Relazione per progetto **Snake**
Corso di "Programmazione ad Oggetti"

Filippo Pilutti

25 febbraio 2022

Indice

Capitolo 1

Analisi

Il software da realizzare rappresenta un gioco ispirato alla classica famiglia di giochi conosciuta come **Snake**[1]. Una sua prima versione risale alla metá degli anni '70, ma divenne veramente popolare dopo che Nokia decise di precaricarne una propria versione all'interno dei suoi cellulari, rendendolo uno dei giochi piú conosciuti di sempre.

Snake é un gioco in cui il giocatore controlla un *serpente* all'interno di un'area di gioco delimitata da muri, con l'obiettivo di mangiare piú *mele* possibili senza mai andare a sbattere contro i muri o contro sé stesso. Per ogni *mela* mangiata il *serpente* si allungherá sempre di piú, rendendo il gioco via via piú difficile. Nel momento in cui la testa del *serpente* colpisce uno dei muri o un altra parte del suo corpo, il gioco termina in un game over.

1.1 Requisiti

Requisiti funzionali

- All'avvio, l'applicazione presenterá una schermata iniziale che renderá possibile l'avvio del gioco; sará inoltre possibile visionare l'*high score*[2].
- Il giocatore potrá controllare il *serpente*, facendolo muovere lungo le quattro direzioni cardinali (su, giú, destra, sinistra), tramite l'input da tastiera delle quattro freccette direzionali.
- L'area di gioco presenterá giá all'avvio della partita una *mela* posizionata casualmente; una volta mangiata una *mela* ne verrá immediatamente generata un'altra in una posizione casuale della mappa.

[1] `https://en.wikipedia.org/wiki/Snake_(video_game_genre)`
[2] Punteggio massimo ottenuto

- L'interfaccia di gioco presenterá un indicatore per il punteggio attuale della partita; i punti aumenteranno per ogni *mela* mangiata.
- Quando la testa del serpente colpisce un muro o un'altra parte del suo corpo, il gioco dovrá terminare con una schermata di game over che fornirá la possibilitá di iniziare un'altra partita o di terminare l'esecuzione del programma. In caso di game over, il punteggio dovrá venire salvato.
- Lo sviluppo prevede anche l'eventuale presenza di elementi opzionali, che potrebbero venire inseriti durante il corso della progettazione. Questi elementi potrebbero essere delle *mele* diverse da quelle normali, che garantirebbero dei *bonus* o *malus* temporanei, oppure degli ostacoli all'interno dell'area di gioco.

Requisiti non funzionali

- Garantire una minima efficienza tale da poter giocare una partita.

1.2 Analisi e modello del dominio

Le entitá di gioco principali presenti sono due: il *serpente* e le *mele*, che interagiscono all'interno di un'area di gioco. La testa del serpente si puó muovere lungo le 4 direzioni cardinali e il corpo segue passo passo ogni posizione occupata precedentemente dalla testa. Il serpente mangia una mela nel momento in cui la sua testa viene a trovarsi nella stessa posizione in cui ne é presente una. Il risultato dell'aver mangiato una mela é l'incremento del punteggio di un determinato fattore e l'incremento della lunghezza del serpente. La mela é un'entitá passiva; ogni volta che ne viene mangiata una bisogna generarne un'altra in una posizione casuale della mappa in cui non vi sia giá presente una parte del corpo del serpente. Potrebbero esserci delle mele particolari il cui comportamento é diverso da quelle standard, che fornirebbero cioé dei bonus o malus temporanei al gameplay (come ad esempio l'aumento della velocitá di movimento o l'accorciamento del serpente).

Le entitá principali rilevate durante l'analisi sono sintetizzate in fig. 1.1.

La difficoltá primaria sará quella di gestire le collisioni tra le varie entitá in gioco, quindi tra il serpente e le mele, i muri che delimitano l'area di gioco e altre parti del corpo del serpente stesso.

La modalitá multiplayer, inserita come funzionalitá opzionale, richiede una quantitá di lavoro che non rientrerebbe nel monte ore di lavoro previsto e per questo motivo essa non sará presente in questa versione del software.

