Relazione per "Pac-Man Hero"

Nikolas Guillen, Mattia Mondin, Angelo Parrinello,
Giacomo Romagnoli, Albi Spahiu

25 giugno 2020

Indice

Capitolo 1

Analisi

L'applicazione mira alla realizzazione di un remake di Pacman, il noto gioco sviluppato in primis da Namco in Giappone negli anni '80 da Toru Iwatani. Il gioco appartiene al genere Action/Arcade.

1.1 Requisiti

L'applicazione presenta all'avvio un menù principale con le seguenti opzioni: "Start Game", "Create Map", "Ranking", "Quit".

- La modalità di gioco è single player ed è ambientata in un labirinto composto da muri fissi. Il personaggio potrà muoversi per la mappa "mangiando" le pillole; alcune pillole sono definite "PowerPill" poichè consentono a Pacman, per un periodo limitato di tempo, di mangiare i fantasmi che si disperderanno a loro volta nella mappa. Lo scopo del gioco è mangiare tutte le pillole sul campo di gioco.
- Il software avvierà una mappa standard, a meno che non ne sia selezionata una diversa.
- Nella sezione "Ranking" l'utente potrà visualizzare una classifica basata sul punteggio registrato dai giocatori precedenti.
- Il punteggio salirà ogni qualvolta Pacman mangerà pillole, PowerPill e fantasmini.
- Nella finestra "Create Map", sarà possibile disegnare una mappa a proprio piacimento, su cui poter poi giocare.
- "Quit" sarà il bottone presente nel menù che permetterà al giocatore di chiudere l'applicazione.

Requisiti non funzionali:

- Pacman dovrà garantire una buona giocabilità e portabilità.

1.2 Analisi e modello del dominio

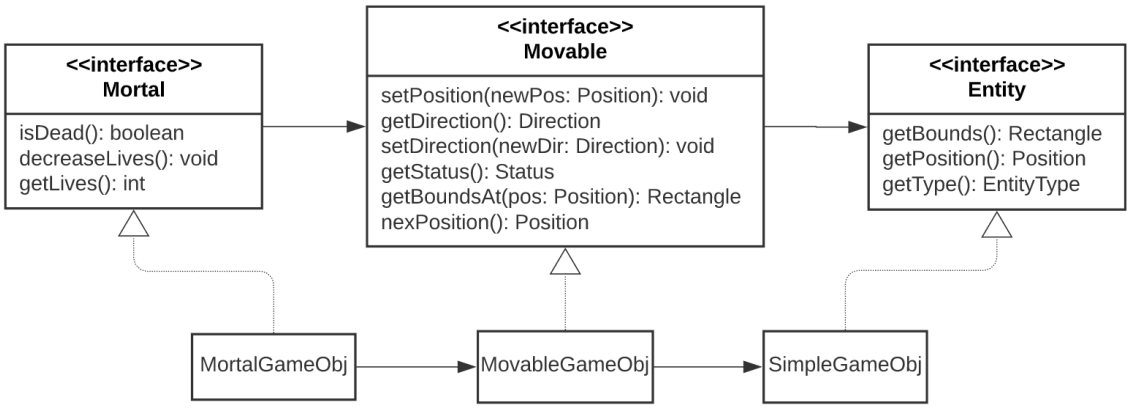

Figura 1.1: L'UML delle Entity

All'interno del gioco coesistono entità passive ed attive. Lo schema descrive una generica entità di gioco passiva come Entity, essa ha caratteristiche generiche quali: una posizione nello spazio, uno spazio occupato nel mondo bidimensionale e un tipo che distingue le diverse entità generiche l'una dall'altra. Con il termine passivo si intende un oggetto di gioco immobile, nella fattispecie pillole, super pillole e muri. Una prima specializzazione di Entity è Movable. Sono Movable tutte quelle Entity che hanno, nella logica del gioco, la capacità di muoversi. Grazie a questo talento chiave, questa tipologia di entità acquisisce un ruolo attivo nel mondo di gioco, questo è rappresentato da uno stato modificato dalle varie interazioni con o tra le altre entità. Per interazione si intende collisione, questa infatti è l'unica modalità con quale i vari oggetti di gioco interagiscono fra di loro; gli effetti delle varie collisioni dipendono dal tipo di entità che entrano a contatto e potrebbero alterare il comportamento o lo stato di terze entità non coinvolte nella collisione stessa. Come specializzazione di Movable troviamo Mortal. Le entità Mortal hanno le stesse caratteristiche dei Movable con l'unica differenza che possono scomparire definitivamente dal gioco. Le funzionalità collegate con la natura effimera delle mortal entity comprendono la possibilità di essere "danneggiate" e ad un certo grado di danneggiamento morire. Il danneggiamento è sempre conseguenza di una collisione. Approfondendo il concetto di tipo

di entità, si ha che per ogni collisione il tipo delle due entità ne definisce gli effetti, agendo sul comportamento delle entità attive. Altri elementi che compaiono nel modello, come ad esempio la posizione, lo status, ecc., sono intuitivi e non necessitano di ulteriori approfondimenti. Le problematiche che sorgeranno da questa visione del progetto riguarderanno i movimenti delle entità Movable, in quanto sarà necessario diversificarli per entità appartenenti a tipologie diverse (es. fantasmini e pacman), inoltre risulterà ostica la gestione delle collisioni fra entità e la loro risoluzione, comprendendo i vari effetti, grafici e non.

Capitolo 2

Design

2.1 Architettura

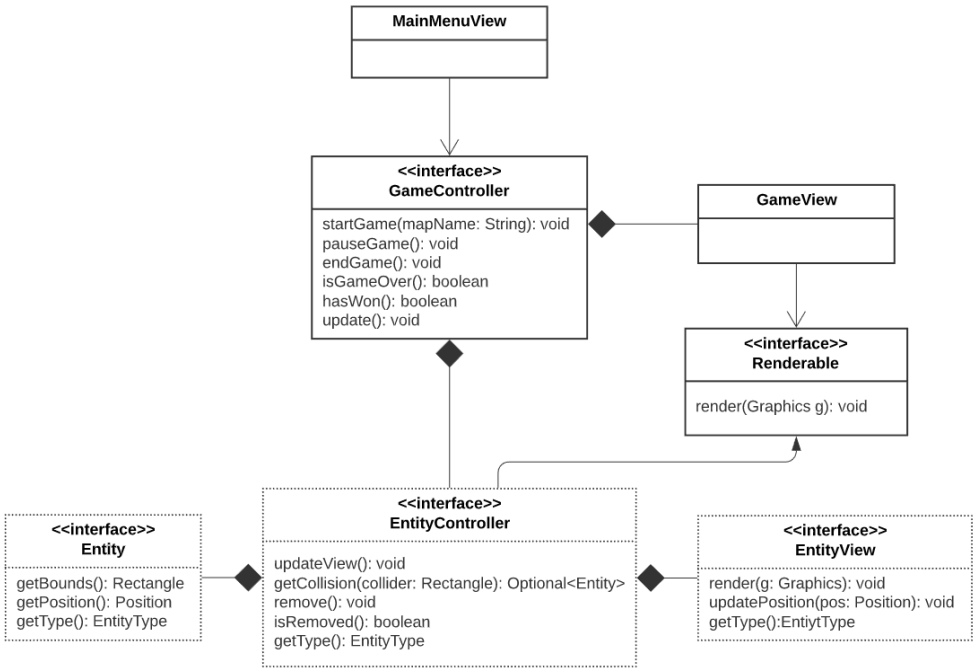

Figura 2.1: UML relativo al MVC

Il progetto sfrutta il pattern architetturale MVC, suddividendo le classi in tre categorie di appartenenza: model, view e controller. In particolare il modello, descritto nella sezione precedente, costituisce la base del progetto, esso però non è concepito per essere completamente autonomo, bensì le

