificare a piacimento.

Requisiti non funzionali

- Nella fase di gioco, l'applicazione dovrà reagire in modo rapido ai comandi del giocatore, quindi sarà necessario garantire delle buone prestazioni del sistema.

1.2 Analisi e modello del dominio

L'entità principale all'interno del modello del dominio dell'applicazione è il livello. Un livello è costituito da un insieme di entità di gioco che interagiscono tra loro e modificano il loro stato secondo gli ordini del livello. Le entità sono di diverso tipo:

- Personaggio: Il personaggio giocabile. Il livello può ordinare al personaggio di muoversi in una specifica direzione, in base ai comandi del giocatore.
- Nemico: un'entità di gioco che si muove all'interno del livello e agisce secondo uno specifico comportamento.
- Trappola: un'entità di gioco che non cambia la sua posizione ma reagisce al passaggio del giocatore in un certo modo.
- Elemento della mappa: un'entità statica che non modifica la sua posizione e costituisce una piattaforma sulla quale personaggio e nemici possono camminare.

Ogni entità di gioco si compone di una fisica, che definisce il modo con cui un'entità si muove all'interno del livello, e una "scatola delle collisioni" (CollisionBox), che concettualizza le dimensioni dell'entità e cattura le collisioni tra un'entità e un'altra, cioè le situazioni in cui la CollisionBox di un'entità entra in contatto con quella di un altro oggetto nel gioco.
La fisica calcola il movimento della sua entità basandosi anche sulle informazioni che riceve dalla CollisionBox riguardo le collisioni con altri oggetti.
Anche il livello possiede una sua scatola delle collisioni che rappresenta le dimensioni del livello. Nessuna entità può uscire dal livello.
Il livello è a sua volta controllato da una componente che costituisce il "motore" del gioco (Engine), che ordina al livello di aggiornarsi durante lo svolgimento della partita.
La parte di creazione del livello è invece gestita da un Editor che modifica il livello, aggiungendo e rimuovendo le entità che lo compongono.

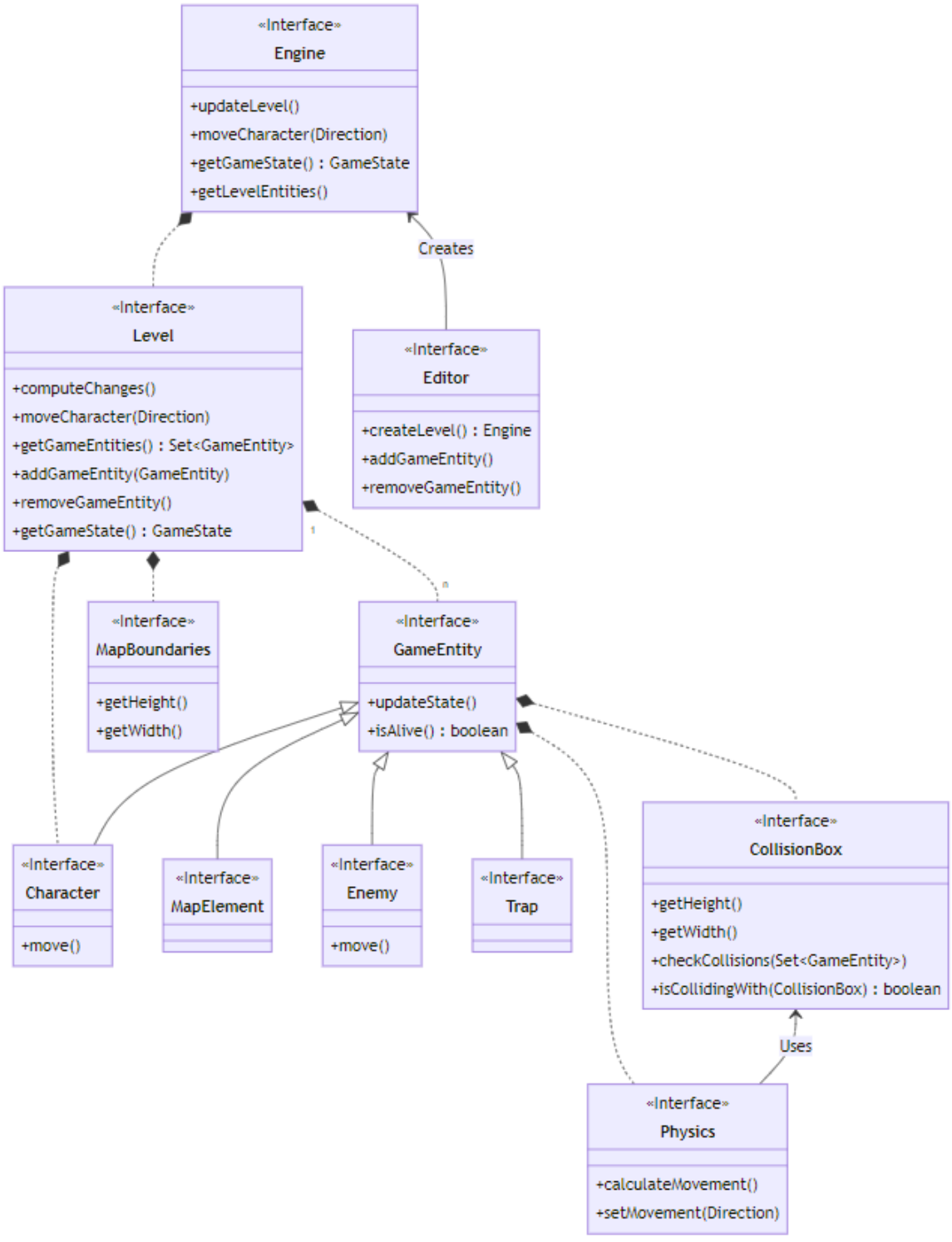

Figura 1.1: Schema UML del modello di dominio.

Capitolo 2
Design

2.1 Architettura

L'architettura dell'applicazione è basata sul pattern architetturale MVC (Model-View-Controller). Il Controller si divide in diverse sezioni, ognuna con un ruolo specifico. In particolare, il Controller si compone di:

- Un LevelRunner, che ha il compito di gestire la fase di gioco.
- Un LevelEditor, che gestisce l'editor del livello
- Un LevelSerializer che si occupa di salvare e caricare da file i livelli che devono essere giocati o creati.

Il LevelRunner ha il compito di gestire il Game Loop, cioè la serie di operazioni che devono essere svolte ogni volta che il livello si aggiorna.
In particolare, il LevelRunner riceve dalla View i comandi dell'utente e li passa all'Engine, che si occupa della loro esecuzione. Inoltre, il LevelRunner componente dice periodicamente alla View di aggiornarsi, richiedendo quindi il render del livello. Il LevelRunner, quindi, rappresenta una componente attiva dell'applicazione.
Il LevelEditor, invece è una componente passiva che si occupa di richiedere all'Editor le operazioni di modifica del livello nel momento in cui la View registra un input da parte dell'utente.
Il Controller si compone anche di un modulo di input/output su file costituito dal LevelSerializer.
Editor e Engine rappresentano l'entry point del Model, e costituiscono una sorta di interfaccia con cui le componenti del Controller comunicano con il Level.

