PACMAN

Relazione per il progetto di Programmazione ad Oggetti (OOP)

Barzi Eddie 0000875148
Belloni Sofia 0000873985
Rossi Davide 0000890282
Vissani Filippo 0000880686

15 maggio 2020

Indice

Capitolo 1

Analisi

1.1 Requisiti

L'obiettivo è quello di realizzare un videogioco ispirato a Pac-Man del 1980. Il giocatore deve guidare Pac-Man all'interno di un labirinto facendogli mangiare il maggior numero di palline possibile, ed al tempo stesso, evitare le ondate di fantasmi.

Requisiti funzionali

- Realizzazione di un menu iniziale che permetta di iniziare il gioco e di accedere alle schermate delle impostazioni, della classifica dei giocatori e delle istruzioni di gioco con schema dei comandi.
- Realizzazione della schermata impostazioni che offra la possibilità di attivare/disattivare il suono, resettare la classifica e cambiare la mappa di gioco.
- Realizzazione di una schermata di "Game Over" con possibilità di inserire il nome che verrà salvato nella classifica.
- Presenza di cinque tipi di fantasmi, che si distinguono per il loro "carattere", cioè il modo in cui si muovono.
- Nei bordi della mappa di gioco sono presenti dei tunnel che portano dall'altra parte della mappa.
- Durante la partita è possibile visualizzare il numero di vite rimanenti, il punteggio totale, il tempo rimanente per il livello corrente e il record attuale.

- Possibilità di mettere il gioco in pausa.
- La classifica riporta i nomi dei giocatori in base al numero di livelli superati e al punteggio totale.

Requisiti non funzionali

- Presenza di effetti sonori ricollegati a determinate azioni.
- Fluidità delle animazioni degli elementi in movimento.
- Realizzazione di un'interfaccia grafica user friendly.

1.2 Analisi e modello del dominio

Le entità all'interno del gioco sono Pac-Man e i fantasmi. Il giocatore deve guidare Pac-Man all'interno di un labirinto facendogli mangiare il maggior numero di palline possibile, ed al tempo stesso, evitare le ondate di fantasmi, i quali aumentano all'aumentare del livello. Mangiando le palline disseminate nella mappa si totalizzano punti, e al raggiungimento di un determinato punteggio la situazione si inverte e Pac-Man può mangiare i fantasmi per un certo lasso di tempo. All'inizio di ogni livello parte un timer, al termine del quale si passa al livello successivo, inizia una nuova ondata di fantasmi e le palline si rigenerano. I fantasmi si differenziano in base al loro comportamento:

- Blinky: è il fantasma più aggressivo, insegue Pac-Man per tutta la durata del gioco, cercando sempre di raggiungere il punto esatto in cui si trova, ma solo dopo aver raggiunto l'angolo in alto a destra della mappa.
- Pinky: meno aggressivo rispetto a Blinky; diversamente da quest'ultimo, non insegue direttamente Pac-Man, ma cerca di prevedere dove si muoverà per anticiparlo e catturarlo, ma solo dopo aver raggiunto l'angolo in alto a sinistra della mappa.
- Inky: la sua direzione si basa su due fattori fondamentali: la posizione di Pac-Man e quella di Blinky; all'inizio di ogni livello raggiunge l'angolo in basso a destra della mappa;
- Clyde: se Pac-Man si trova nel raggio di sette caselle, si rifugerà nella parte in basso a sinistra dello schermo, se invece è lontano, inizierà a dargli la caccia, mentre all'inizio di ogni livello raggiunge l'angolo in basso a sinistra della mappa.

- Randy: i fantasmi che non vengono mangiati si trasformano in Randy nei livelli successivi, e assumono un comportamento imprevedibile. Sono distinguibili dal loro colore verde.

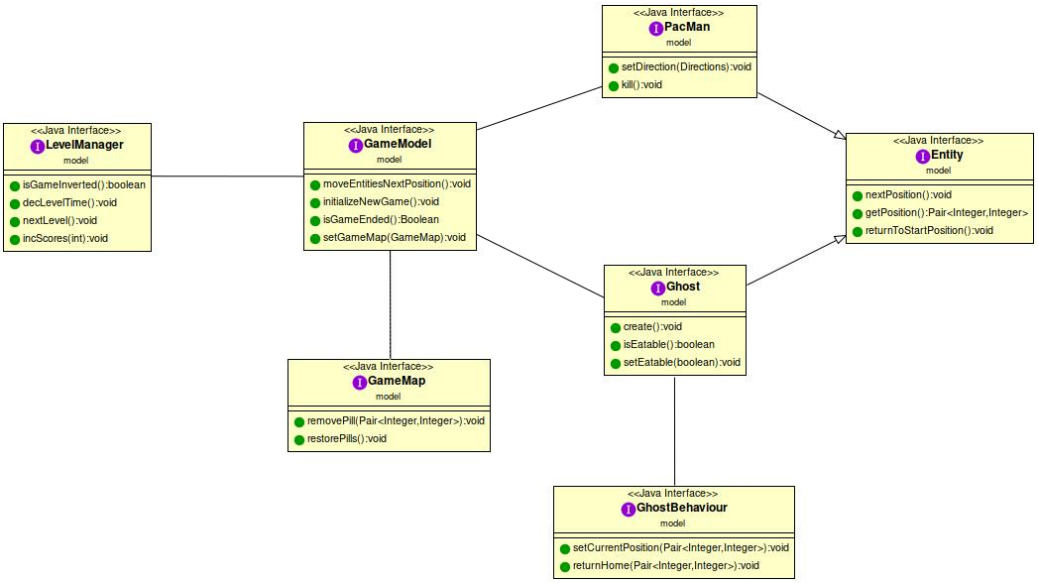

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro.

Capitolo 2

Design

2.1 Architettura

Per la realizzazione di questo software si è scelto di utilizzare il pattern architetturale Model-View-Controller(MVC). In questo modo si è potuto suddividere la logica funzionale in:

- **Model**: si occupa della logica di gioco, della gestione delle diverse entità, delle loro interazioni, della mappa e del livello di gioco. Il model è indipendente dal controller e dalla view, infatti, come mostrato in figura, non ha nessun riferimento alle altre componenti.
- **Controller**: il controller è il componente che collega model e view, ed ha il compito da un lato di aggiornare periodicamente il model e renderizzare i cambiamenti avvenuti, dall'altro trasmettere gli input dell'utente al model; grazie al controller si rimuove completamente la dipendenza tra model e view favorendo il riutilizzo delle componenti.
- **View**: gestisce la parte grafica del gioco e l'interazione con l'utente, si occupa inoltre di mostrare a video lo stato delle entità e della mappa esattamente come sono nel model.

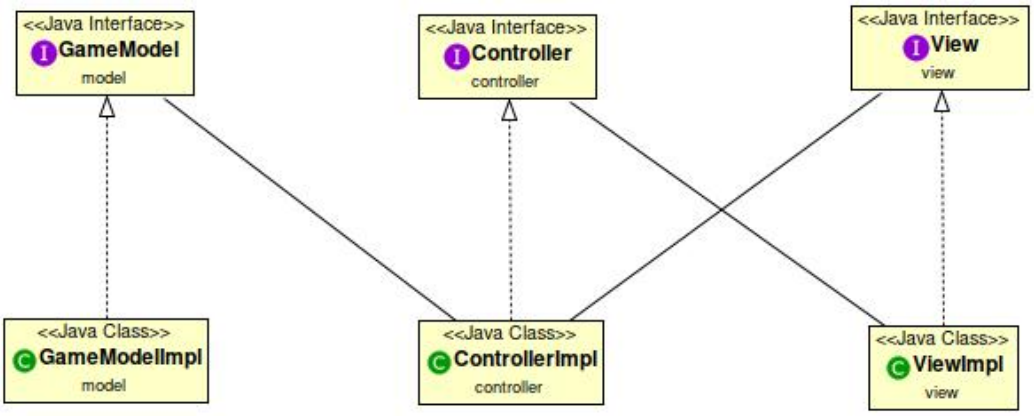

Figura 2.1: Schema UML del pattern MVC.