classi di model contengono funzionalità minimali che necessitano di coordinazione per performare vere azioni complesse. Prendiamo come esempio il movimento delle entità mobili; un Movable sa calcolare la prossima posizione e sa impostarne una nuova, ma per performare un vero e proprio movimento dovrà, come minimo, impostare la sua posizione attuale alla prossima posizione che ha calcolato. Questo ruolo di "coordinatore" è affidato alle classi che compongono la C di MVC, i controller. Quindi nella nostra interpretazione la parte dei controllori assume un duplice ruolo, il primo è quello di far comunicare modello e vista, come da pattern, mentre il secondo è quello di coordinare le entità, sia fra di loro, come nel caso delle collisioni, sia con loro stesse per performare azioni complicate, siccome spesso queste ultime dipendono dalle collisioni che avvengono. Entrando nel dettaglio, le collisioni vengono gestite facendo comunicare fra di loro i vari controller, quindi se il controllore del pacman necessita di verificare se nel suo prossimo movimento incontrerà un muro, semplicemente delegherà il compito al controllore del muro che risponderà, rendendo noto a chi lo ha invocato se la collisione è avvenuta o meno. Infine la parte di view comprende classi che hanno come unico compito quello di rendere visibile ogni entità del gioco. Ognuna di queste classi è collegata ad un oggetto di model e coordinata da un controller, quest'ultimo si occupa di aggiornare la vista dopo ogni cambiamento all'interno del gioco. Ricapitolando, ogni oggetto di model è associato ad un oggetto di view tramite un controller che fornisce tutti i servizi necessari per utilizzare l'oggetto di gioco e renderlo visibile; è evidente che in questa descrizione manca coordinazione fra le varie terne di model, view, controller. Per risolvere tale problematica si è deciso di creare un "GameController" e una "GameView". GameController si occupa di coordinare tutti i controllori decidendo in che ordine essi effettueranno le varie azioni all'interno del gioco ed infine comunicherà alla GameView ciò che va visualizzato a schermo dopo che è stato aggiornato tutto il mondo di gioco. Oltre alla appena descritta struttura del progetto, compaiono diverse altre interfacce per gestire l'input/output, per esempio riguardo al caricamento della mappa.

2.2 Design dettagliato

Mattia Mondin
Mi sono occupato di sviluppare la gestione del loop di gioco, il GameController, che è servito a creare un punto utile all'interazione delle parti del gioco, lo score in-game, gli effetti sonori e parti minori riguardanti l'acquisizione delle immagini.

Ho creato la parte del GameLoop, che si occupa di gestire la continuità del gioco, attraverso la creazione di un loop. Questa fase è stata gestita tramite l'uso di un Thread, implementato alla classe GameLoopImpl. L'obbiettivo di questa classe era di scandire i tempi di aggiornamento a ogni componente del gioco ad ogni frame, sia nel model che nella view.

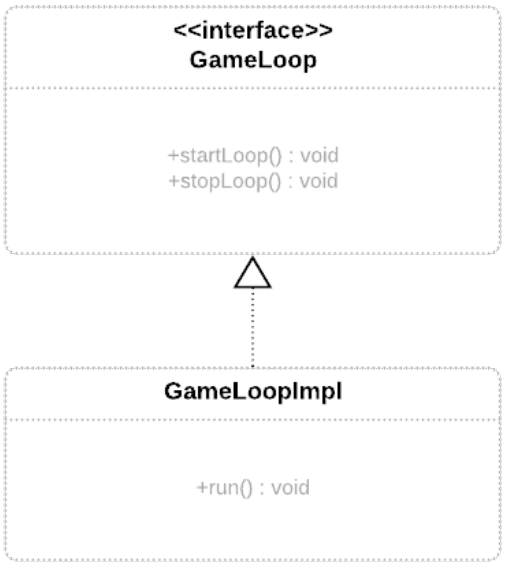

Figura 2.2: Immagine relativa al UML del GameLoop

Ho poi creato il GameController, il quale si occupa di gestire il gioco, dall'avvio fino al completamento dell'applicazione. In particolare, richiama il terreno di gioco e coordina i vari avvenimenti. Per quanto riguarda la gestione delle collisioni, è stato deciso verso la fine del progetto di annetterlo alle parti del progetto riguardanti Pacman e Ghost, discostandolo di più dalle funzionalità di gioco. Quindi il GameController richiederà un aggiornamento del gioco tramite il metodo update, che richiamerà i metodi implementati nelle classi responsabili.

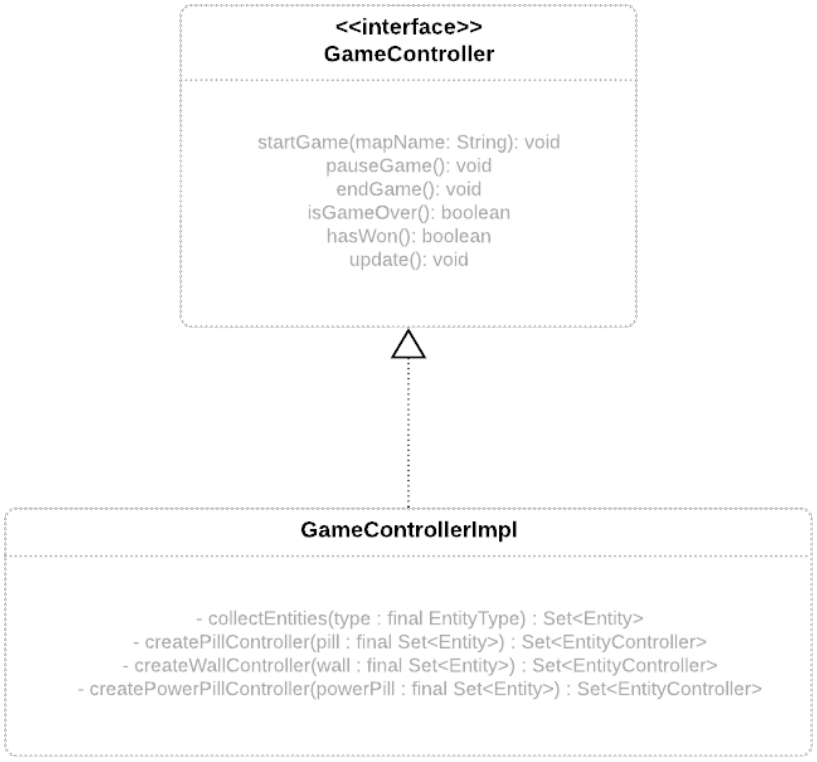

Figura 2.3: Immagine relativa al UML del GameController

In relazione alla gestione dei punteggi, ho creato Scoring, PlayerScore e la sua implementazione, PlayerScoreImpl. Scoring è una Enum che definisce un valore per ogni differente entità mangiata da Pacman (spiegato nella prima parte della relazione) da utilizzare per il calcolo della classifica. L'aggiornamento del punteggio viene effettuato da PlayerScoreImpl tramite updateScore. La scrittura su file avviene al termine della partita, non affrontata da me.

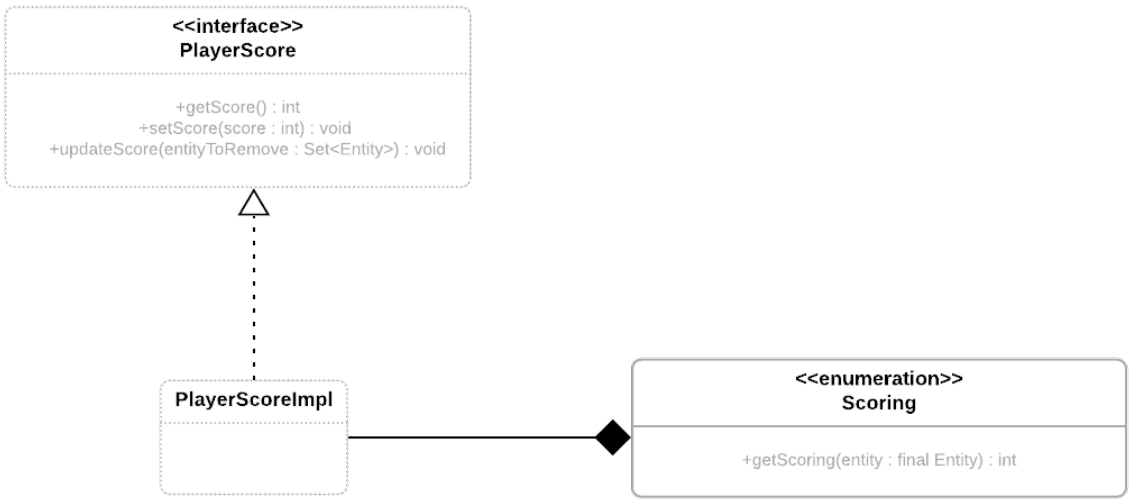

Figura 2.4: Immagine relativa al UML dello Score in-game

La parte sugli aspetti grafici, tramite BufferedImageLoader, ha riguardato la modalità di acquisizione dell'immagine che verrà poi utilizzata dalla Mappa per caricare il gioco e dalle Entità per cambiare l'immagine a seconda dell'evento avvenuto.