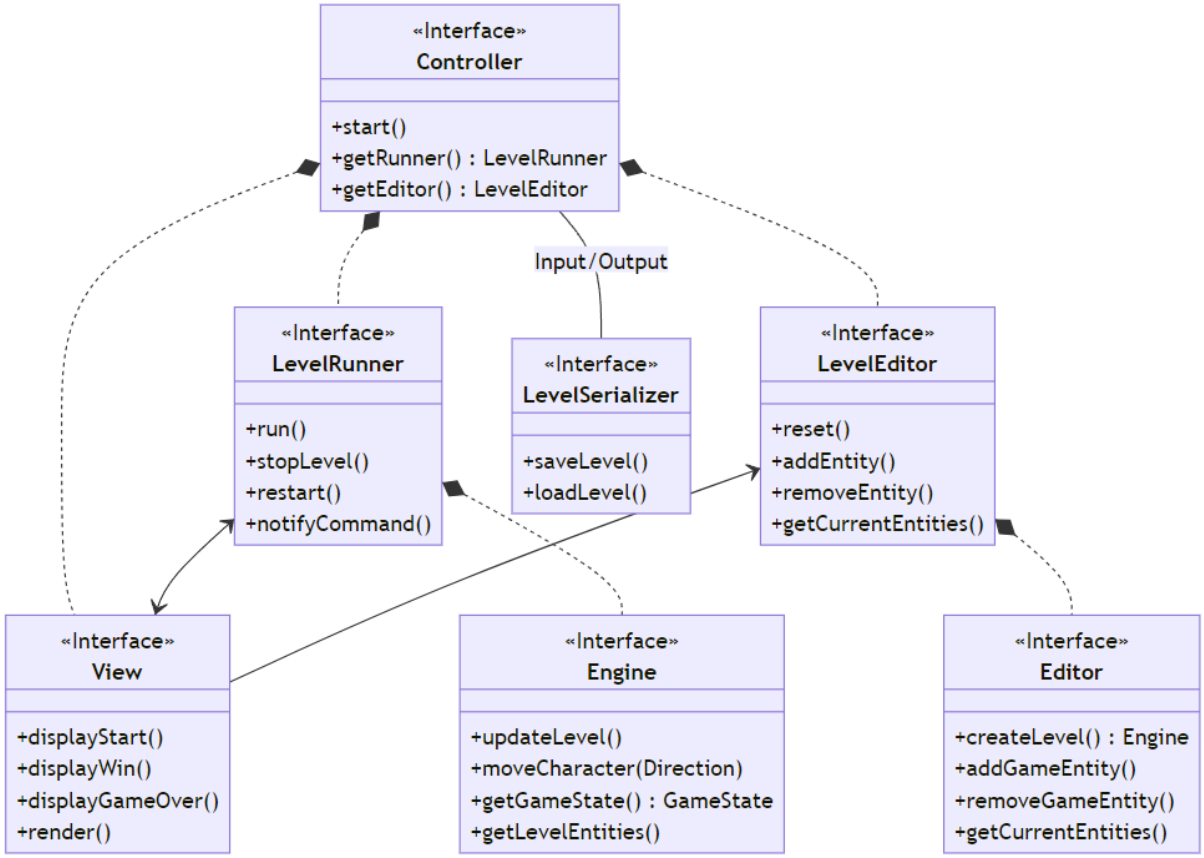

Figura 2.1: Schema UML architetturale dell'applicazione. Engine e Editor rappresentano l'entry-point del Model

2.2 Design dettagliato

Alessandro Ronchi

Fisica di un'entità

Problema: Realizzare un sistema che possa gestire la fisica delle varie entità tenendo conto del fatto che ogni entità implementa un meccanismo di movimento differente. La fisica deve dunque essere configurabile, attraverso specifici parametri, a seconda delle esigenze di chi la utilizza.

Soluzione: La gestione della fisica utilizza il *pattern Builder* per far sì che le classi utente possano creare istanze dell'interfaccia Physics delegando l'istanziazione alla classe PhysicsBuilder e configurando la fisica nella maniera desiderata. Le implementazioni di Physics sono due: la

LinearPhysics, che implementa un movimento lineare su entrambi gli assi, e la AcceleratedPhysics, che estende la Linear implementando un'accelerazione sull'asse X e introducendo la gravità sull'asse Y.
A seconda della configurazione voluta, il PhysicsBuilder sceglie se istanziare l'una o l'altra classe.
L'estendibilità di questa soluzione è garantita dal meccanismo di ereditarietà che consente di aggiungere una variazione della fisica attraverso l'estensione di LinearPhysics.

Per risolvere questo problema è stato preso in considerazione un *decorator Pattern* in cui la LinearPhysics sarebbe stata decorata da implementazioni di Physics che avrebbero aggiunto le funzionalità richieste. Tuttavia, l'utilizzo di questo pattern avrebbe comportato problemi legati al fatto che il decoratore avrebbe avuto la necessità di modificare il comportamento e lo stato dell'oggetto decorato (la AcceleratedPhysics agisce modificando la velocità della Linear), e nel farlo si sarebbe avuta una parziale violazione dell'incapsulamento dell'implementazione a causa dell'aggiunta di appositi setter pubblici.
D'altro canto, anche il Decorator avrebbe fornito ottime opportunità di estendibilità.

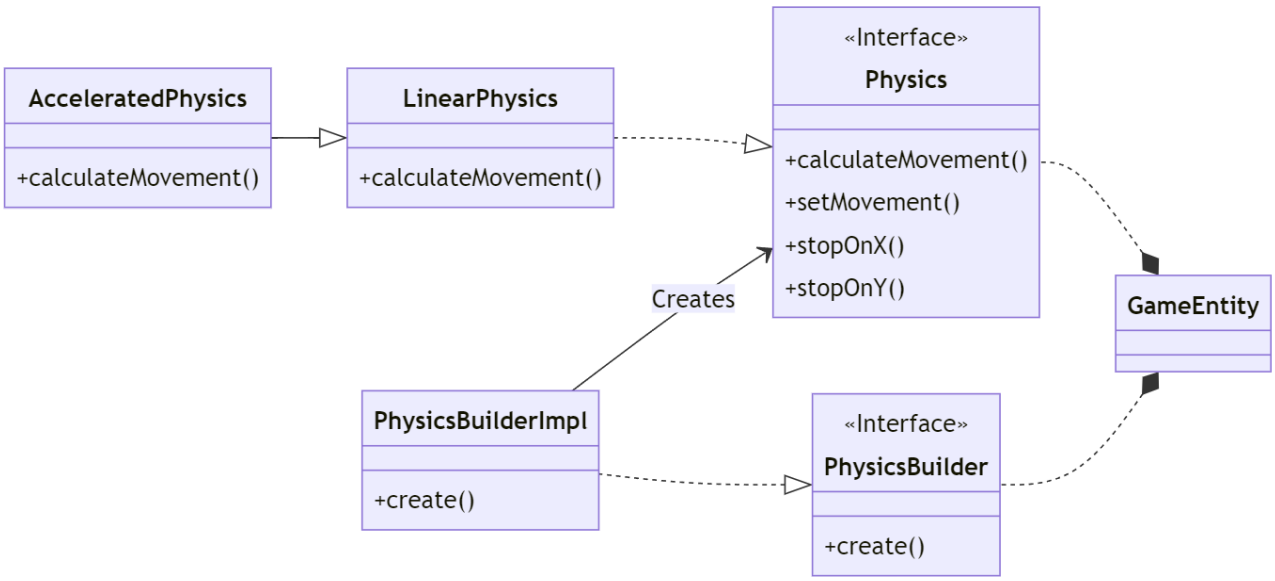

Figura 2.2: Schema UML delle entità nella gestione della fisica. GameEntity costituisce la classe user della fisica.