2.2 Design dettagliato

Davide Rossi

Fantasmi

Per la realizzazione dei fantasmi ho subito pensato ad una classe astratta che raggruppasse le similarità tra di essi, e successivamente una Simple Factory che gestisse la loro creazione. Per alleggerire il codice ho scelto di creare i fantasmi tramite delle classi anonime all'interno della factory, e qui ho iniziato ad avere problemi relativamente ai campi che andavano aggiornati, per esempio posizione e direzione. Ho pensato quindi di suddividere l'implementazione delle caratteristiche dei fantasmi da quelle dei loro behaviour, sfruttando il principio di decomposizione.

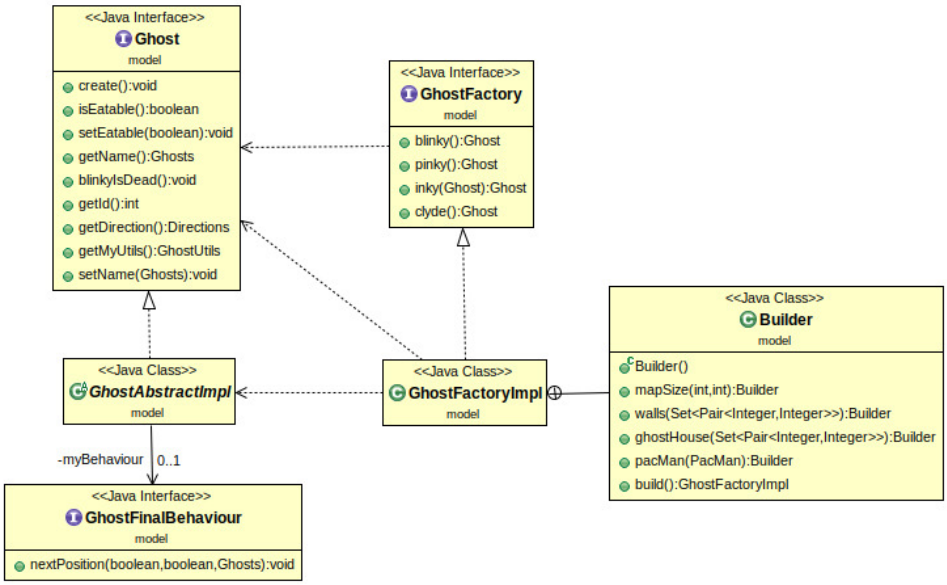

Figura 2.2: Rappresentazione UML dello schema creazionale dei fantasmi

Il contratto dei fantasmi è definito dall'interfaccia Ghost, che va ad aggiungere metodi a quelli già definiti dall'interfaccia Entity che estende, per esempio quelli relativi alla gestione dello stato dei fantasmi, che possono essere mangiabili o meno. Avendo la stessa interfaccia ed essendo molto simili tra loro, ho ritenuto necessario raggruppare il codice comune dei fantasmi in una classe astratta GhostAbstractImpl. All'interno di essa viene anche dichiarato il comportamento generico GhostFinalBehaviour, che rappresenta l'interfaccia Strategy per la scelta a runtime dell'algoritmo, in base al fantasma che viene creato. La factory concretizza la creazione dei fantasmi, utilizzando un metodo per ciascuno di essi in cui viene creata una classe anonima. La mancanza di costruttore delle classi anonime ha posto la necessità di controllare ad ogni invocazione di alcuni metodi il controllo della avvenuta creazione, tramite il metodo checkCreation(). Vista la generosa dimensione del costruttore, per migliorare la comprensibilità e la creazione della factory ho deciso di utilizzare il pattern creazionale Builder.

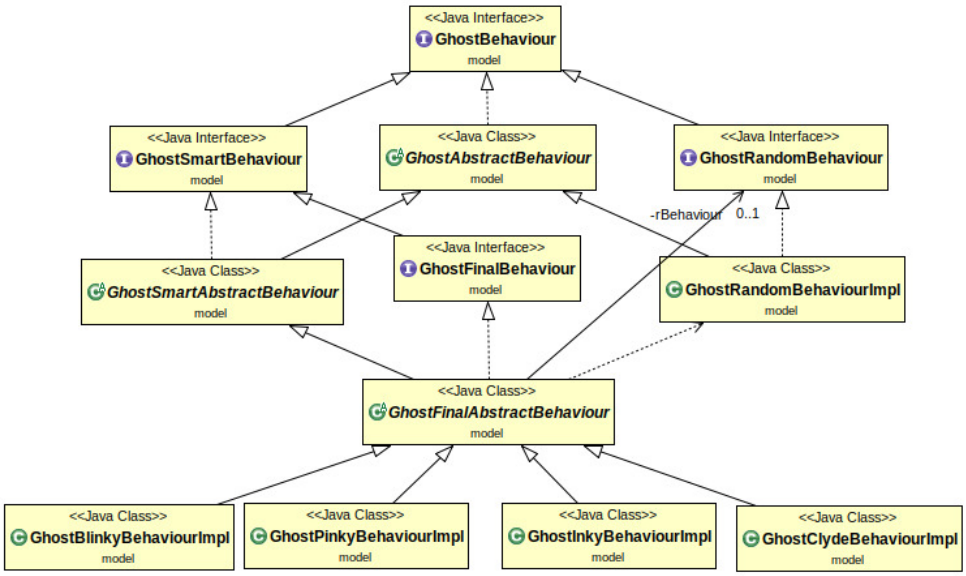

Figura 2.3: Rappresentazione UML dello schema generale del comportamento dei fantasmi

Riguardo la modellazione comportamentale, ho iniziato a progettare un'unica classe astratta per gestire i vari stati dei fantasmi, concentrandomi sul funzionamento corretto degli algoritmi. Raggiunto questo obiettivo, mi sono reso conto che avrei avuto bisogno di sfruttare l'ereditarietà multipla per gestire i comportamenti dei fantasmi in classi astratte differenti e in maniera ottimale, e a tal scopo ho trovato molto utile la simulazione di ereditarietà multipla presente nelle slide del corso.