Figura 2.5: Immagine relativa al UML del BufferedImageLoader

Ho infine realizzato la parte Sound (nella quale utilizzo la classe javax.sound.sampled.Clip) che mi permette la riproduzione in loop di un suono senza ulteriori controlli sullo stato di quest'ultimo nel GameLoop. Successivamente ho realizzato la classe SoundController per la gestione dei vari effetti sonori, creati come Optional. La classe Optional ha funzionalità che rendono possibile l'acquisizione di un oggetto vuoto, senza creare errori dovuti al null. Inoltre, i metodi getter che restituiscono gli effetti sonori, sono stati implementati come statici, per permettere il loro utilizzo nelle altre classi senza la necessità di istanziare molteplici volte SoundController.

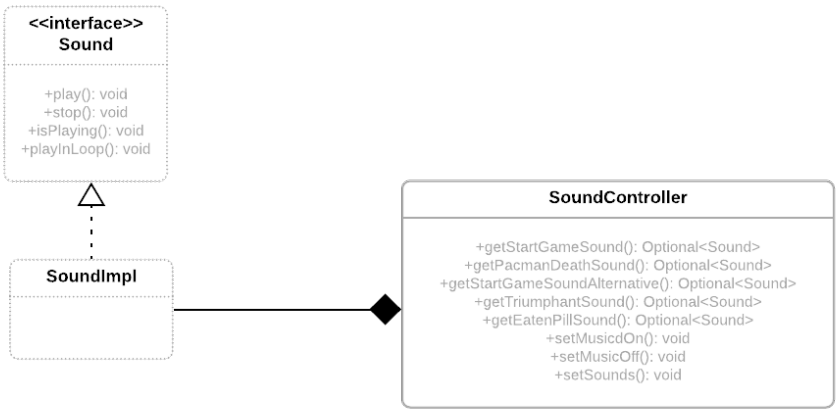

Figura 2.6: Immagine relativa al UML dei Suoni

Giacomo Romagnoli

Uno dei dilemmi in fase di progettazione è stato decidere in che modo evidenziare le differenze fra entità di gioco. La logica descritta in fase di analisi è rimasta invariata, mentre dopo varie sperimentazioni sono arrivato alla conclusione che le differenze fra entità di tipologia diversa erano quasi inesistenti. Basta pensare a due oggetti come pillola e muro: concetti completamente opposti ma con comportamenti simili. Per questo nel progetto esiste una sola implementazione per categoria di entità, ovvero Entity, Movable e Mortal. Questo però poteva risultare controintuitivo e poco chiaro, quindi per nascondere il più possibile alle altre parti del progetto l'ambiguità implementativa, ho deciso di fornire una factory (nella sua forma interfaccia-implementazione) che si occupi di creare tutti i vari oggetti di gioco, rispettando la logica dell'information hiding e distaccando l'implementazione dal concetto a cui fa riferimento.

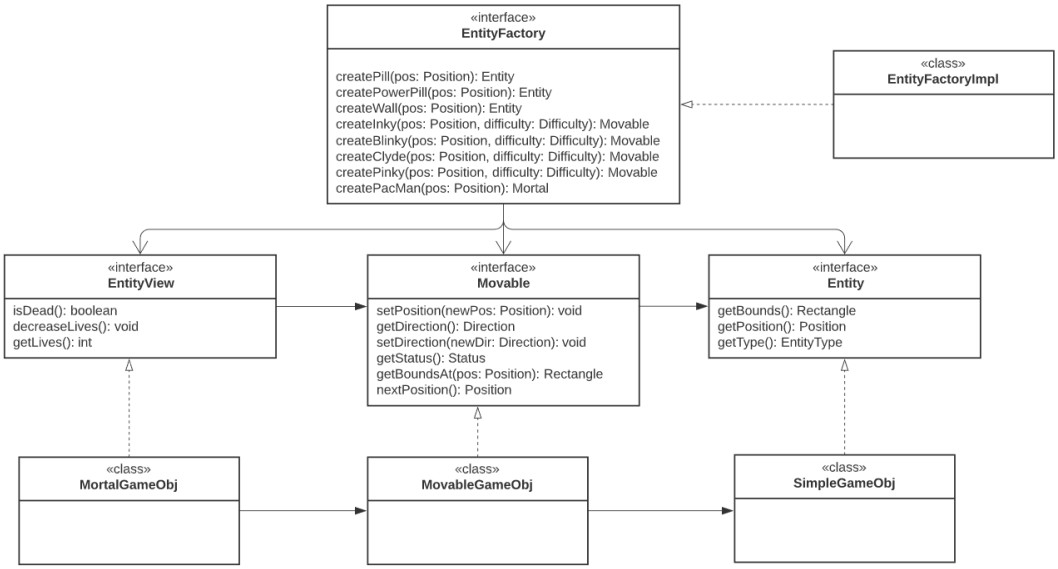

Figura 2.7: L'UML della EntityFactory

Per quanto riguarda invece la sezione dei controller, ho notato che fantasmi e pacman avrebbero avuto molte similitudini, soprattutto sul movimento. Perciò ho applicato il pattern decorator per lasciare aperta la possibilità di creare oggetti di gioco mobili differenti fra di loro, infatti nel dominio in discussione capita spesso di dover solo "decorare" un oggetto già esistente per crearne uno nuovo. Non avendo, per il momento, entità mobili diverse da pacman o dai fantasmi (che già rappresentano una decorazione) è stata creata un' astrazione che rappresenti il MovableController più generico, il SimpleMovableController, che viene poi decorato dando vita a PacmanControllerImpl e GhostControllerImpl.