Controller

Problema: Progettare il Controller in modo che consenta di gestire le due parti principali dell'applicazione: quella reattiva rappresentata dalla gestione della partita e quella passiva rappresentata dalla funzione Editor. Inoltre, entrambe le funzioni necessitano di effettuare operazioni di lettura e/o scrittura su file, e realizzare metodi specifici in ogni componente avrebbe portato a duplicazioni di codice.

Soluzione: Per evitare di concentrare una quantità eccessiva di responsabilità nel Controller, si è deciso di suddividere le funzioni del Controller in due componenti: il LevelRunner, che gestisce l'esecuzione della partita e il LevelEditor, che gestisce la fase di modifica del livello, in modo da rispettare l'SRP. Il Controller principale svolge unicamente la funzione di delega delle operazioni. Inoltre, per evitare la duplicazione di codice nella lettura e scrittura su file, si è deciso di raggruppare queste funzionalità in un'ulteriore componente del Controller responsabile dell'I/O su file: il LevelSerializer.
Essendo il LevelRunner una componente attiva, esso deve necessariamente utilizzare un Thread separato che lavora comunicando con l'EDT. In questo modo, si riesce a rispettare il requisito non funzionale illustrato nella sezione 1.1.
A pagina seguente lo schema UML.

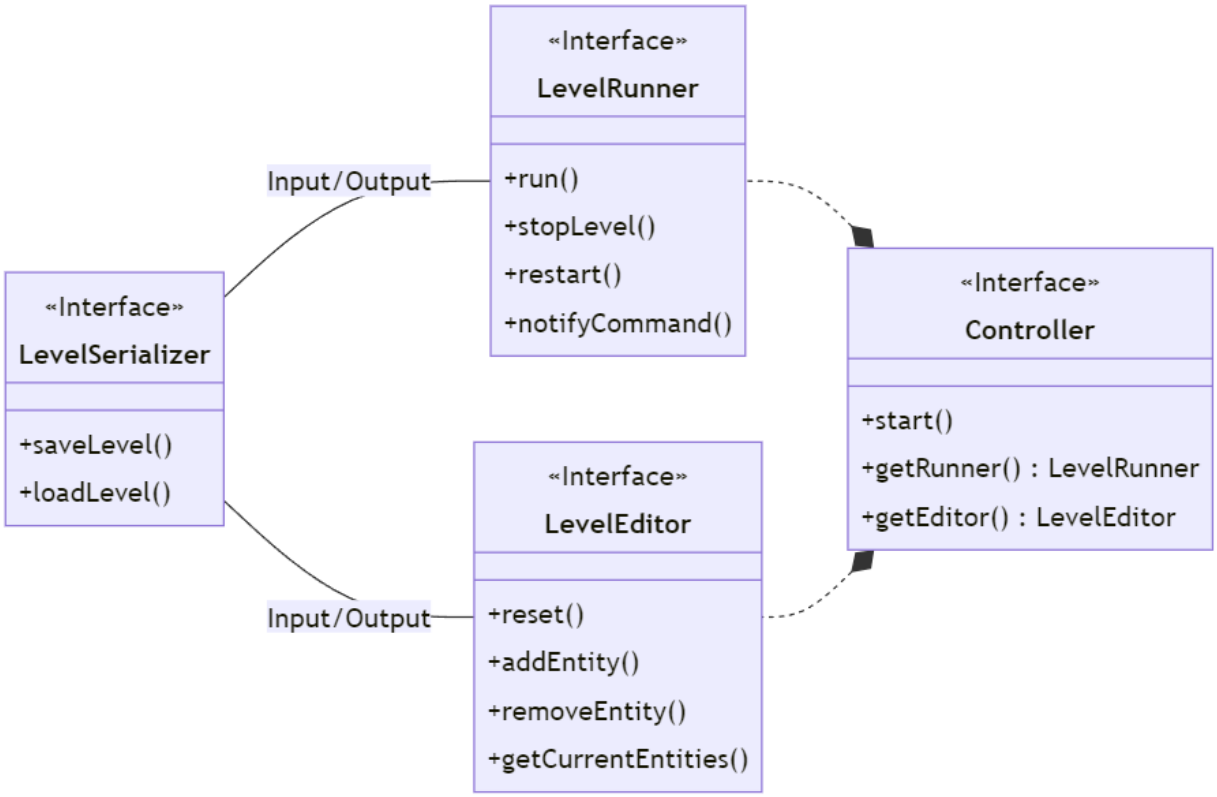

Figura 2.3: Schema UML che mostra le relazioni tra LevelRunner e RunnerAgent e la comunicazione con la View.

Carpi Simone

Durante lo sviluppo del progetto mi sono dedicato allo sviluppo dei vari elementi presenti all'interno del livello quali: personaggio, nemici, trappole e i vari componenti della mappa come i blocchi o la Finish location (il traguardo). Mentre per quanto riguarda la parte al di fuori del Model mi sono dedicato a parte del Controller e alla creazione dell'interfaccia utente.

Gestione delle GameEntity

Problema In questa parte mi sono dedicato a come accomunare le diverse GameEntity presenti nel gioco. Sono presenti più entità nel livello e possono essere distinte in: Character, Enemy, Trap e MapElement.

Soluzione Queste diverse GameEntity condividono una logica simile ed ho pensato fosse utile raccogliere tutti questi elementi e rappresentarli attraverso una interfaccia comune detta GameEntity. Questa è l'interfaccia

con la quale le altre classi interagiscono con le varie GameEntity. Era necessario un modo per poter distinguere queste GameEntity e per questo motivo ad ognuno di queste entità viene assegnato un tipo che viene rappresentato da un Enum (EntityType) che mi permette di differenziarle.