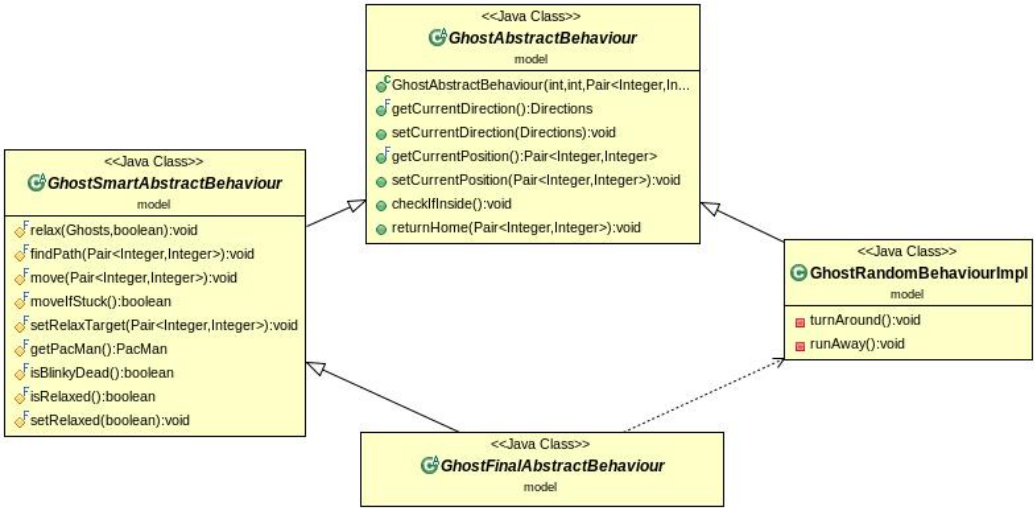

Figura 2.4: Rappresentazione UML dello schema astratto del comportamento dei fantasmi

Ogni fantasma ha due lati comportamentali, il lato Smart che utilizza per raggiungere un target ben definito, e il lato Random che consiste nel movimento casuale nella mappa, i quali estendono la classe GhostAbstractBehaviour che mantiene i metodi in comune a entrambi. La classe GhostFinalAbstractBehaviour è il soggetto che sfrutta la simulazione di ereditarietà, in quanto eredità la classe Smart e si compone della classe Random.

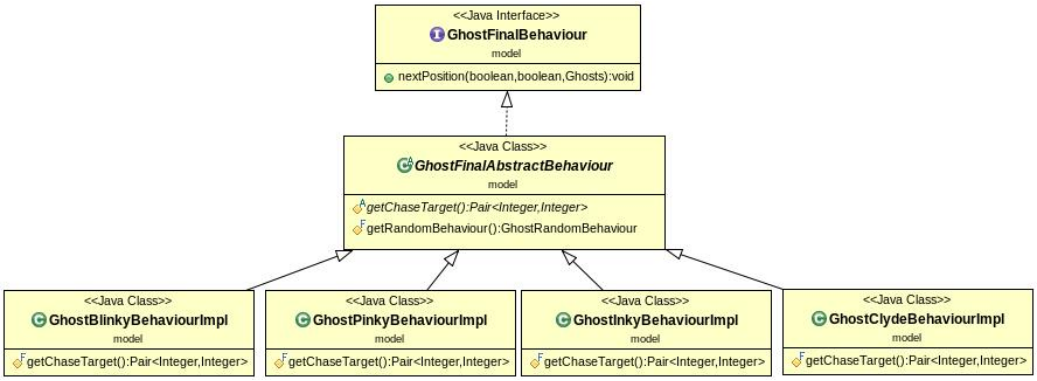

Figura 2.5: Rappresentazione UML dello schema concreto del comportamento dei fantasmi

Il metodo nextPosition della classe GhostFinalAbstractBehaviour è il metodo centrale di tutta l'architettura comportamentale; il suo compito è passare il controllo all'algoritmo appropriato in base allo stato del fantasma.

Questo è di fatto un Template Method, poiché è egli stesso a chiamare il metodo astratto getChaseTarget(), il quale viene ridefinito nelle sottoclassi, che rappresentano il comportamento concreto di ogni fantasma nella fase di movimento verso uno specifico target.

Filippo Vissani

Mappa di gioco

La mappa di gioco è una griglia composta da *tile* che si distinguono tra loro per il tipo (ad esempio *pill* o *wall*) e per la posizione.
Per creare l'oggetto che gestisce la mappa di gioco è necessario specificare la grandezza della mappa in orizzontale e in verticale, il punteggio dato da una singola pillola e le posizioni di muri, pillole, casa dei fantasmi e quella di partenza di Pac-Man.
Un costruttore con tutti questi parametri sarebbe stato molto complicato da utilizzare, quindi ho scelto di adottare il pattern Builder, che semplifica drasticamente il passaggio di parametri in fase di creazione dell'oggetto, favorendo inoltre un'interfaccia "fluent".

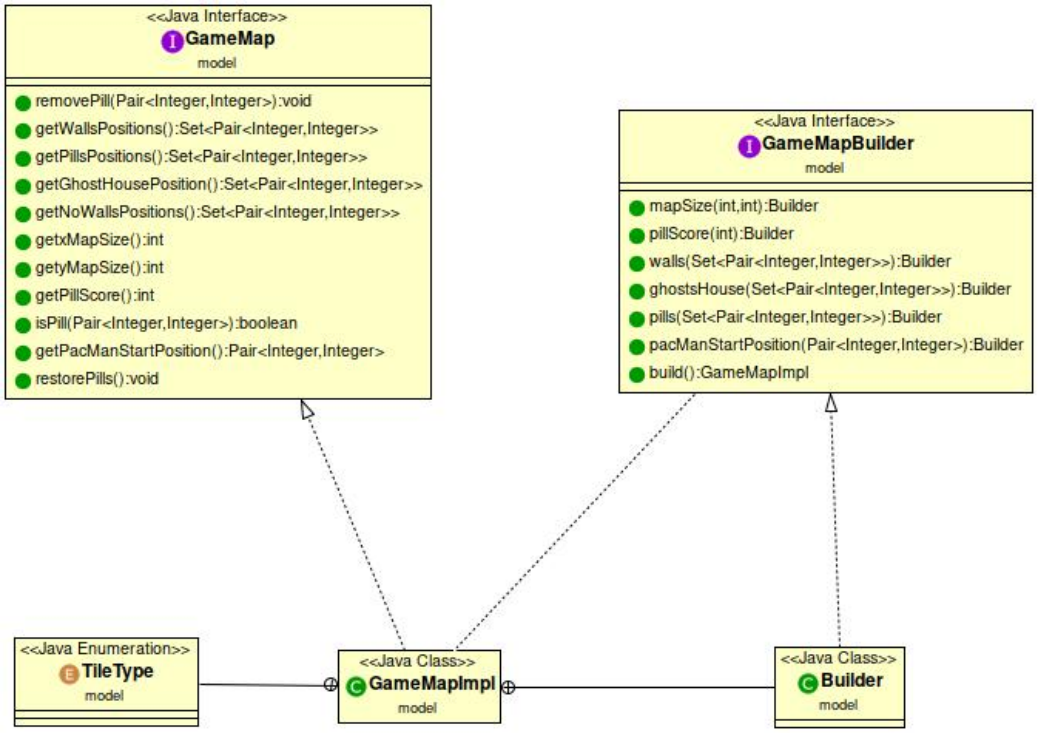

Figura 2.6: Rappresentazione UML del pattern Builder utilizzato per creare la mappa

Durante la partita, ogni volta che Pac-Man passa sopra ad una pillola, questa viene rimossa. Ogni volta che inizia un nuovo livello tutte le pillole che sono state mangiate vengono rigenerate e la mappa ritorna al suo stato iniziale. Nel gioco è anche possibile scegliere una delle mappe tra le varie disponibili. Questa funzionalità è fornita dalla classe GameMapLoaderImpl, che si occupa di caricare ed interpretare il file (strutturato in un determinato modo) che rappresenta la mappa. Sfruttando questo metodo è stato possibile creare mappe differenti senza bisogno di modificare il codice.