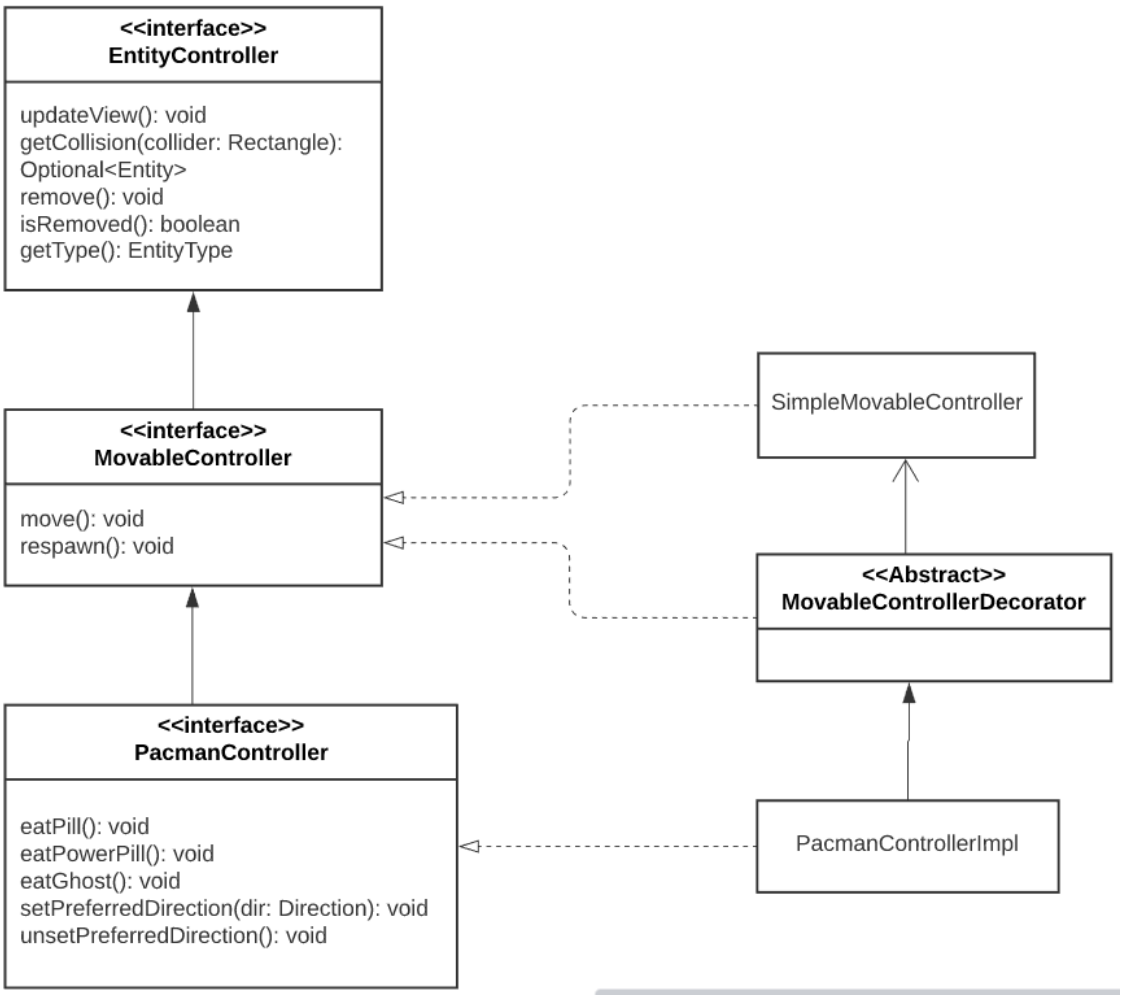

Figura 2.8: An image of DecoratorUML

Per spendere una parola in più sugli aspetti progettuali dei controllori, è facile notare che esiste una correlazione molto forte con le entità, però la necessità di performare le azioni tipiche del personaggio Pacman mi ha portato a preferire una specializzazione di MovableController, quale PacmanController. Per azioni tipiche si intende la possibilità di mangiare le altre entità, con tutti gli effetti che comporta. La soluzione che ho scelto prevede di sfruttare strategy, vedendo PacmanController come cliente dei metodi getCollision, fear e remove contenuti in EntityController e GhostController. Ragionamento simile vale per PacmanController e PlayerScore, salvo per i metodi che compongono la strategia, in questo caso, solo il metodo updateScore di PlayerScore.

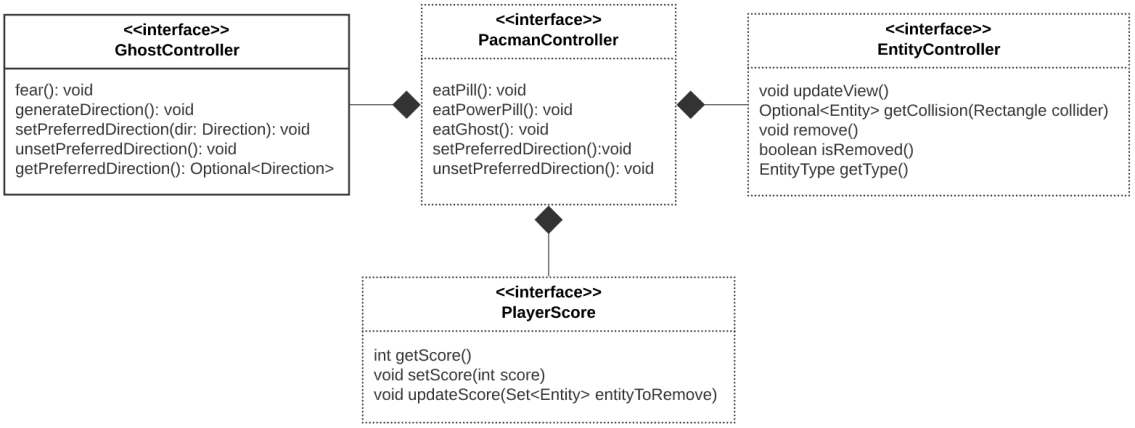

Figura 2.9: An image of StrategyUML

Angelo Parrinello

Mi sono occupato principalmente della gestione e della logica dei fantasmini, in tutte e tre le sue componenti: model, view e controller.

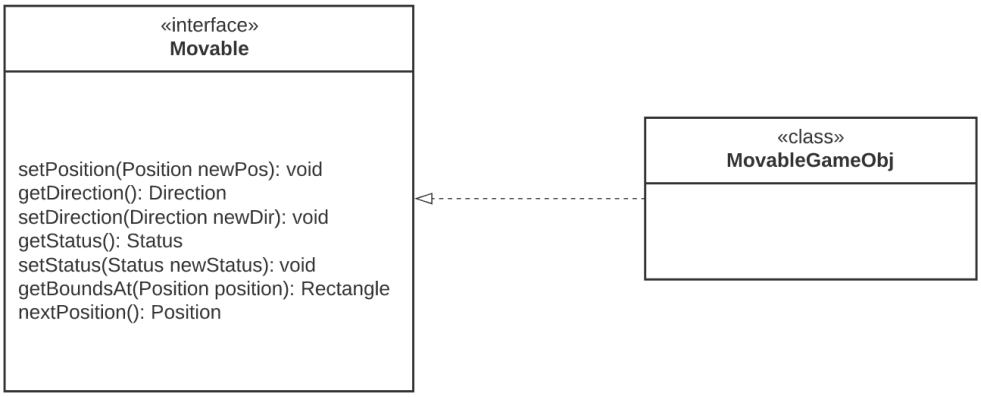

Figura 2.10: L'UML del model del Ghost

La prima fase di analisi e progettazione era molto complicata da svolgere in autonomia, in quanto fin dalle prime riunioni di gruppo ci siamo resi conto che molte parti, sarebbe stato meglio progettarle assieme a Giacomo Romagnoli, che si è occupato del Pacman. Insieme abbiamo progettato la nostra parte di Model, in particolare, io mi sono dedicato in autonomia all'interfaccia Movable e la sua implementazione MovableGameObj. Quest'ultima classe è proprio un'implementazione adatta ad un Entity che si può muovere, ma che non può morire, quindi, un fantasma.

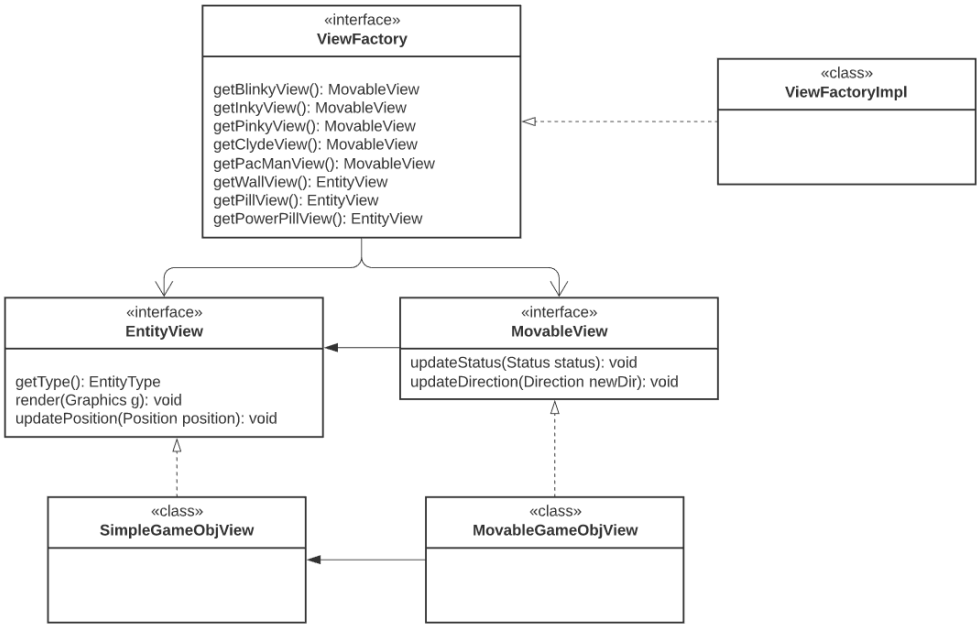

Figura 2.11: L'UML della Factory delle View

Per quanto riguarda la progettazione delle view, ho notato importanti somiglianze tra Mortal e Movable. Questo, mi ha portato a scrivere due interfacce: una per le Entity, EntityView con annessa implementazione, e una per le Mortal/Movable, MovableView più implementazione. Durante la gestione delle View, invece, ho riscontrato la necessità di distaccare le implementazioni della vista dal loro concetto logico. Per questo, ho deciso di usare il pattern Factory nella classe ViewFactory, così facendo sono riuscito a rispettare le norme dell'information hiding , inoltre ho reso più agevole l'utilizzo delle classi di view per altri parti del progetto come il GameController. La ViewFactory andrà a creare una view per ogni singolo elemento di gioco.