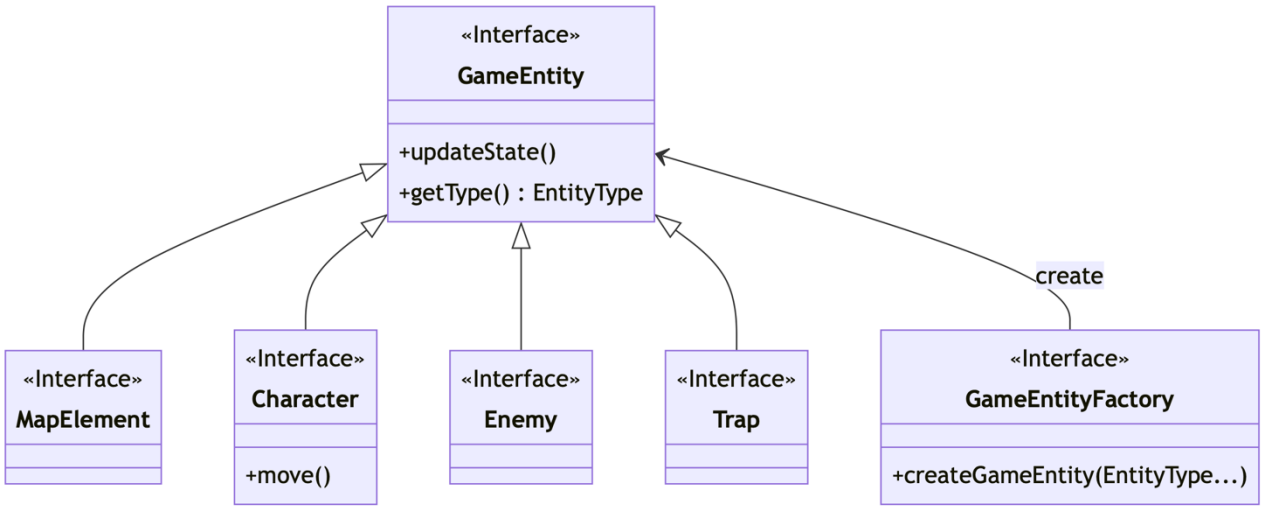

Figura 2.4

Creazione delle GameEntity

Problema Vista la presenza di molte GameEntity di tipo diverso mi sembrava necessario trovare un modo per poter costruire queste varie entità in un unico modo, invece di andare a reperire ogni volta le diverse implementazioni.

Soluzione Per questo motivo ho optato, per risolvere questo problema, di utilizzare il pattern Factory attraverso un factory method presente nell'interfaccia GameEntityFactory, il quale mi restituisce proprio una GameEntity. Mi sembrava la scelta più legittima perché tutte le GameEntity prendono, durante la loro implementazione, gli stessi argomenti. Inoltre, bisognava anche trovare un modo per chiamare le diverse implementazioni delle entity e quindi scegliere il tipo di entità da creare. In questo caso ho deciso di fargli prendere come argomento anche il tipo di entità che si vuole che venga restituita. Questo permette di poter

istanziare altre entità nel caso vengano aggiunte diversi nuovi tipi di GameEntity.

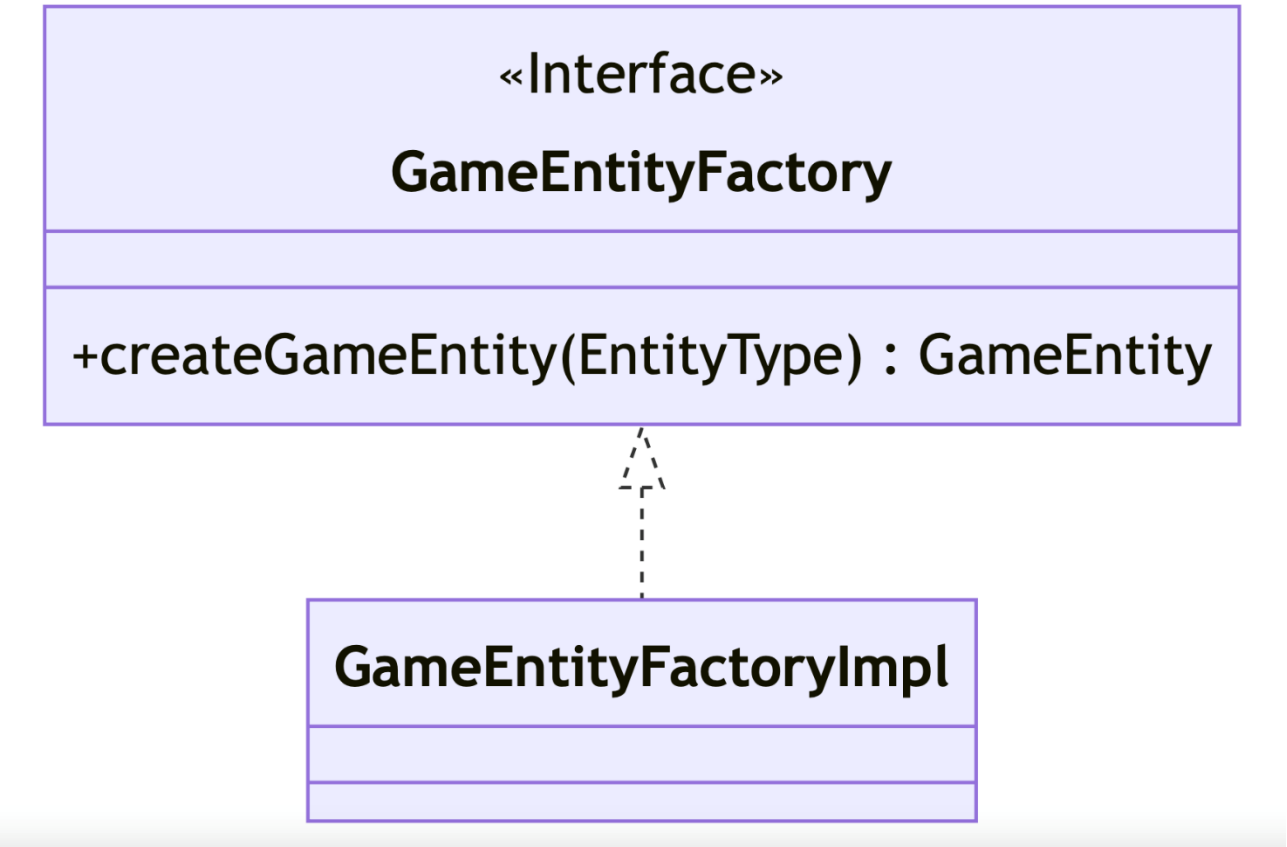

Figura 2.5

Differenziazione dei nemici

Problema All'interno del gioco sono presenti (attualmente) due tipologie di nemici e bisognava trovare il modo di poterli differenziare o caratterizzarli in modo che fossero diversi nel comportamento, rendendo più dinamico il gioco.

Soluzione In questo caso ho deciso di lavorare sulla ereditarietà, partendo dall'interfaccia Enemy e dalla sua implementazione EnemyImpl. I due nemici quali, SimpleEnemy e StrongEnemy, estendono dalla classe EnemyImpl. Questa classe si occupa di gestire il funzionamento generale dei nemici e che quindi non cambia in base alla tipologia del nemico stesso.

L’unica differenza che li distingue è il modo con cui attaccano il player. Infatti, mentre il SimpleEnemy si muove normalmente avanti e indietro, lo StrongEnemy se vede che il player si trova nel suo raggio visibile, allora lo insegue. Ho utilizzato un template method per risolvere questo inconveniente, in modo che delego alle sottoclassi come operare e quest’ultime si occupano di scegliere come si dovrà muovere il nemico. Il template method nella classe astratta EnemyImpl è updateState() perché al suo interno utilizza un metodo astratto che è move() che gestisce come si dovrebbe comportare il nemico. Avevo preso in considerazione anche oltre al template method l’utilizzo del pattern Builder per la costruzione dei nemici. Il pattern Builder è un pattern che mi avrebbe permesso di creare nemici molto variegati impostando loro una dimensione, velocità o intelligenza, però da come sono state implementate le altre classi non sembrava adatto proprio perché il nemico poteva essere già modificato con il Builder della fisica che mi permette di impostare una velocità al nemico ed inoltre questi due nemici sono due tipi di EntityType a cui può essere modificata o assegnata una dimensione.