Collisioni

Le collisioni sono gestite nella classe Collision e possono avvenire in due modi:

- **Tra Pac-Man e una pillola**: in questo caso viene effettuato un controllo sulla posizione, se quella di Pac-Man coincide con quella di una pillola, allora significa che è avvenuta una collisione.
- **Tra Pac-Man e uno o più fantasmi**: in questo secondo caso il controllo da effettuare è decisamente più accurato. Oltre a replicare lo stesso controllo che viene fatto per le pillole (quindi basato solo sulla posizione), bisogna anche controllare che un fantasma e Pac-Man non

facciano uno scambio di posizioni (ad esempio quando "si guardano" e sono in posizioni adiacenti). Quando avviene questo tipo di collisione, se il gioco è in modalità invertita, vengono rimossi i fantasmi interessati, altrimenti viene tolta una vita a Pac-Man.

Gestione dei livelli e del punteggio
La gestione dei livelli e del punteggio è affidata alla classe LevelManagerImpl. Durante la partita, ogni volta che avviene una collisione tra Pac-Man e una pillola, il punteggio viene incrementato. Viene anche tenuto conto del punteggio parziale, usato per invertire il gioco, in questa modalità Pac-Man può mangiare i fantasmi.
Ogni livello ha una durata di 60 secondi, il tempo viene gestito tramite un timer presente nel controller, che ad ogni secondo si occupa di decrementare il tempo rimanente. La situazione di gioco invertita invece dura 10 secondi, anche in questo caso la gestione del tempo viene affidata al timer.

Sofia Belloni

CONTROLLER
Il Controller, in accordo con il pattern MVC, è il componente che collega Model e View: in particolare, nel nostro caso, deve svolgere i seguenti compiti:

- Meccanismo di aggiornamento di gioco
- Presenta il timer che rappresenta ogni livello
- Salvataggio su file dei risultati ottenuti a fine partita e, di conseguenza, gestione della classifica.
- Lettura da file della mappa di gioco

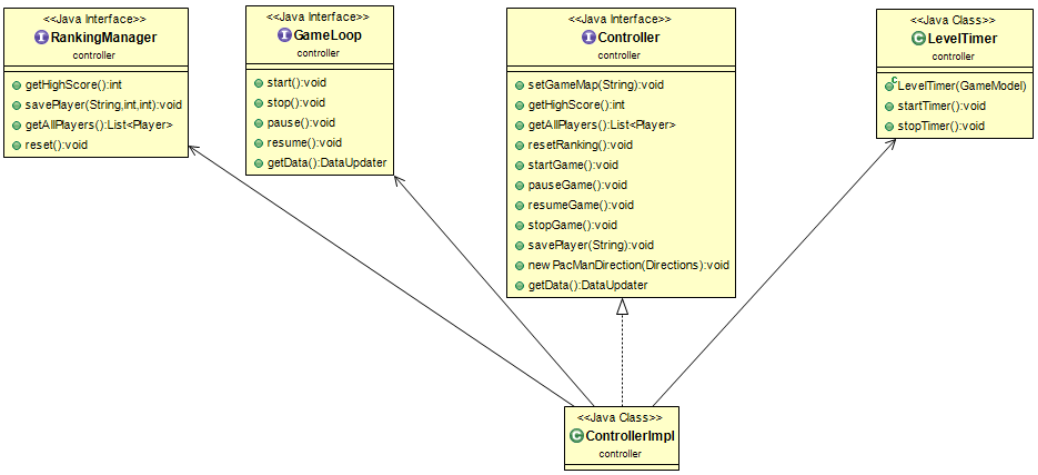

Figura 2.7: Rappresentazione UML del controller

GAMELOOP
Il meccanismo di aggiornamento del gioco è regolato dall'interfaccia GameLoop, che permette di separare la progressione del tempo di gioco dall'input dell'utente e dalla velocità del processore. La classe GameLoopImpl implementa le interfacce GameLoop e Runnable in modo che possa essere eseguita in maniera parallela, in quanto rappresenta quel meccanismo che permette di aggiornare periodicamente Model e, in seguito, renderizzare i cambiamenti avvenuti. Tali aggiornamenti sono in realtà delegati, in accordo con il pattern *Delegation*, alla classe DataUpdater, creata per rendere più pratico l'aggiornamento dei dati mappabili ed il loro passaggio tra model e view nel rispetto del pattern MVC. Questa ha anche il compito di adattare i dati letti dal model a quelli richiesti dalla View, laddove necessario.
E' la View infatti che, dopo aver ricevuto dal Controller la richiesta di aggiornarsi, legge dal Controller stesso i dati di cui necessita. Il compito di quest'ultimo è quello di provvedere a restituire i dati aggiornati.
GameLoopImpl, in quanto implementato come thread separato, deve permettere i comandi start, pause, resume e stop.

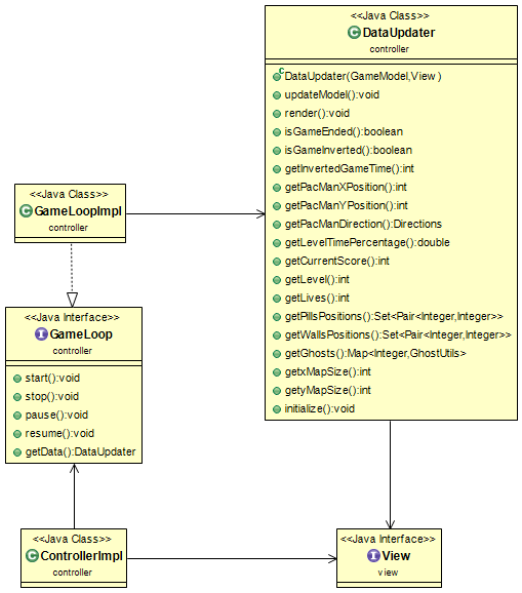

Figura 2.8: Rappresentazione UML che mostra l'interazione tra Controller e View: Si può notare come in questa interazione il Controller sia una parte principalmente "passiva". L'unica interazione avviene tramite DataUpdater che richiede alla View di aggiornarsi.

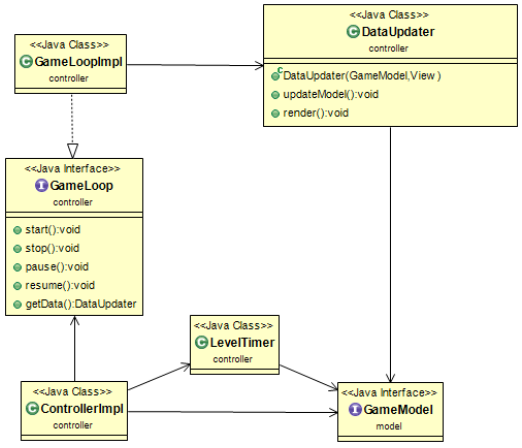

Figura 2.9: Rappresentazione UML che mostra l'interazione tra Controller e Model: ControllerImpl ha i il compito di impostare la mappa di gioco, di inizializzare una nuova partita e di impostare la direzione di Pac-Man; DataUpdater deve invece aggiornare periodicamente il Model, mentre LevelTimer svolge il ruolo di cronometro usato dal Model sia per i livelli sia per la modalità invertita.

TIMER DEL LIVELLO:
Nella nostra rivisitazione del gioco Pac-Man, come spiegato inizialmente, dopo un prefissato lasso di tempo, si accede al livello successivo. La durata di ogni livello ed il passaggio da un livello al successivo sono implementati nel Model, mentre nel Controller è gestito il cronometro. La classe LevelTimer infatti implementa il concetto di timer, il quale viene eseguito in un thread apposito. Esso internamente sfrutta le classi java.util.Timer e java.util.TimerTask.