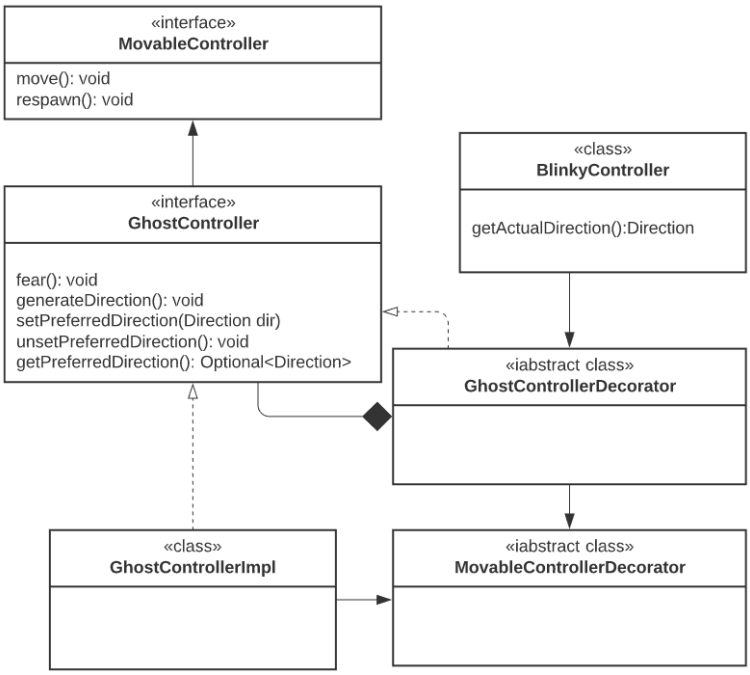

Figura 2.12: L'UML del Decorator sul GhostController

Analizzando la realizzazione dei controller, ho notato che ogni fantasma aveva peculiarità molto simili, ciò che li distingueva era il movimento. Infatti, anche nel cabinato degli anni 80', Blinky e Co. avevano caratteristiche diverse relative allo spostamento all'interno del labirinto. La mia scelta è stata, quindi, di utilizzare il pattern Decorator, in modo tale da decorare in maniera agevole i metodi di base, utilizzati nei movimenti, a seconda delle necessità. L'interfaccia che verrà decorata è la GhostController. Ad oggi è stato implementato il movimento di Blinky attraverso il BlinkyController, che decora GhostControllerImpl (che è essa stessa generata da una decorazione), ovvero un'implementazione generica per il controllo di un fantasma. In futuro si potranno creare facilmente specializzazioni più specifiche per ciascun controller dei fantasmi.

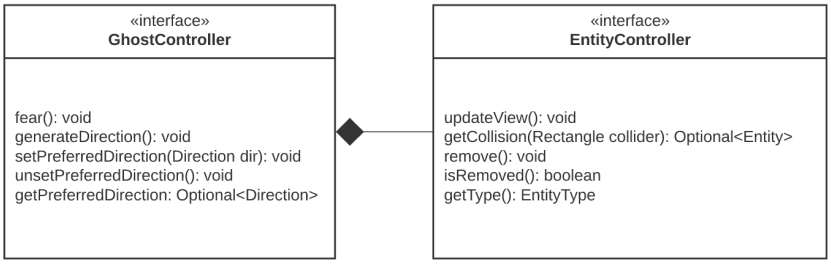

Figura 2.13: L'UML dello Strategy sul GhostController

Infine, ho avuto la necessità di specializzare la MovableController con azioni tipiche dei fantasmini, così ho creato l'interfaccia GhostController. Di fatti, i fantasmi hanno bisogno anche di: "spaventarsi" quando il Pacman mangia una Superpillola e avere dei metodi a supporto della generazione pseudo-randomica delle Direzioni. Quest'ultima verrà generata una volta che il fantasma impatta un muro. A tal proposito, ho deciso anche di sfruttare il pattern Strategy utilizzando come cliente del pattern, il GhostController, sul metodo getCollision (metodo utilizzato nel controllo delle collisioni tra i muri e il fantasma), contenuto nell'interfaccia EntityController.

Nikolas Guillen

La mia parte di progetto consisteva nella creazione della mappa di gioco, della leaderboard e del menu di fine partita.
Per quanto riguarda la parte di Controller della mappa, mi sono occupato del processo di creazione a partire dalla lettura del file di testo, per proseguire poi con la produzione delle varie entità. Ciò è stato realizzato tramite la classe LoadMap che, mediante l'utilizzo di una classe di utility MapIO, itera il file di testo generando gli elementi di gioco attraverso l'utilizzo di una Entity Factory realizzata da Giacomo Romagnoli. Per realizzare la View ho invece adottato il pattern Builder per rendere più snella e comprensibile la costruzione dell'oggetto; inoltre, per una maggiore coerenza stilistica del gioco, sia questa che le altre View sfruttano una GUIFactory (con relativa implementazione GUIFactoryImpl) adibita alla creazione dei vari JComponent utilizzati.
Dopo un'analisi iniziale del progetto, mi sono reso conto che tutte le View avevano in comune dei metodi atti a mostrare/nascondere il frame. Per questo motivo ho realizzato l'interfaccia ViewableUI, che definisce i metodi Show e Close utilizzati da tutte le GUI del gioco. Ho inoltre creato l'interfaccia RenderableUI, che estende ViewableUI e definisce in più il metodo Render: questa viene implementata da GameView.

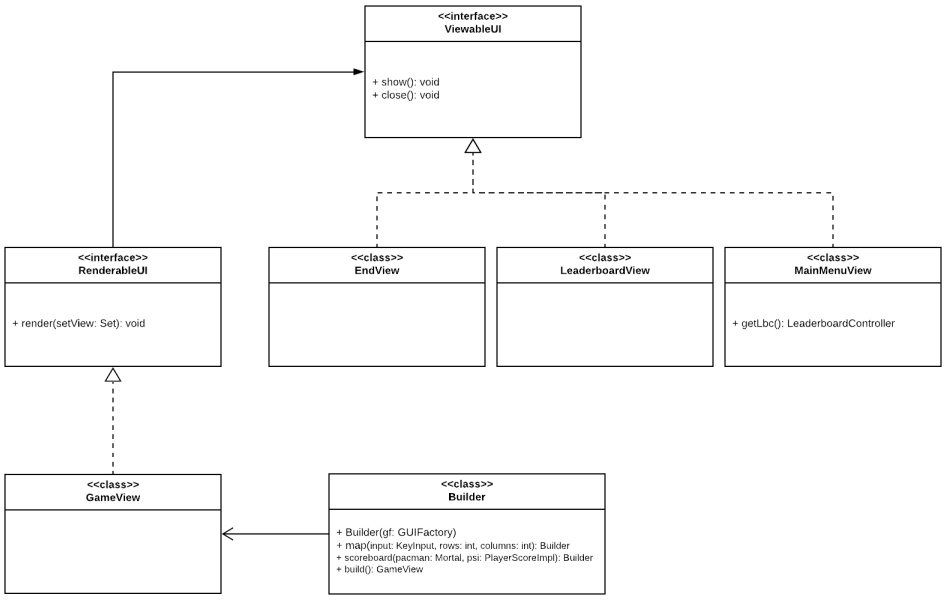

Figura 2.14: Gerarchia delle View del progetto

La classe EndView si occupa della creazione di una GUI che permetta al giocatore di visualizzare il punteggio finale e di salvarlo assieme al proprio nome utente, liberamente inseribile in questa stessa interfaccia grafica. La classe adotta il pattern Adapter per adattare il metodo privato di salvataggio su file ad ActionListener del tasto "Submit". Il punteggio viene scritto su file con una chiamata al LeaderboardController.

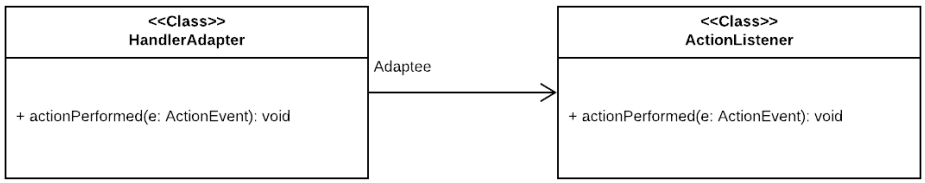

Figura 2.15: Utilizzo di Adapter per l'actionlistener del tasto Submit

La Leaderboard è stata realizzata seguendo il principio MVC, con un Controller che si occupa di fare da tramite tra la parte di Model e quella di View della Leaderboard. Questo Controller legge la classifica da file tramite la classe LeaderboardIO e successivamente la passa ad un nuovo oggetto Leaderboard. Quest'ultimo si occupa di ordinarla, per poi passarla come parametro di ritorno.