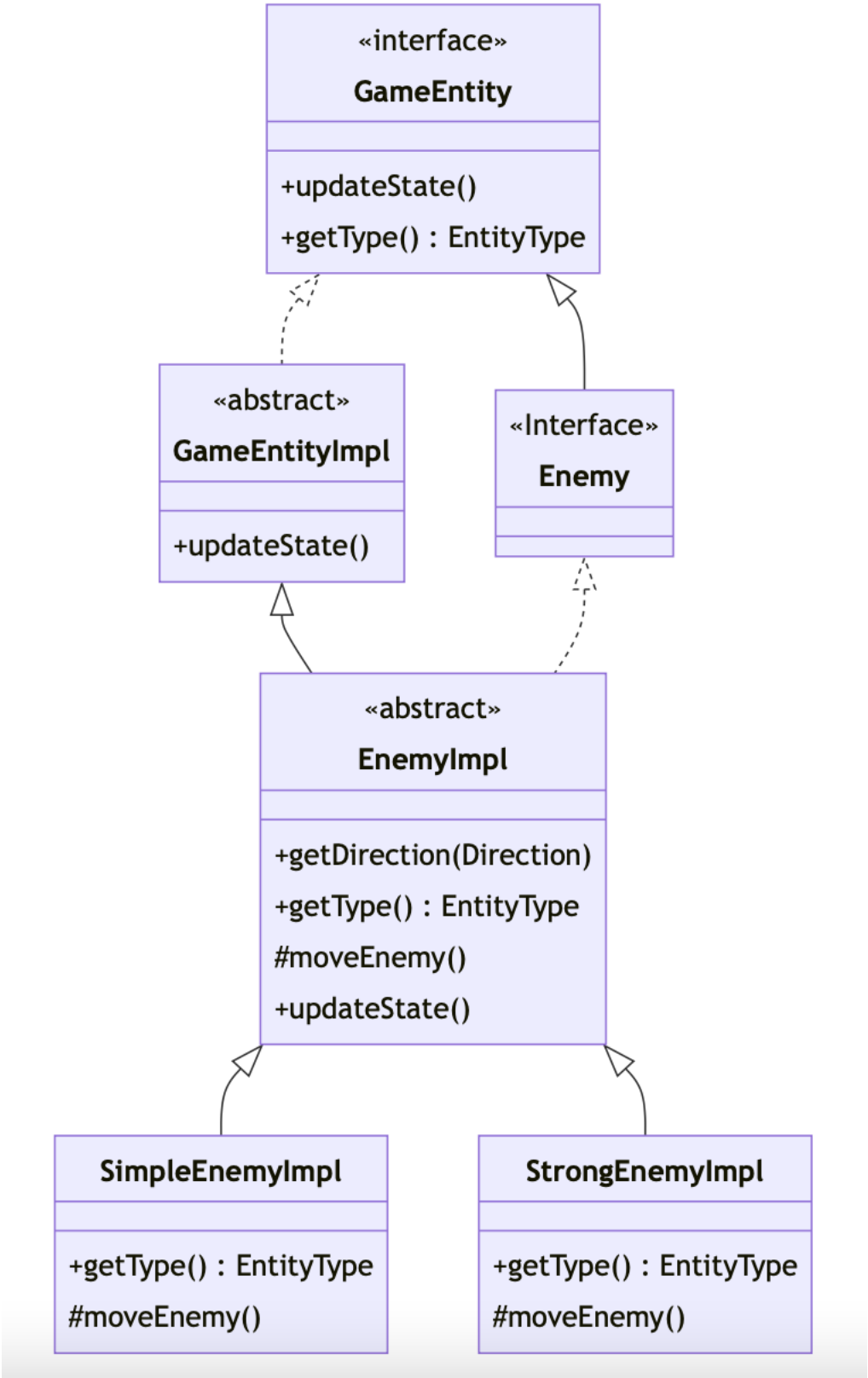
«interface»
GameEntity
+updateState()
+getType() : EntityType
«abstract»
GameEntityImpl
+updateState()
«Interface»
Enemy
«abstract»
EnemyImpl
+getDirection(Direction)
+getType() : EntityType
#moveEnemy()
+updateState()
SimpleEnemyImpl
+getType() : EntityType
#moveEnemy()
StrongEnemyImpl
+getType() : EntityType
#moveEnemy()

Figura 2.6

GUI

Per la GUI mi sono dedicato alla creazione dell'interfaccia che appare all'avvio dello schermo (che viene definita dalla TitleScreen), dell'interfaccia dell'editor e di quella del level. La TitleScreen si compone sia di un LevelView, che si occupa di avviare la GUI del livello in esecuzione, sia di un EditorView che permette all'utente di poter scegliere i vari GameEntity da aggiungere nel livello che si sta creando.
Il tutto viene gestito dalla view principale View, la quale delega i compiti alle sue sottoclassi.

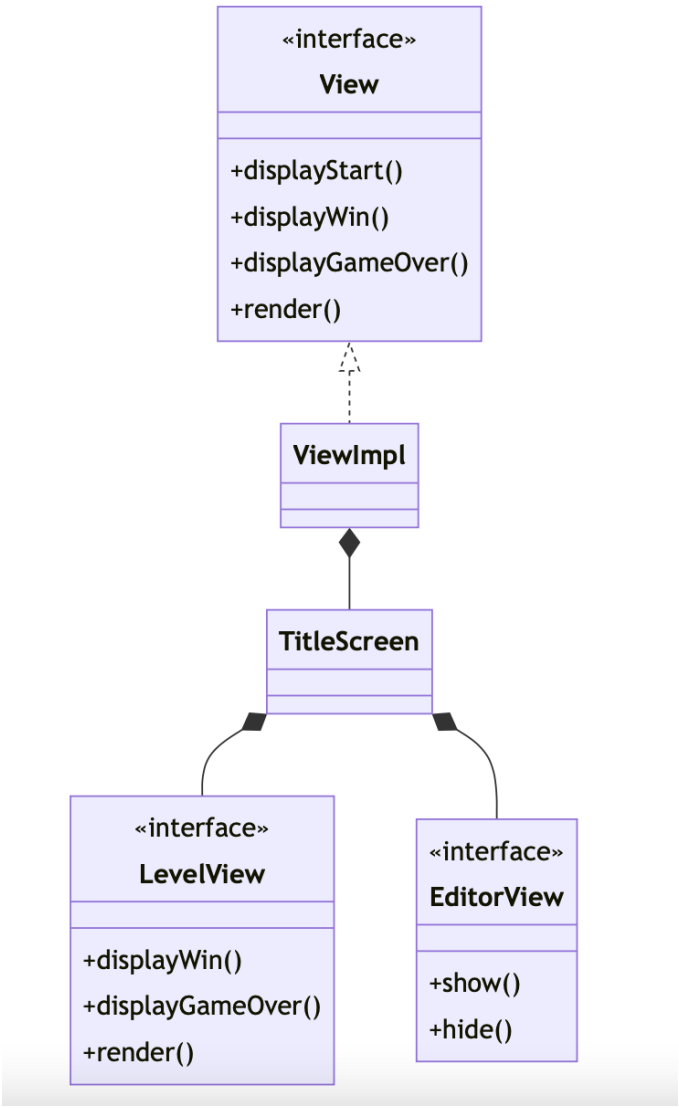

Figura 2.7

Controller

All'interno del Controller mi sono dedicato al LevelEditor, gestore delle operazioni svolte all'interno dell'editor dove l'utente può aggiungere, rimuovere elementi, caricare o salvare file. LevelEditor viene implementato da LevelEditorImpl una classe che delega i compiti da svolgere. Si compone di un Editor che è il motore del gioco e di un LevelSerializer che si occupa di salvare e caricare i file.