CARICAMENTO MAPPA DI GIOCO:
La lettura della mappa da file avviene tramite la classe GameMapLoaderImpl, appositamente creata da Vissani, ed in seguito viene impostata nel Model. Questo avviene per la prima volta appena il Controller viene creato nella classe principale dell'applicazione (PacManApp), ed in seguito tutte le volte che l'utente decide di cambiare la mappa di gioco nella schermata "Settings".

GESTIONE CLASSIFICA E SALVATAGGIO DATI SU FILE:
Al termine di ogni partita viene salvato il nome del giocatore, il punteggio ed il livello raggiunto. L'interfaccia RankingManager si occupa della gestione della classifica, permettendo di sapere l'attuale miglior punteggio, tutti i giocatori salvati con i rispettivi risultati, di salvare un nuovo giocatore e di resettare la classifica. La classe RankingManagerFileImpl rappresenta l'implementazione concreta di questa interfaccia e si occupa quindi anche della lettura e scrittura su file dei dati da mantenere. Tale classe crea nella home dell'utente la cartella .PacMan contenente il file .PacManScore.json, nel quale vengono salvati i dati in formato JSON.

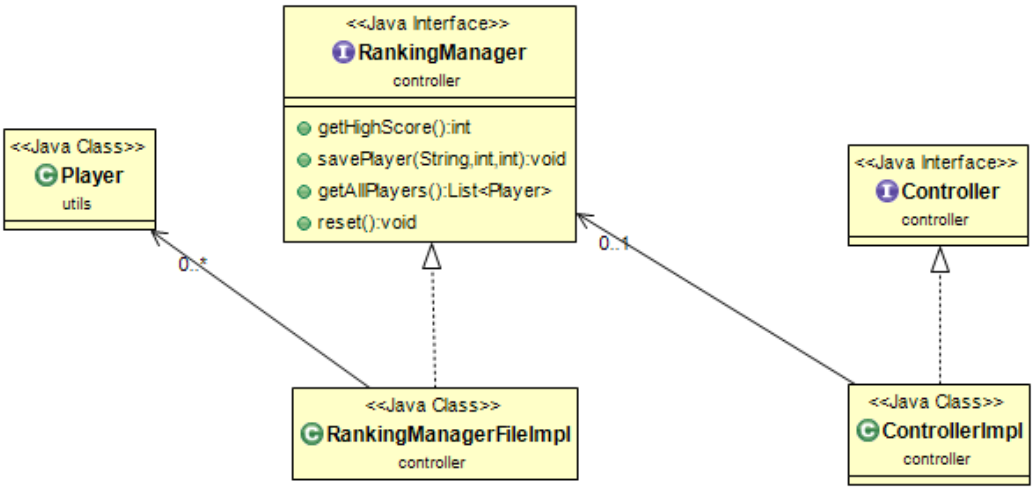

Figura 2.10: Rappresentazione UML della gestione della classifica

VIEW

La sezione View gestisce la parte grafica del gioco e l'interazione con l'utente. Gli elementi principali sono rappresentati:

- Dall'interfaccia View, che permette la visualizzazione ed il passaggio tra diverse scene e, durante la partita, il rendering del gioco;
- Dalla classe astratta SceneController che ha il compito di gestire le varie scene.

Ogni classe che estende SceneController gestisce una certa scena dell'applicazione e per identifcare le diverse scene è stata usata un'enumerazione (GameScene).

SOUNDMANAGER:

La classe SoundManager si occupa della riproduzione degli effetti sonori ed è stata realizzata seguendo il pattern creazionale Singleton con "lazy initialization" e "thread-safe". Tale scelta è risultata necessaria per evitare di dover portare i riferimenti all'oggetto nei campi di tutte le classi che usano SoundManager; infatti concettualmente tale classe deve avere un'unica istanza ma, nella pratica, il suo uso è richiesto in molteplici punti della View. Per identificare i diversi suoni che possono essere riprodotti è stata usata un'enumerazione (Sound).

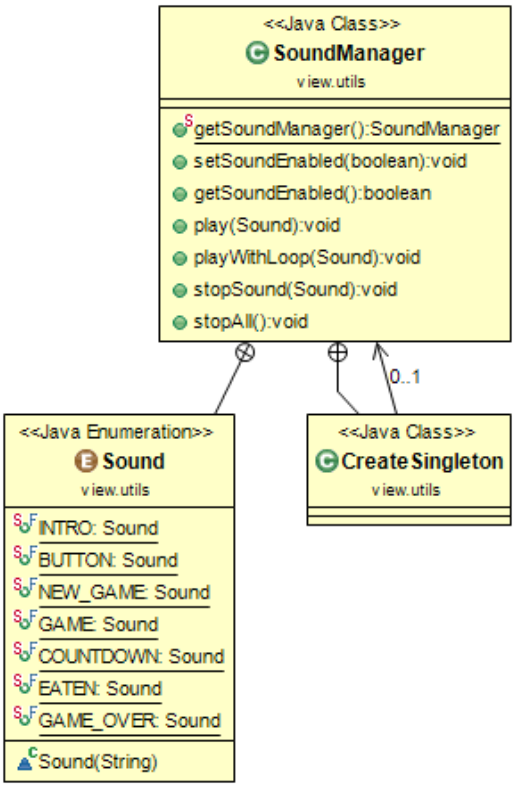

Figura 2.11: Rappresentazione del SoundManager

Eddie Barzi

Pacman

Per la realizzazione di Pacman sono partito dallo scrivere un'interfaccia generica *Entity* che rappresentasse le diverse entità del gioco (in questo caso i fantasmi e Pacman), la quale definisce i metodi che ogni entità deve avere. È stata poi creata una classe astratta dove sono implementati i metodi comuni a tutte le entità. In questo modo è stata resa possibile l'implementazione di altre entità in futuro.
Per l'inizializzazione dell'oggetto Pacman sono necessari molti parametri, per questo ho da subito deciso di utilizzare il **pattern builder** che permette un chiaro settaggio dei parametri a differenza di un costruttore. Ho realizzato una classe Builder innestata dentro la classe PacManImpl utilizzando una interfaccia "fluent". In questo modo ho potuto rendere il costruttore di PacManImpl privato e quindi rendere obbligatorio l'uso del builder. La classe Builder ha i metodi per settare le varie proprietà dell'oggetto da costruire, e quando tutto è pronto si chiama il metodo build che restituisce l'oggetto

PacManImpl. I campi nel Builder sono inizializzati come opzionali vuoti, in modo da fare un check nel metodo build finale e quindi lanciare eccezione in caso di parametri non settati. È presente inoltre un controllo sul settaggio delle vite che non permette l'inserimento un valore minore o uguale a 0.