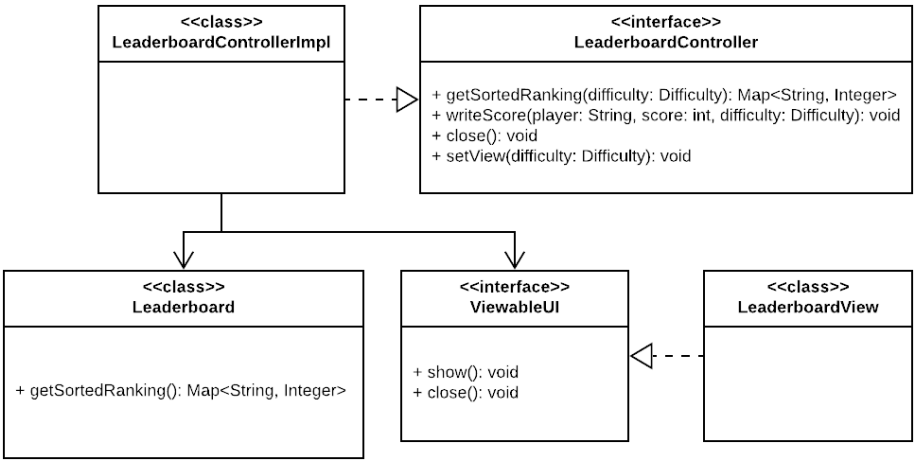

Figura 2.16: Gestione MVC della Leaderboard

Successivamente ho creato alcune classi di Utility come BackImagePanel, PacFont, LeaderboardIO e il metodo readMap della classe MapIO.

Albi Spahiu

Mi sono occupato principalmente della realizzazione dell'editor delle mappe, del loro salvataggio e del menù iniziale di gioco.

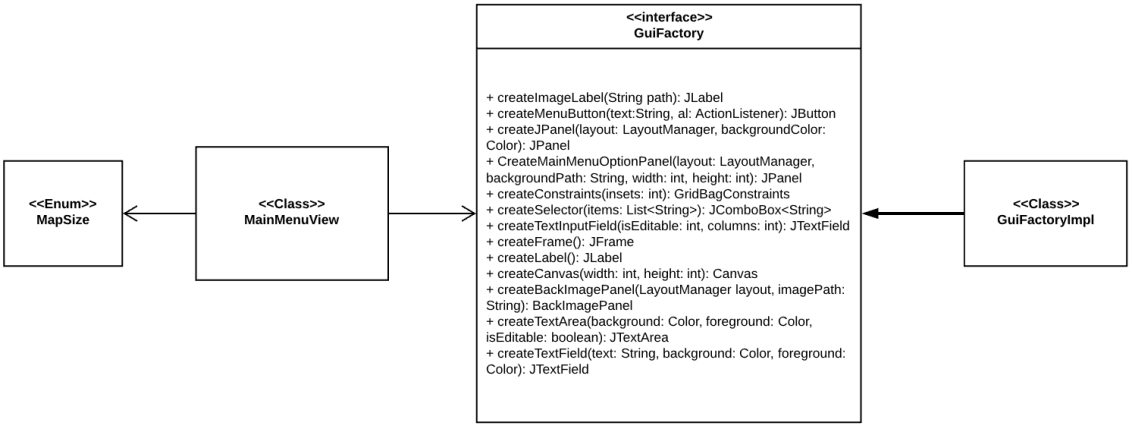

Figura 2.17: UML sulla GUI Factory

Per la realizzazione del menù iniziale di gioco ho inizialmente implementato l'interfaccia ViewableUI creata da Nikolas Guillen. Insieme si è poi proceduto alla creazione di una Factory per la creazione dei vari JComponent presenti nelle nostre varie view. Si è dunque proceduto alla creazione dell'interfaccia GUIFactory e della relativa implementazione GUIFactoryImpl. Per il menù ho deciso di creare JPanel intercambiabili, uno per ciascuna delle varie configurazioni di scelte possibili. Per lo "Start Game", viene fatta scegliere la mappa di gioco, la scelta di Music On/Off e la difficoltà Normal/Hard; per la "Create Map" viene chiesta la dimensione della mappa; mentre per il "Ranking", quale classifica aprire.

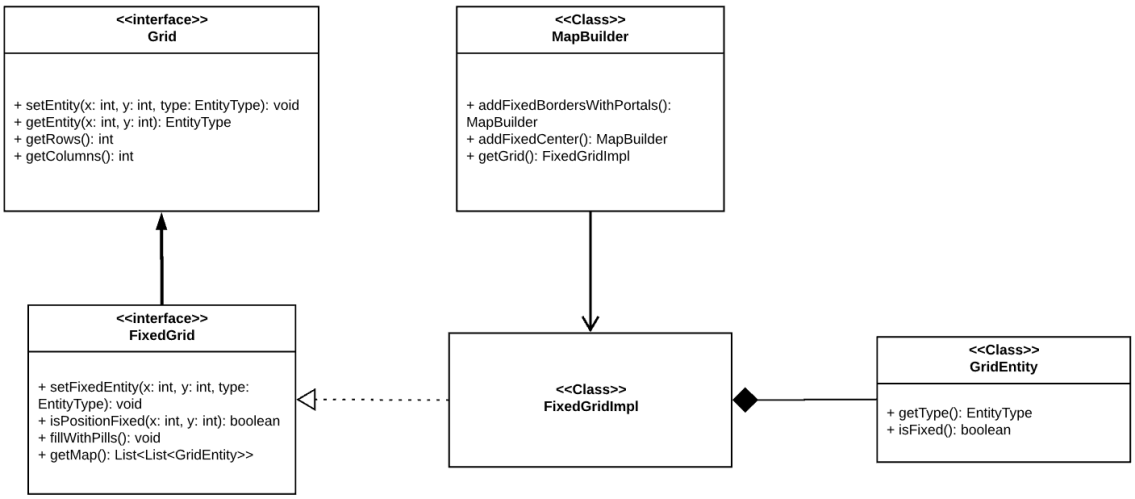

Figura 2.18: UML Model del map editor con Builder

Per quanto riguarda l'editor delle mappe ho deciso di realizzare un'interfaccia che modellasse una comune griglia sulla quale fosse possibile posizionare una determinata entità di gioco. FixedGrid estende questa interfaccia e permette inoltre di gestire entità "fisse". Questa scelta si è rivelato necessaria per impedire che un errata creazione della mappa potesse provocare situazioni di stallo durante la partita. Un centro nel quale vengono fissati sia il pacman che i fantasmini permette infatti di garantire sempre almeno una fine (pacman mangiato 3 volte). Per gestire queste entità è stata creata la inner class GridEntity, che permette di stabilire attraverso il metodo isFixed() le entità fisse da quelle modificabili. Per la creazione di queste mappe è stato utilizzato il pattern Builder, con il quale è possibile aggiungere in maniera flessibile alla mappa bordi, portali e centro.