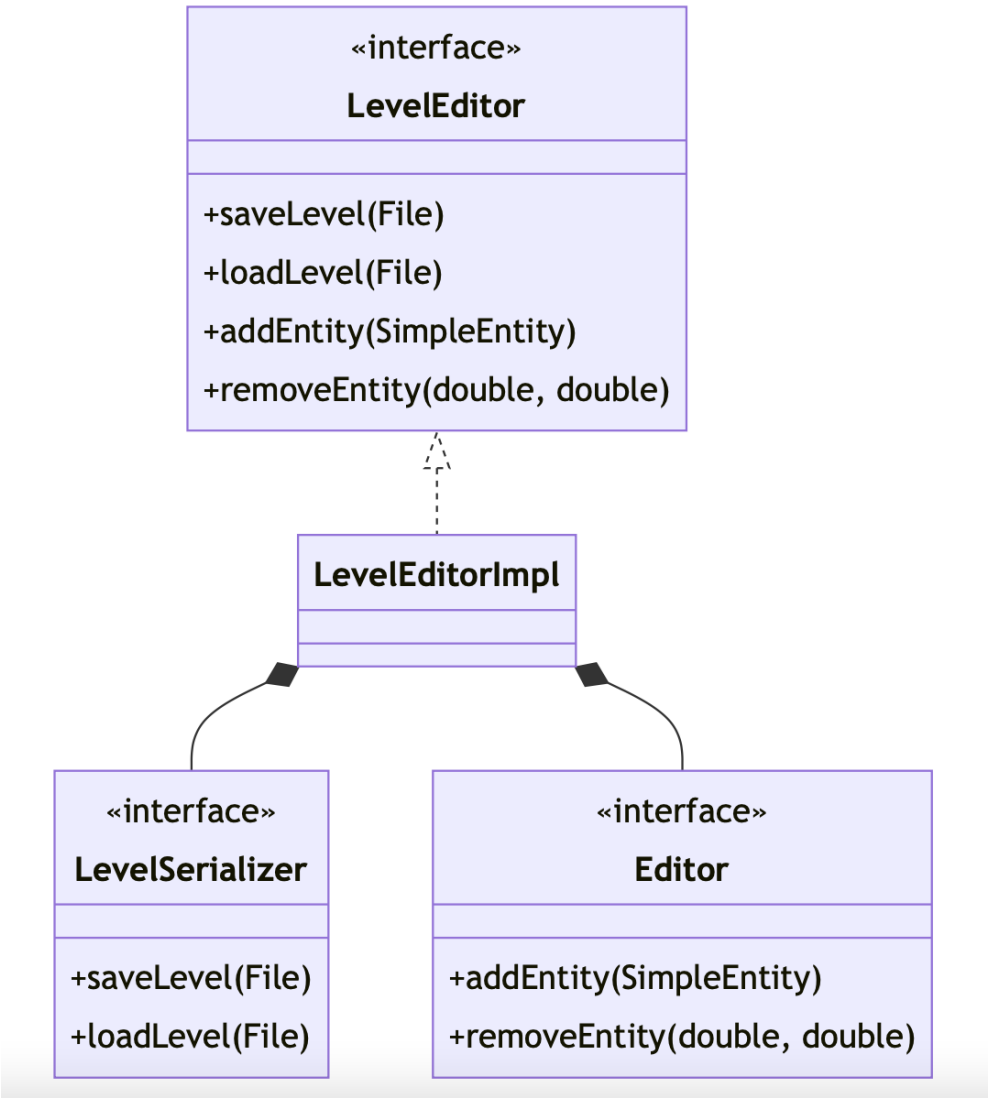

Figura 2.8

Daniele Zecchini

Problema: Come individuare e gestire le Collisioni che avvengono tra le varie entità di gioco e i confini del livello.

Soluzione: È stata creata la classe CollisionBox è stata utilizzata la classe CollisionBox allo scopo di individuare e restituire le informazioni di collisioni riguardanti una game entity che verrà passata al costruttore quando la classe viene istanziata. Per gestire le informazioni riguardanti il lato dell'entità di gioco su cui è avvenuta la collisione è stata creata l'interfaccia Collision per poi essere estesa da due interfacce: una per descrivere la collisione con i confini della mappa e una per descrivere le collisioni con le altre entità di gioco, che oltre a riportare la direzione su cui è avvenuto l'urto riporta anche l'entità con cui è avvenuta la collisione.

Problema: Come individuare le varie collisioni.

Soluzione: Per farlo è stata creata la classe Boundaries che attraverso la posizione dell'entità descrive i suoi bordi.

Gabriele Abeni

Problema: Gestire la manipolazione del livello da parte di Editor, Engine e delle GameEntity presenti al suo interno senza violare l'incapsulamento e senza introdurre problemi di sicurezza legati ad una possibilità di modifica del livello dall'esterno durante l'esecuzione.

Soluzione: Ho utilizzato il *pattern Proxy* per gestire l'accesso al livello in modo trasparente. Il Proxy è costituito dalla classe UnmodifiableLevel, la quale controlla l'accesso al livello disabilitando le operazioni che lo modificano direttamente, come addGameEntity e removeGameEntity. L'Editor utilizza il livello in modo diretto, ma passa all'Engine e alle GameEntity create un'istanza di UnmodifiableLevel. In questo modo, GameEntity e Editor non possono aggiungere o togliere direttamente le entità dal livello.