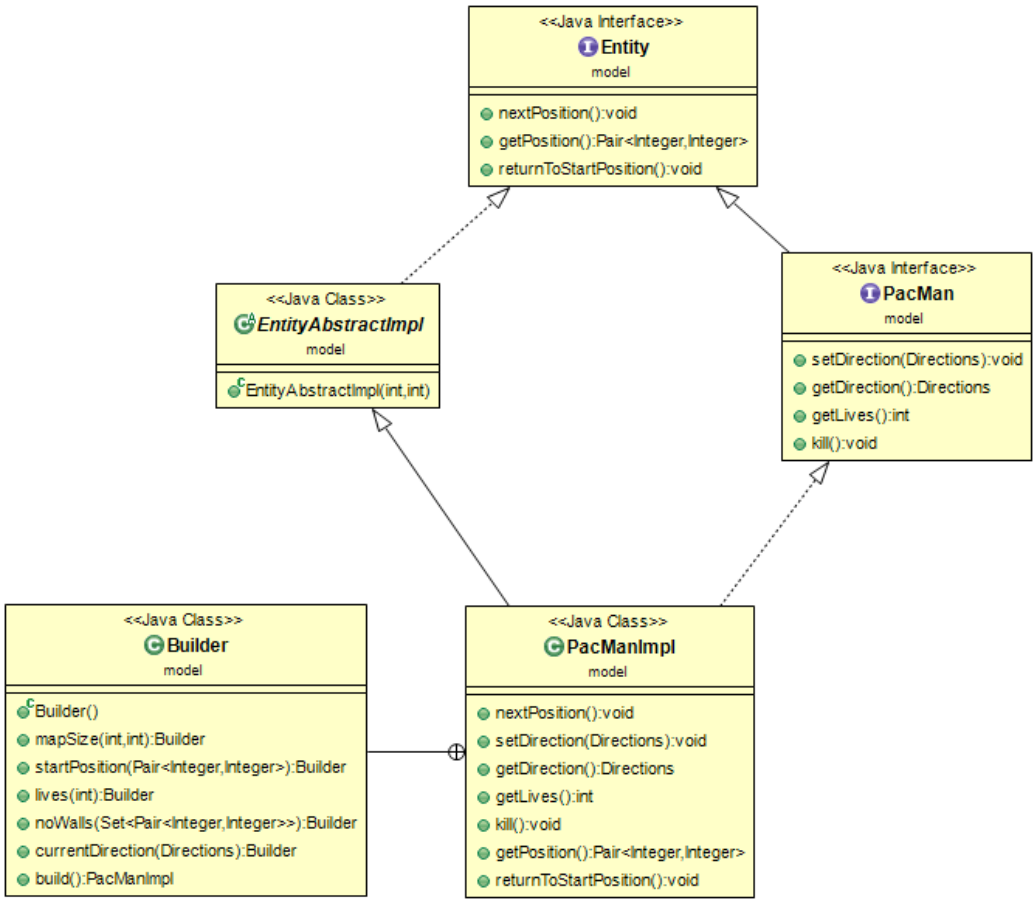

Figura 2.12: Rappresentazione UML di Pacman e del suo Builder

Gestione degli input

Per acquisire gli input da tastiera è stato creato il metodo onKeyPressed nella classe astratta SceneController. Ogni classe che estende SceneController potrà sovrascrivere tale metodo per implementare il codice da eseguire alla pressione di un tasto della tastiera. Il metodo non è stato dichiarato astratto in modo da lasciare libertà alla classe finale di sovrascriverlo, e quindi assegnare l'azione da eseguire alla pressione di un tasto, oppure no.

Tutte le classi riguardanti le scene della View dovranno estendere la classe

astratta SceneController in modo da avere un riferimento sia alla classe principale della View, per utilizzare il metodo di cambio scena, sia al controller, per richiedere e inviare dati.

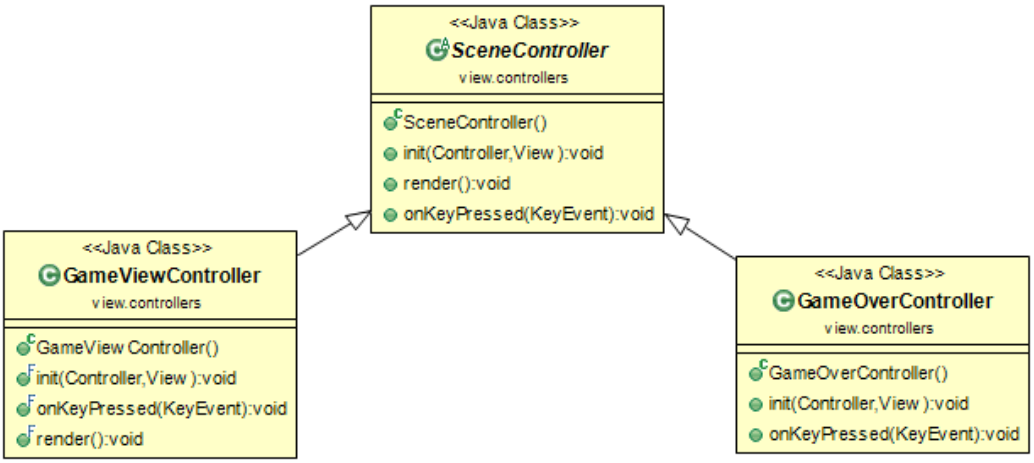

Figura 2.13: Rappresentazione UML della classe astratta SceneController e di due delle sue sottoclassi, ovvero GameViewController e GameOverController.

Per far partire il gioco, per metterlo in pausa e riprenderlo ho deciso di utilizzare solamente il tasto spacebar. Ho quindi creato una classe GameState per tenere in memoria lo stato attuale del gioco in modo da richiamare il metodo corretto alla pressione di spacebar. I possibili stati del gioco sono dichiarati nell'enum innestata dentro GameState.

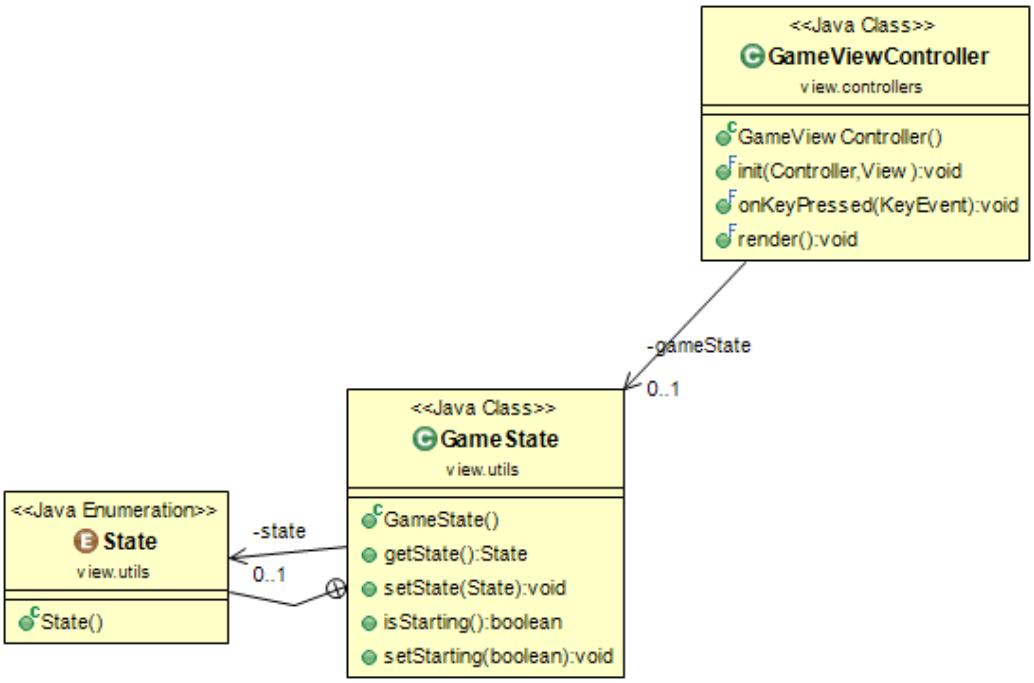

Figura 2.14: Rappresentazione UML della classe relativa allo stato del game.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Durante la creazione del progetto sono stati realizzati dei test con l'ausilio di Junit5 per controllare la correttezza di alcune classi.