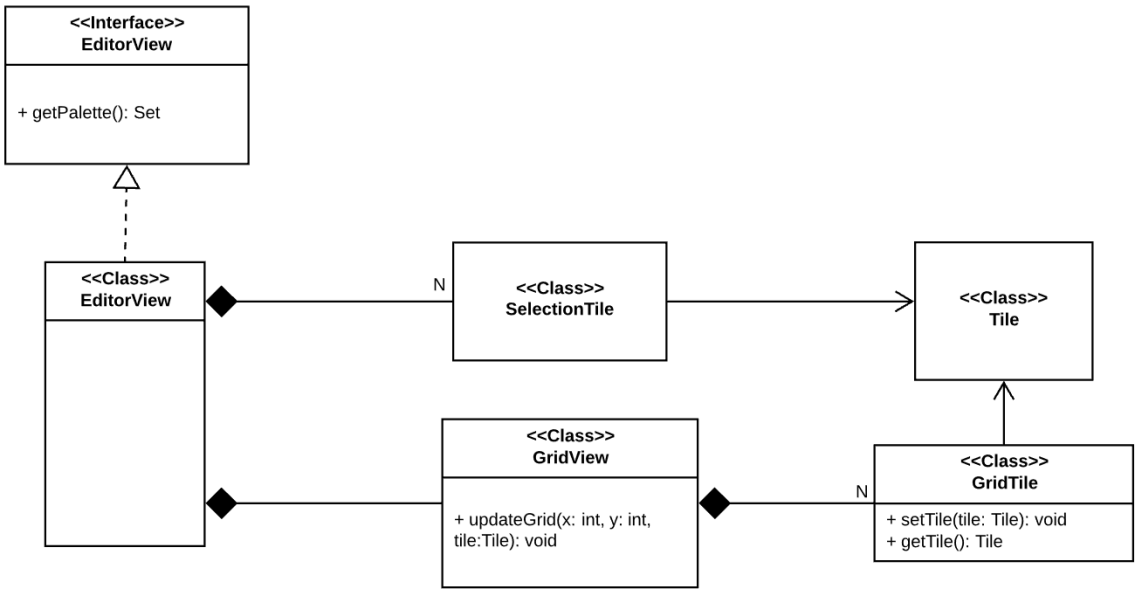

Figura 2.19: UML View del map editor

Per la View dell'editor, ho iniziato implementando la classe Tile, la quale modella un semplice tassello dell'editor. I tasselli della mappa, cosi come i bottoni della "Palette delle entità", utilizzano questa classe per reperire le informazioni base, come ad esempio l'entità da mostrare. La palette, dalla quale è possibile selezionare l'entità da aggiungere alla mappa, è posizionata nella parte superiore dell'editor; mentre subito sotto è posizionata la mappa, rappresentata dalla classe GridView. Questa viene aggiornata dal Controller ad ogni nuova iterazione dell'utente, attraverso la classe GridListener.

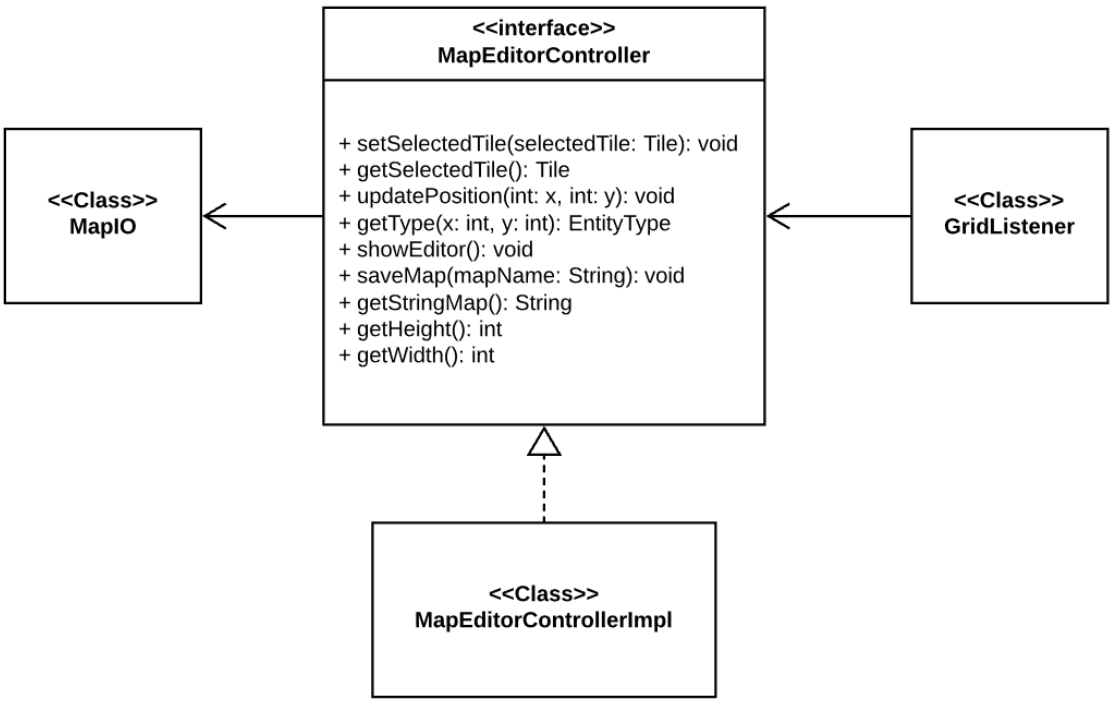

Figura 2.20: UML Controller del map editor

Il coordinamento tra Model e View è svolto dal MapEditorController/ MapEditorControllerImpl. È qui che viene aggiunta alla GridView il GridListener, che determina, in base al click del mouse, dove vadano posizionate le entità, richiamando i metodi del controller per aggiornare View e Model. Il controller si occupa anche di effettuare il salvataggio della mappa, attraverso scrittura su un file .txt delle varie entità (identificate dal numero presente nella EnumType) in base alla loro posizione nella mappa.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Gran parte delle problematiche sul funzionamento dell'applicazione erano riscontrabili solo in casi limite molto difficili da ricreare in un ambiente automatizzato, per questo abbiamo optato per una verifica manuale del corretto funzionamento del gioco, avvenuta in sessioni di testing/debug con la partecipazione del team al completo. Nello specifico, nelle sessioni di testing è stato creato un branch test sul quale sono stati corretti i malfunzionamenti grafici del programma. Anche se minimali sono presenti alcuni test JUnit sviluppati per: gestione dei fantasmi (TestGhost), gestione del Pacman (TestPacman), creazione delle mappe (testGrid) e infine logica della Leaderboard (testLeaderboard).

3.2 Metodologia di lavoro

La fase di analisi e la successiva di design architetturale sono state svolte in gruppo. Ci siamo aggiornati periodicamente attraverso Microsoft Teams per gran parte del lavoro, per definire i requisiti del software e aggiornarci sul lavoro di ognuno in modo da procedere nel modo più chiaro possibile.

- Mattia Mondin: realizzazione del GameLoop, implementazione del GameController, del punteggio in game, inserimento degli effetti sonori e implementazione di un buffer utile al prelievo delle immagini da caricare
- Giacomo Romagnoli: gestione del pacman in tutti i suoi aspetti, tradotto nell'implementazione di MortalGameObj, SimpleGameObj, EntityType, SimpleGameObjView, SimpleMovableController, MovableCon-

trollerDecorator e PacmanControllerImpl. Creazione della EntityFactory.

- Angelo Parrinello: Progettazione dei fantasmi nella sua interezza. Nel dettaglio, le implementazioni di: Movable, MovableView, EntityController, GhostController, GhostControllerDecorator, BlinkyController e BlinkyDirectionGenerator. Realizzazione delle Enum di Model Utility, Status e Direction. Gestione della creazione delle view con ViewFactory.

- Nikolas Guillen: gestione della Leaderboard a 360 gradi, ovvero nei suoi tre aspetti di Model, View e Controller. Implementazione della LoadMap per creare le varie entità in gioco e della GameView per la loro renderizzazione in tempo reale. Realizzazione della schermata di fine gioco EndView e di alcune classi di utility come LeaderboardIO, PacFont, BackImagePanel e il metodo readMap in MapIO.

- Albi Spahiu: gestione dell'editor delle mappe, a livello di Model, View e Controller, del salvataggio delle mappe con la classe MapIO e della enum MapSize per le dimensioni di default delle mappe. Gestione del menu principale e della GUIFactory.

Ciascun membro del gruppo ha sviluppato, quando possibile, le proprie parti in tutti e tre gli aspetti di MVC. Al fine di permettere un'ottima integrazione tra tutte le parti sviluppate in autonomia, il controller di gioco è stato modificato da vari elementi del gruppo e discusso nelle fasi di sviluppo. Altre sezioni sono frutto di forti collaborazioni, il package model.entities e view.entities sono perfetti esempi, nonostante la formale divisione dei compiti, di fatto, Giacomo Romagnoli e Angelo Parrinello hanno contribuito in maniera complementare per la realizzazione dei due package.

3.3 Note di sviluppo

- Mattia Mondin:
  - Optional nella classe SoundController. Questo aspetto è stato scelto in quanto è una componente di gioco variabile, scelta dall'utente prima di avviare il gioco. Gli effetti sonori che ho utilizzato sono stati scaricati dal sito open-source OrangeFreeSounds
  - Uso delle Lambda e degli Stream, in particolare nella classe GameControllerImpl

    Per comprendere come effettuare il caricamento dei suoni, argomento non trattato a lezione, ho fatto uso di snippet di codice dal progetto Bomberman2018.
- Giacomo Romagnoli
  - Optional nel tipo di ritorno del metodo getCollision. Nel metodo getCollision l'incertezza dell'avvenuta collisione è rappresentata dal valore di ritorno opzionale.
  - Uso di Optional per la direzione preferita di pacman. La preferredDirection è banalmente un aspetto opzionale del movimento del pacman che potrebbe non essere presente in alcuni momenti del gioco.
  - Uso massivo di Stream e lambda expressions soprattutto in PacmanControllerImpl.
- Angelo Parrinello
  - Accenno di Multi-Threading in BlinkyDirectionGenerator e MyTimer.
  - Uso di Optional in GhostControllerImpl.
  - Utilizzo di Stream e Lambda Expression in buona parte delle mie classi implementate.
- Albi Spahiu
  - Uso di StringBuilder e inner class nella FixedGridImpl.
  - Inizializzazione non statica nella classe Tile per il caricamento delle immagini nella EnumMap.
  - Uso di lamba, Optional e Collections.