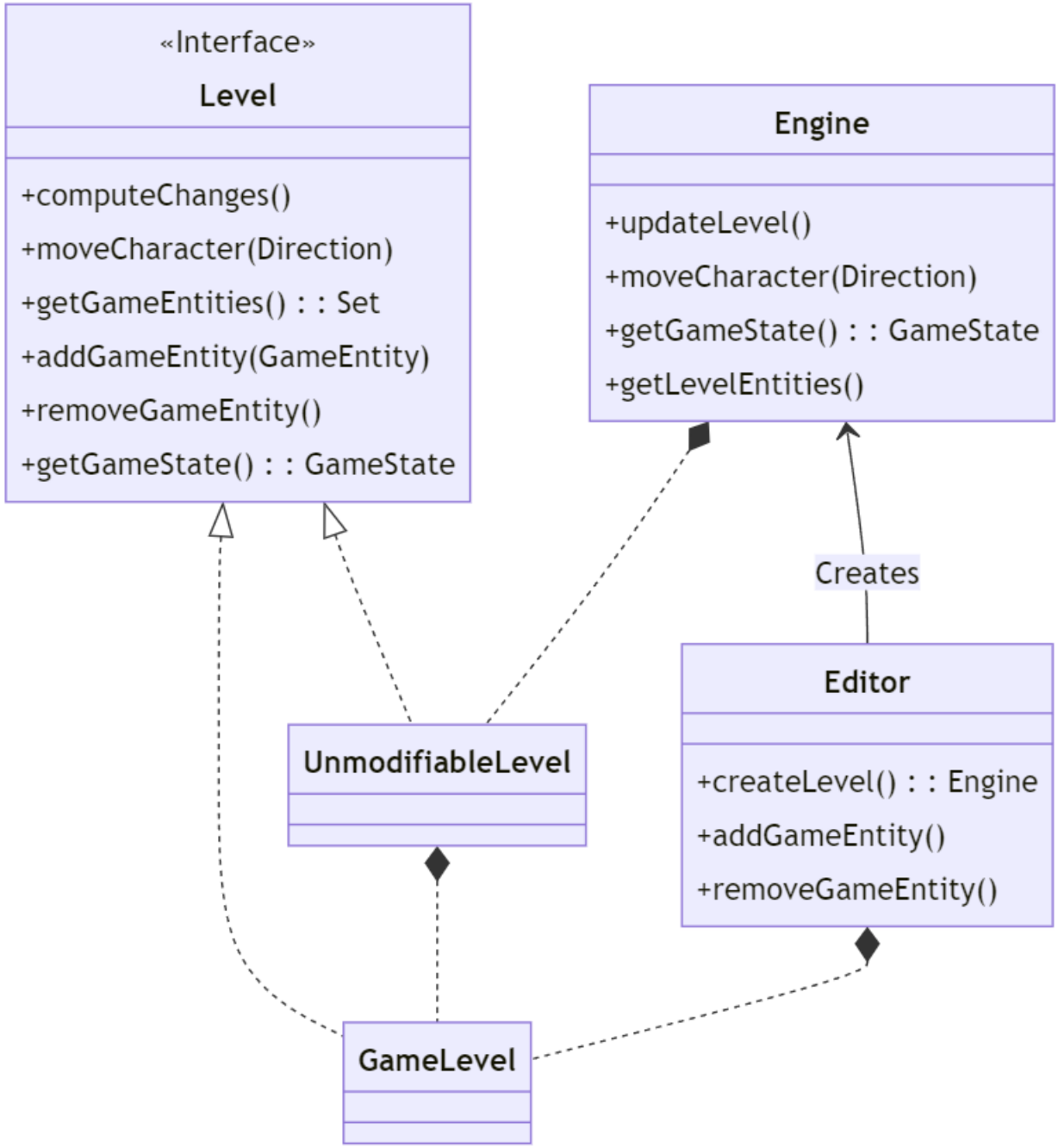

Figura 2.9

Problema: Creare un'entità che incapsula il concetto di creatore del livello all'interno del Model.

Soluzione: Per l'Editor è stato utilizzato un *pattern Builder,* dove l'Editor crea un livello attraverso appositi metodi che specificano l'entità da aggiungere o rimuovere al livello in creazione. Inoltre, la createLevel() crea il livello solo se la configurazione è valida.

Capitolo 3
Sviluppo

3.1 Testing automatizzato

Per il test automatizzato delle classi è stato utilizzato JUnit 5.

Alessandro Ronchi

Il test automatizzato è stato svolto sulle classi che implementano la gestione della fisica delle entità, e dunque LinearPhysics, AcceleratedPhysics e PhysicsBuilderImpl.

Per LinearPhysics, il test si è concentrato sul verificare il corretto movimento lineare e la corretta gestione delle collisioni nei diversi casi e nelle diverse configurazioni (testando sia l'arresto in caso di collisione, sia il rimbalzo).
Per AcceleratedPhysics il test verifica se la gestione della velocità in modo accelerato funziona nel modo corretto. Inoltre, verifica che, anche se la gravità è abilitata su un oggetto, questo non cade se al di sotto vi è un altro oggetto.
Per PhysicsBuilderImpl, il test verifica che le classi istanziate siano corrette e abbiano il comportamento desiderato in base alla configurazione inserita. Inoltre, il test verifica che in caso di una configurazione non valida, il metodo di creazione tiri eccezione.

Simone Carpi

Ho implementato due tipi di classi di test: TestCharacter e TestEnemy.
TestCharacter si occupa di controllare il giusto funzionamento del personaggio. In questo caso si controlla il corretto ricevimento di determinati comandi e il controllo sul movimento. Inoltre vengono testate

eventuali collisioni con i nemici o trappoli cercando di capire se il personaggio muore o meno in determinate situazioni.

Riguardo il TestEnemy si effettuano dei test per capire se il nemico si muove correttamente nel livello, controlli sullo stato dei nemici e in particolare varie situazioni con il personaggio e come reagisce il nemico con il player nei paraggi.

Daniele Zecchini

Nel test è stato verificato il funzionamento della CollisionBoxImpl nell'individuare correttamente le collisioni e la loro direzione.
È stato anche testato il funzionamento della classe Boundaries per quanto riguarda i metodi che controllano se due Boundaries si intersecano.

Gabriele Abeni

Non è stato svolto alcun test automatizzato sulle parti che ho realizzato.

3.2 Note di sviluppo

Alessandro Ronchi

Utilizzo di Stream e method reference
Largamente utilizzato. Un esempio di entrambi è:
https://github.com/AR0018/OOP23-platform-craft/blob/b35da7896f43d7639916f56f7ab82dd5fb79bd0f/src/main/java/it/unibo/model/physics/impl/LinearPhysics.java#L58

Utilizzo della libreria JTS Topology Suite per la gestione delle posizioni in due dimensioni
Permalink: https://github.com/AR0018/OOP23-platform-craft/blob/b35da7896f43d7639916f56f7ab82dd5fb79bd0f/src/main/java/it/unibo/model/physics/impl/Position2D.java#L12

Utilizzo di lambda expressions
Nell'esempio linkato, utilizzato per il pattern Strategy nella gestione dei comandi al Controller: https://github.com/AR0018/OOP23-platform-craft/blob/b35da7896f43d7639916f56f7ab82dd5fb79bd0f/src/main/java/it/unibo/controller/api/Command.java#L15

Utilizzo di Optional
Permalink: https://github.com/AR0018/OOP23-platform-craft/blob/b35da7896f43d7639916f56f7ab82dd5fb79bd0f/src/main/java/it/unibo/controller/impl/LevelRunnerImpl.java#L22

Programmazione multi-threading
Utilizzata la gestione del Controller della GUI reattiva.
Permalink: https://github.com/AR0018/OOP23-platform-craft/blob/b35da7896f43d7639916f56f7ab82dd5fb79bd0f/src/main/java/it/unibo/controller/impl/RunnerAgent.java#L16

Switch in forma lambda
Permalink: https://github.com/AR0018/OOP23-platform-craft/blob/b35da7896f43d7639916f56f7ab82dd5fb79bd0f/src/main/java/it/unibo/controller/impl/RunnerAgent.java#L49

Utilizzo della libreria Jackson per la serializzazione in json dei livelli
Permalink: https://github.com/AR0018/OOP23-platform-craft/blob/b35da7896f43d7639916f56f7ab82dd5fb79bd0f/src/main/java/it/unibo/controller/impl/SerializerImpl.java#L19