- TestGameMapBuilding: viene testato il builder della mappa di gioco, il quale lancia un'eccezione nel caso in cui non vengano impostati uno o più parametri.
- TestGameMapLoader: viene testata la classe che si occupa del caricamento da file della mappa di gioco; nel caso in cui il percorso del file della mappa sia sbagliato viene lanciata un'eccezione.
- TestPacManEntity: viene testata la corretta creazione ed il corretto comportamento dell'entità pacman. L'oggetto pacman viene inizializzato tramite il builder il quale, in caso di parametri errati o mancanti, lancia un'eccezione. Viene inoltre controllato lo spostamento dell'oggetto pacman, il quale deve rispettare le posizioni consentite (non può andare sopra ai muri e deve spostarsi correttamente nel caso toroidale).
- TestFileManager: tale test ha lo scopo di controllare il corretto funzionamento della classe RankingManagerFileImpl che implementa la gestione della classifica ed il salvataggio su file. Il particolare viene testato:
  - Il corretto inserimento in classifica di un nuovo giocatore secondo un ordinamento decrescente dei punteggi, mantenendo un numero massimo prefissato di giocatori (vengono eliminati quelli con punteggio più basso);

– La corretta restituzione del punteggio più alto;
– Il corretto reset della classifica.

3.2 Metodologia di lavoro

Al termine dello sviluppo dell'applicazione, possiamo affermare di aver rispettato la suddivisione dei compiti dichiarata inizialmente, a cui è stato aggiunto lo svolgimento delle funzionalità opzionali. Sono di seguito elencate più in dettaglio le implementazioni svolte da ciascun membro:

- Filippo Vissani: creazione di più labirinti e dei loro elementi, gestione delle collisioni, dell'effetto toroidale, delle due modalità di gioco (normale ed invertita, cioè quando Pac-Man può mangiare i fantasmi), dei livelli e del punteggio. Inoltre, per quanto riguarda la View, la realizzazione della schermata 'Settings' e la rappresentazione grafica della mappa.

- Eddie Barzi: creazione e gestione di Pac-Man (Model e View), gestione degli input, gestione perdita di vite e realizzazione della schermata di 'Game-over';

- Davide Rossi: creazione e gestione dei fantasmi (Model e View) e realizzazione del menu iniziale;

- Sofia Belloni: Controller, possibilità di mettere il gioco in pausa e gestione della classifica utenti. Per quanto riguarda la View, la realizzazione dello HUD di gioco, del Sound Manager e delle schermate 'Ranking' e 'Instructions'.

Il GameViewController è stato realizzato in collaborazione da tutti i membri del gruppo, ognuno dei quali ha implementato la parte di propria competenza.

Impiego del DVCS

Lo sviluppo dell'intero progetto è stato accompagnato dall'utilizzo del DVCS Git su piattaforma BitBucket. L'approccio iniziale era quello di creare, oltre al branch develop, ulteriori tre branch (model, controller e view), tuttavia nonostante le varie parti di lavoro fossero ben distinte erano presenti dei collegamenti tra esse e quindi serviva un branch in comune sul quale testare le funzionalità sviluppate. Per tali motivi abbiamo quindi lavorato quasi esclusivamente su *develop*, creando dei feature-branch quando necessario.

3.3 Note di sviluppo

Davide Rossi
Per la realizzazione della parte grafica dei fantasmi ho utilizzato la libreria JavaFX, in particolare la funzione di caricamento di immagini e la gestione delle loro animazioni da un punto A a un punto B. Per quanto riguarda la creazione del menu iniziale, ho utilizzato il programma SceneBuilder in modo da dividere l'implementazione grafica in FXML da quella in JavaFX relativa al funzionamento dei bottoni. Relativamente alla logica dei fantasmi, ho implementato una versione dell'algoritmo di Dijkstra[1] iterativo che facesse al mio caso, in modo da trovare sempre il percorso migliore in maniera efficiente, mentre per la loro parte creazionale ho utilizzato dei campi optional tramite il pattern Builder della ghost factory.

Sofia Belloni

- Lambda expression per rendere il codice più compatto e leggibile;
- Libreria JavaFX per la realizzazione della parte grafica. In particolare ho diviso la parte di design (racchiusa in un file FXML) da quella del codice che la rigurda.
- Libreria Gson per poter convertire oggetti Java nella loro rappresentazione JSON e viceversa.

Per l'utilizzo delle librerie sopra citate, mi sono affidata principalmente alla documentazione ufficiale. Per quanto riguarda il GameLoop invece ho consultato varie guide online che ne elencano i diversi tipi, trovando particolarmente utile la seguente:
`https://gafferongames.com/post/fix_your_timestep/`
Infine i suoni riprodotti durante il gioco sono stati scaricati dai seguenti siti, i quali consentono liberamente l'uso delle proprie risorse fintanto che il prodotto finale non sia a scopo di lucro.
`https://www.classicgaming.cc/classics/pac-man/sounds`
`https://opengameart.org/`

Filippo Vissani
Nella gestione della mappa di gioco ho utilizzato spesso le lambda expressions e gli stream, dato che tutte le posizioni vengono salvate dentro oggetti di tipo Map e Set, questa scelta ha permesso di mantenere il codice compatto e facilmente comprensibile.
Per definire una posizione all'interno della mappa ho usato la classe Pair

presentata durante il corso, che fa uso dei generici X e Y (in questo caso X rappresenta la posizione sull'asse delle ascisse e Y sull'asse delle ordinate).
I campi del builder della mappa sono definiti come Optional, quindi in fase di creazione dell'oggetto è possibile verificare se tutti i campi sono stati impostati correttamente senza utilizzare *null*.
Ho utilizzato la libreria JavaFX per creare graficamente la mappa di gioco in modo che mantenesse determinate proporzioni, così facendo viene sfruttata al massimo la dimensione della finestra dell'applicazione.

Eddie Barzi
Sono stati utilizzati campi optional nel builder di Pacman.
Nella view per fare l'animazione di Pacman "che mangia" ho usato l'oggetto AnimationTimer che in questo caso cambia l'immagine di pacman ogni decimo di secondo. Per fornirgli le immagini ho pensato di implementare un iteratore ciclico, in modo da iterare una lista composta dalle immagini usate per formare l'animazione richiamando il metodo next. Essendo l'iteratore *ciclico* deve restituire le immagini dalla prima all'ultima e poi ricominciare una volta arrivato alla fine. Dopo qualche ricerca ho visto che nella libreria com.google.common.collect.Iterables era già presente quindi ho utilizzato quello.
È stata utilizzata la libreria JavaFX per la realizzazione della schermata di gameover e per la rappresentazione di pacman nella schermata di gioco. È stata suddivisa la parte di design racchiudendola in un file FXML. Nella classe GameOverController è stato utilizzato un filtro sull'opzionale del nome utente da salvare. Nel caso in cui l'utente lasci il campo vuoto viene salvato come Guest.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Davide Rossi
E' stato un progetto molto ambizioso e molto impegnativo, ho imparato tantissime cose nell'ambito della progettazione Java in gruppo, dei pattern, della creazione di GUI e delle dipendenze tra classi. La mia parte era quasi del tutto indipendente e quindi sono riuscito a lavorare da solo senza particolari complicazioni, concentrandomi sul funzionamento degli algoritmi. Non è stato facile trovare tutti i bug presenti nei fantasmi, data la loro complessa struttura algoritmica, ma con l'aiuto dei miei compagni che mi segnalavano eventuali comportamenti anomali durante il gioco sono riuscito a identificarli e correggerli. Ho trovato qualche difficoltà nel capire come integrare una classe JavaFX con il linguaggio FXML, ma dopo essermi informato sono riuscito nel mio intento, anche grazie al software SceneBuilder. Per quanto riguarda Git, ho impiegato un po' di tempo a capirne i meccanismi e le potenzialità, e probabilmente avrei potuto sfruttarlo meglio. Sicuramente questo progetto è stato un ottimo modo per imparare concretamente la progettazione Java, che ora è senza dubbio il linguaggio che ho approfondito meglio in questo corso di laurea.