- Nikolas Guillen
  - Scrittura e lettura da file nelle classi LeaderboardIO e MapIO. In quest'ultima ho realizzato solo il metodo readMap.
  - Utilizzo degli stream nel metodo Render della GameView e nell'ordinamento dei dati della classifica presa dal file di testo. Questo secondo stream è stato reperito online (https://dzone.com/articles/how-to-sort-a-map-by-value-in-java-8).
  - Classe PacFont creata prendendo come base la classe GameFont del progetto bmbman del 2019.

Capitolo 4

Commenti finali

Data la totale inesperienza del gruppo, in un progetto così esteso, è stato necessario informarsi su Youtube e con materiale di terze parti, esterne alle dispense fornite a lezione. La più grande sfida è stata senza dubbio la progettazione del codice e la coordinazione nel lavoro di gruppo. Il tutto è stato accentuato dall'emergenza Covid19, che ha influito non solo sull'organizzazione generale, ma anche sulla condizione psico/fisica.

4.1 Autovalutazione e lavori futuri

- Mattia Mondin: Questo progetto mi ha permesso di capire quali sono gli aspetti positivi e negativi di uno sviluppo di gruppo. Sono soddisfatto di aver saputo rispondere all'opportunità del lavoro su un progetto non semplice, nella quale è stato richiesto da parte nostra una documentazione approfondita tramite materiale reso disponibile su BitBucket, Youtube e una comunità online, che spesso trova difficoltà nelle medesime problematiche che abbiamo riscontrato. Il risultato finale dell'elaborato mi soddisfa, in quanto ha una buona fluidità ed è stato realizzato con un buon approccio alla programmazione a oggetti. La fase di sviluppo mi ha procurato non poche difficoltà, avendo una parte particolarmente astratta come il GameController. Lo sviluppo del gioco da parte del gruppo non ha visto un buon lavoro passo passo, secondo la mia opinione, che ha reso buona parte del mio lavoro una fase di aggiornamento continuo per rendere più chiara la direzione della mia classe, della quale non mi è stato possibile ottenere una versione parziale. Questa situazione mi ha costretto a una documentazione più profonda, attraverso il materiale e sopra citato. Sono comunque contento perché documentandomi e cercando di capire le parti del programma,

che ognuno ha gestito in modo autonomo, attraverso una costante comunicazione da parte mia, il risultato della classe è efficiente. A causa di un ricalcolo del lavoro durante il progetto e una differente implementazione delle collisioni a fine progetto, mi sono reso conto che il mio carico di lavoro era leggermente inferiore rispetto a quello dei miei compagni, per questo ho implementato nell'ultimo periodo la funzionalità di punteggio in-game e gli effetti sonori. Concludo dicendo che credo ognuno si sia profondamente interessato allo sviluppo del gioco, nonostante la situazione di emergenza e gli impegni diversi di ognuno, e che il risultato sia un buon gioco, con non poche potenzialità di implementazione.

- Giacomo Romagnoli: Sono soddisfatto del lavoro da me portato a termine e anche di quello del mio gruppo nella sua interezza. Questo è stato per tutti il primo progetto di grandi dimensioni e non nascondo che ha rappresentato una grossa sfida per me, ma nonostante le difficoltà riscontrate credo di aver raggiunto un buon risultato. Detto questo, penso che il margine di miglioramento di Pacman Hero sia grande, le idee non sono mancate al team in ogni fase e sono certo delle potenzialità del nostro operato. Per il futuro sono già state gettate le basi per nuove versioni del software che potrebbero rendere il classico Pacman un arcade più frizzante e moderno, oltre che un'ottima palestra per l'ingegneria del software.

- Angelo Parrinello: A fine di questo percorso, posso dire di essere soddisfatto del mio lavoro. Il Pacman creato ha rispettato le nostre aspettative, e ciò vuol dire che il gruppo intero ha lavorato bene ed equamente. In particolare, sono rimasto appagato dal movimento dei fantasmi. Inizialmente questi, si muovevano in maniera molto banale, ora sono leggermente più intelligenti e in particolare una prima specializzazione del controller, BlinkyController, permette al fantasma di cambiare direzione non solo quando sbatte contro il muro. Avrei, in tutta sincerità, voluto implementare anche un movimento davvero furbo dei fantasmi attraverso l'algoritmo di Dijkstra, ma ciò non è stato possibile per difficoltà implementative. Ritengo, infine, che il progetto sia stato un ottimo allenamento anche per quanto riguarda lo studio dell'Ingegneria del Software e che volendo, si potrebbero aggiungere elementi di gioco e grafici, senza troppi problemi.

- Nikolas Guillen: Credo che questo progetto si sia rivelato una preziosa esperienza per migliorare le nostre capacità di programmazione e lavoro coordinato. Nonostante le difficoltà dovute alla quarantena, ritengo che siamo riusciti a realizzare un lavoro piuttosto buono. Io in particolare ho acquisito una visione molto più ampia e chiara di come ci si muova in un vero team di sviluppo.

- Albi Spahiu: Sono complessivamente soddisfatto del lavoro svolto sia da me che dai miei compagni. La mia parte di progetto, sconnessa dalle dinamiche di gioco, non ha richiesto un elevato coordinamento con i miei colleghi, se non riguardo al salvataggio (e la lettura) delle mappe. Questo mi ha permesso molta libertà sulle scelte implementative, ma è anche mancato un confronto sulle varie decisioni prese a livello logico e architetturale. Non sono contento del menu principale, che andrebbe riprogettato per evitare la creazione di tutti i JPanel nei vari metodi privati. Ciò ha appesantito il codice, non aderendo ai principi di buona programmazione. Mi reputo invece contento delle altre classi e della loro progettazione. Mi piacerebbe in futuro poter modificare l'editor con l'aggiunta di una camera, cosi da non avere limiti sul numero di righe e colonne.

4.2 Difficoltà incontrate e commenti per i docenti

- Giacomo Romagnoli: In generale la sfida più grande è stata quella legata alla fase di progettazione, infatti durante il corso sono state gettate solamente le basi necessarie per affrontare questa sfida; oltre a questo enpass iniziale ho trovato molto difficile documentarmi sulla creazione di un videogioco, aspetto della programmazione molto oscuro per lo studente medio, è stato infatti necessario spendere molto tempo per capire come procedere nello sviluppo del software. Già dai primi incontri di gruppo ci si è reso conto che il progetto avrebbe richiesto molte più ore di quelle previste a lezione, per questo ritengo sia opportuno un ridimensionamento del monte ore richiesto, di modo che risulti più rispondente alle effettive tempistiche.

- Angelo Parrinello: Questo progetto mi ha messo di fronte a delle mie debolezze che, posso dire fieramente, sono stato in grado di superare, ovvero il lavoro coordinato in team e la progettazione di un software. Inoltre ho avuto molte difficoltà iniziali nel capire come gestire collisioni

e movimenti in un gioco 2D. A tal proposito prima di partire a scrivere o progettare il codice, ho dovuto seguire un corso online che mi spiegava come gestire questi aspetti videoludici, andando a ricreare un reale giochino (davvero banale e mal progettato). Ritengo anche che il carico reale di lavoro non corrisponda minimamente alle 80 ore dichiarate. Uno studente che, come me, non si è mai avvicinato alla realizzazione di un gioco non impiegherà mai un monte ore simile, ma molte di più. Penso anche che in una laurea triennale indirizzo Informatico, sia indispensabile un progetto così ampio, ma che vada semplicemente dichiarato che in realtà le ore non sono quello che sono. Un ultimo suggerimento: sarebbe molto utile, avere dispense e/o video-tutorial, atte alla spiegazione di alcuni lati implementativi di un gioco 2D. Come ,ad esempio, la spiegazione della gestione delle collisioni attraverso intersect(), getBounds() e la classe Rectangle.