Simone Carpi

- Uso di Stream largamente utilizzato. Un esempio: https://github.com/AR0018/OOP23-platform-craft/blob/b35da7896f43d7639916f56f7ab82dd5fb79bd0f/src/main/java/it/unibo/model/entities/impl/CharacterImpl.java#L63
- Uso di Optional. Un esempio: https://github.com/AR0018/OOP23-platform-craft/blob/b35da7896f43d7639916f56f7ab82dd5fb79bd0f/src/main/java/it/unibo/view/impl/EditorGUI.java#L75

Daniele Zecchini

- Uso della libreria JTS Topology Suite per la gestione dei Boundaries: https://github.com/AR0018/OOP23-platform-craft/blob/b35da7896f43d7639916f56f7ab82dd5fb79bd0f/src/main/java/it/unibo/model/collisions/impl/BoundariesImpl.java#L20

Gabriele Abeni

- Utilizzo di Optional: https://github.com/AR0018/OOP23-platform-craft/blob/b35da7896f43d7639916f56f7ab82dd5fb79bd0f/src/main/java/it/unibo/model/engine/impl/EditorImpl.java#L78
- Utilizzo di lambda e stream frequente. Un esempio è: https://github.com/AR0018/OOP23-platform-craft/blob/b35da7896f43d7639916f56f7ab82dd5fb79bd0f/src/main/java/it/unibo/model/engine/impl/EditorImpl.java#L155

Capitolo 4
Commenti finali

4.1 Autovalutazione e lavori futuri

Alessandro Ronchi

Ritengo che il lavoro da me svolto sia un buon esempio di progettazione Object-Oriented e qualità del codice prodotto. Riconosco, riguardando a posteriori il lavoro svolto che alcuni aspetti del design delle sezioni da me realizzati potrebbero essere migliorati e integrati con pattern di progettazione che avevo considerato, ma scartato per motivi implementativi.
All'interno del gruppo, ho svolto buona parte della progettazione dell'applicazione e ho ricoperto un ruolo di supporto nei confronti degli

altri membri, in particolare per quanto riguarda la risoluzione di bug o problemi ed eventuali consigli sull'approccio implementativo da utilizzare in fase di programmazione.
Mi piacerebbe in futuro estendere le funzionalità del progetto perché ritengo che sia sulla strada giusta per diventare una buona applicazione.

Simone Carpi

Questo progetto, complessivamente, è stato impegnativo perché si può considerare il mio vero e primo grande progetto visto che non mi era mai capitato di formare un piccolo team di sviluppatore dediti ad un unico obiettivo comune. Si può considerare una grande esperienza formativa che mi ha permesso di capire come sarà il mondo del lavoro. Inizialmente è stato difficile progettare il design perché bisognava capire cosa effettivamente si volesse creare e soprattutto il modo in cui farlo. Nel corso del tempo mi sono abituato al ritmo di un programmatore e mi sono anche divertito perché è stimolante avere qualcosa di difficile da risolvere. Riguardo alla mia parte mi posso considerare abbastanza soddisfatto, avrei potuto fare di più o mi sarei potuto concentrare maggiormente in alcune parti se il gruppo fosse stato coeso. All'interno del progetto mi sono dedicato alle entità del gioco, parte del controller e gran parte della view. Il mio punto di forza (la parte in cui mi sento più fiducioso) potrebbe essere la realizzazione della trappola perché mi piaceva il concetto. La parte più complicata a mio avviso è stata la GUI. Mi sento di aver dato una mano nel progetto e dall'altra parte sono riuscito un po' a migliorare il mio stile di programmazione grazie all'analisi statica.

Daniele Zecchini

A causa di diverse problematiche personali non mi è stato possibile lavorare alla mia parte del progetto se non negli ultimi giorni prima della consegna, questo ha comportato diverse carenze all'interno del codice e ad un lavoro molto approssimativo.

Gabriele Abeni

Mi posso considerare alquanto contento del lavoro svolto durante questo

periodo di studio intenso visto che mi sono dovuto ricontrollare tutto ciò che era stato fatto nel corso del trimestre. L'inizio è stato faticoso e frustrante non essendo abituato a progetti così grossi.

4.2 Difficoltà incontrate e commenti per i docenti

Alessandro Ronchi

Il corso fornisce una buona preparazione per quanto riguarda la progettazione Object-Oriented. Tuttavia, ritengo che alcuni degli argomenti trattati dovrebbero ricevere una attenzione maggiore di quella che è stata loro riservata durante lo svolgimento del corso. In particolare, ritengo che la sezione sugli aspetti avanzati del linguaggio Java dovrebbe ricevere una maggiore attenzione (in particolare per quanto riguarda l'utilizzo di Stream e generici). Inoltre, anche gli argomenti più pratici trattati in laboratorio (e fondamentali per il corretto svolgimento del progetto) dovrebbero, a mio avviso, essere trattati con maggiore chiarezza.

Appendice A
Guida utente

Funzione play

Cliccando su "PLAY" nel menu principale, l'applicazione aprirà un file chooser dal quale si potrà scegliere il livello da giocare. L'unico formato accettato per un file è .json, quindi qualsiasi file senza tale estensione non sarà aperto. Si raccomanda di ridimensionare la finestra una volta che questa si è aperta per una corretta visualizzazione del livello

Funzione Editor

Cliccando su "EDITOR" nel menu di start, l'applicazione aprirà la finestra di editor. Una volta che la finestra si è aperta, si

raccomanda di ridimensionarla per una corretta visualizzazione dell'editor.

Nell'Editor, è possibile selezionare un oggetto da aggiungere al livello cliccando su uno dei pulsanti sulla destra. Dopo aver cliccato su uno degli elementi di gioco, è possibile cliccare sul livello per aggiungere tale elemento.

Per rimuovere un oggetto, cliccare su "REMOVE" e poi cliccare sull'oggetto che si vuole rimuovere. La funzione di rimozione rimane attiva finché non si seleziona un altro oggetto dalla lista di pulsanti.

L'editor non consente di inserire un oggetto nella stessa posizione di un altro, per impedire che le immagini degli oggetti si sovrappongano.

Dopo aver creato un livello, lo si può salvare cliccando su "SAVE". Se la configurazione del livello è valida, un file chooser verrà aperto. Dal file chooser, l'utente può scegliere dove salvare il file. Il nome del file creato DEVE OBBLIGATORIAMENTE contenere l'estensione .json, altrimenti il livello non verrà salvato.

Una configurazione del livello è considerata valida se il livello ha un Personaggio (Character) e un traguardo (End). Non è possibile aggiungere più Character al livello e lo stesso vale per il traguardo.

Nell'editor è anche possibile editare un livello esistente cliccando su "LOAD" e scegliendo il livello da aprire.

Il tasto "RESET" cancella tutti gli oggetti nella schermata dell'editor.

Livello demo

Al primo avvio dell'applicazione, questa installerà nella home directory dell'utente una cartella denominata "PlatformCraft", contenente il livello di prova. Questo livello può essere caricato, giocato e modificato.

Comandi

Nel corso di una partita, il giocatore può muovere il personaggio usando i tasti WASD oppure le freccette. Il tasto BARRA SPAZIATRICE ha la stessa funzione della freccetta su e del tasto W, cioè quella di far saltare il personaggio.

Appendice B
Esercitazioni di laboratorio

B.0.1 alessandro.ronchi7@studio.unibo.it

- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=147598#p209362
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=149231#p211509
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212744