Filippo Vissani
All'inizio dello sviluppo del progetto non avevo idea di come si sarebbe conclusa l'impresa, perché avevamo diversi dubbi su come progettare ed implementare alcune parti.
Complessivamente sono molto contento del risultato finale, raggiunto grazie all'impegno costante di tutti i membri del gruppo. Le conoscenze acquisite durante il corso sono state indispensabili durante lo sviluppo del gioco, prima

delle lezioni non avevo mai sentito parlare di pattern e lambda expression.
Forse avrei dovuto progettare la mia parte sfruttando più pattern, credo di non essere riuscito ad individuare tutti i casi in cui potevano essere utilizzati.
Le lacune più grandi si sono presentate nello sviluppo dell'interfaccia grafica, non avevo mai usato JavaFX e FXML prima, quindi ho dovuto guardare diverse guide su come utilizzarli, oltre a studiarne la documentazione.

Sofia Belloni
Mi ritengo soddisfatta del lavoro compito, sviluppare questo progetto mi ha permesso di apprezzare maggiormente quanto studiato durante questo corso e di capire l'importanza delle fasi iniziali della progettazione. Non mi ero mai cimentata prima d'ora nella realizzazione di un videogioco: la parte più gratificante è stata sicuramente vedere sullo schermo i risultati concreti del nostro lavoro e dei nostri sforzi. Per quanto riguarda la parte di mia competenza, ho cercato dove possibile di applicare al meglio quanto imparato a lezione ma sono consapevole che non tutto è stato svolto nel migliore dei modi, sia per una questione di inesperienza sia di tempistiche; nonostante ciò il mio giudizio è complessivamente positivo. Sicuramente la realizzazione di questo progetto è stata tra le esperienza più interessanti e formative affrontate finora.

Eddie Barzi
All'inizio di questo progetto sapevo che sarebbe stato molto impegnativo per tutti, perchè nessuno di noi aveva mai lavorato nè ad un progetto così grande nè in gruppo prima d'ora. Alla fine posso dire di essermi trovato molto bene con questo gruppo in quanto abbiamo dato tutti il 100% per realizzare al meglio il gioco e c'è stata una grande soddisfazione nel vedere il progetto completato e funzionante. All'inizio si era pensato di utilizzare la libreria FXML, un vero e proprio framework per la realizzazione di giochi in 2D. Poi però è stato considerato controproducente l'uso di quella libreria, dato che lo scopo principale dell'esame è mostrare la nostra metodologia di progettazione.
Ho cercato di applicare il più possibile tecniche di sviluppo avanzate imparate a lezione, ma probabilmente c'è qualche punto dove avrei potuto fare meglio.
Ho ritenuto molto interessante l'uso del DVCS Git che non conoscevo prima di questo corso.
Ritengo che questo sia stato uno dei corsi più completi e interessanti tra quelli frequentati fin'ora. Un esame di questo tipo, con prima una prova in laboratorio e poi un progetto a piacere da svolgere in gruppo, ci ha spinto nel cimentarci a 360 gradi su tutti gli argomenti trattati nel corso.

4.2 Difficoltà incontrate e commenti per i docenti

In fase di progettazione si sono presentati problemi relativamente all'utilizzo della notazione UML, in particolare per quanto riguarda le relazioni tra classi. Inoltre in fase di sviluppo non è sempre stato facile rispettare il pattern architetturale MVC, probabilmente a causa della mancanza di esperienze precedenti a riguardo. Le difficoltà maggiori sono state riscontrate nell'utilizzo della libreria JavaFX, per questo suggeriamo di approfondire maggiormente l'argomento durante le lezioni.

Appendice A

Guida utente

All'avvio del gioco viene aperta la schermata del menu principale, in cui l'utente può scegliere tra le seguenti opzioni:

- New Game: permette di iniziare una nuova partita;
- Ranking: permette di visualizzare la classifica dei giocatori con i punteggi ed i livelli raggiunti.
- Settings: permette di accedere alla schermata delle impostazioni nella quale è possibile resettare la classifica, cambiare la mappa di gioco scegliendo tra quelle disponibili attraverso un menù a tendina, ed attivare/disattivare il suono.
- Instructions: permette di visualizzare i comandi e lo scopo del gioco.

Selezionando 'New Game' viene visualizzata la schermata principale di gioco; per iniziare la partita è necessario premere la barra spaziatrice.
Mangiando un quarto delle pillole delle mappa si passa alla modalità invertita: Pac-Man può mangiare i fantasmi, i quali diventano blu ed iniziano a scappare. Quando i fantasmi diventano grigi significa che si sta per tornare alla modalità normale. Ad ogni livello vengono generati quattro nuovi fantasmi.

Comandi di gioco

L'utente può muovere Pac-Man nelle quattro direzione sia attraverso i tasti W/A/S/D sia tramite le quattro frecce.
ESC: torna al menù principale
SPACE BAR: pause/resume

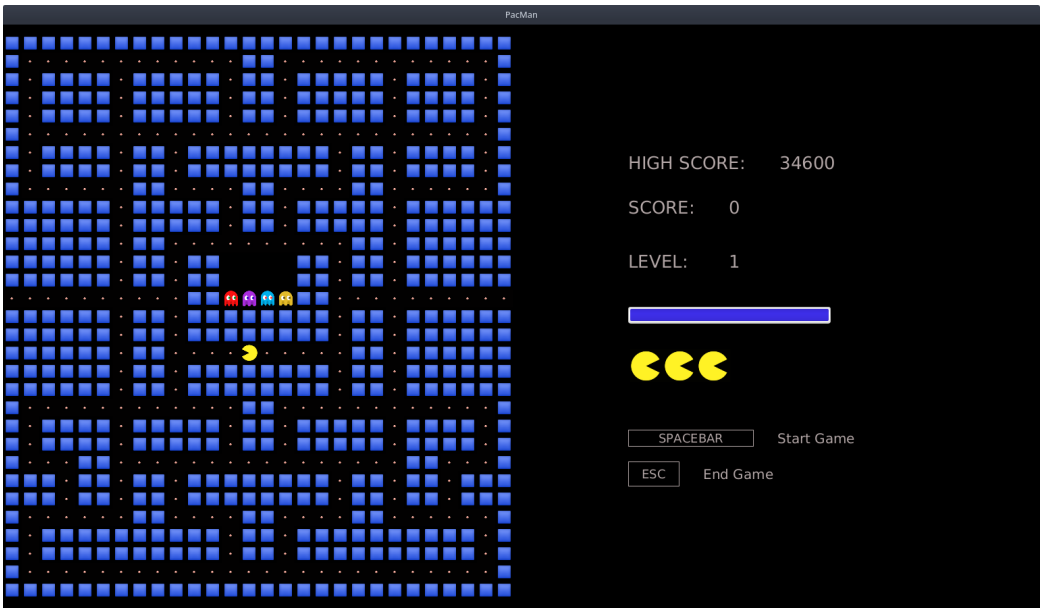

Figura A.1: Schermata di gioco.

Appendice B

Bibliografia

[1] E.W. Dijkstra, Dijkstra Algorithm, 1959