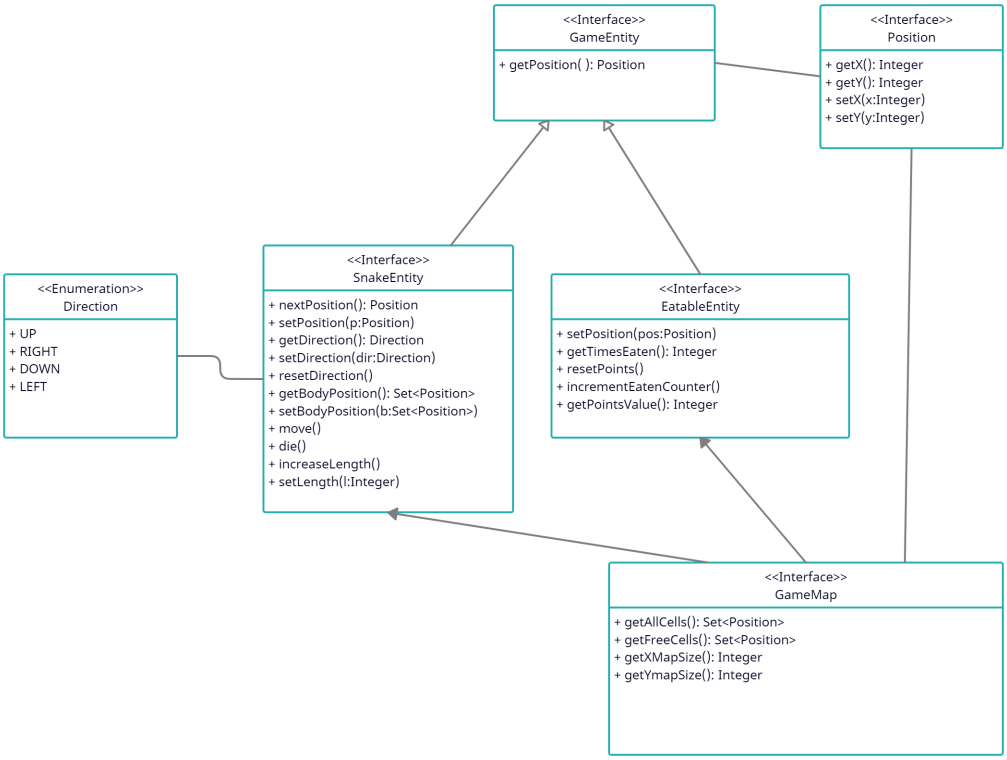

Figura 1.1: Schema UML dell'analisi del problema rappresentante le principali entitá rilevate

Capitolo 2

Design

2.1 Architettura

Per realizzare questo progetto é stato scelto di utilizzare il pattern architetturale MVC (Model-View-Controller), in quanto garantisce una chiara suddivisione del codice dal punto di visto logico e permette di separare nettamente la componente grafica dal dominio dell'applicazione; ció garantisce la possibilitá di cambiare completamente l'aspetto grafico senza modifiche ad altre porzioni di codice al di fuori di quelle della view e anche la possibilitá di riutilizzare la componente del model per altri applicativi che affrontano lo stesso dominio.

Le entitá di gioco individuate nell'analisi del dominio sono definite all'interno del Model. Qui vengono descritte le loro caratteristiche principali e il loro comportamento. Il Model ha il compito di mantenere la logica del gioco e di definire il comportamente delle sue entitá in base a ció che succede e, seguendo il pattern MVC, non deve interagire direttamente con le altre due componenti (View e Controller), per cui al suo interno non vi sono riferimenti ad esse. Quindi il Model fornirá attraverso i suoi metodi la possibilitá al Controller di accedere ai suoi dati e al suo stato attuale di gioco e anche la possibilitá di modificarli. Le due principali entitá presenti sono modellate dalle interfacce EatableEntity e SnakeEntity, che a loro volta estendono la generica interfaccia GameEntity che rappresenta una qualsiasi entitá caratterizzata da una posizione nell'ambiente di gioco.

Il compito della View é quello di visualizzare su schermo tutte le entitá e di fornire all'utente un modo per interagire con l'applicazione. Qua abbiamo delle interfacce che permettono di rappresentare le entitá del gioco a livello grafico, come SnakeView e AppleView che estendono l'interfaccia DrawableGameEntity che rappresenta una qualsiasi entitá disegnabile su schermo, e

altre che permettono la creazione delle schermate di gioco e gestiscono le entitá al loro interno, come MapView e View. L'interfaccia View fornisce i metodi che permettono al Controller di disegnarla e modificarla in base a ció che dice la logica del gioco.

Infine il Controller rappresenta il filo conduttore del programma, il componente che garantisce la corretta interazione tra Model e View. Esso ha il compito di utilizzare le informazioni ottenute dal Model per visualizzare correttamente la View e di utilizzare gli "eventi" che avvengono nella view per modificare il Model. In questo caso il cuore del Controller é individuato proprio nell'interfaccia Controller, che fornisce il metodo per l'avvio del programma e, estendendo le interfacce GameObserver e InputController, fornisce anche lo sviluppo dell'interazione dell'utente attraverso la tastiera e i comandi a schermo. In questa componente avviene anche la gestione delle collisioni tra le entitá in gioco, definita dall'interfaccia CollisionManager.

Utilizzando MVC come pattern architetturale risulta semplice sostituire la componente View visto che in essa non vi sono rimandi diretti alle altre due componenti in quanto tutta l'interazione tra M, V e C é gestita unicamente dal Controller.

In fig. 2.1 é mostrata una semplificazione del diagramma UML architetturale.

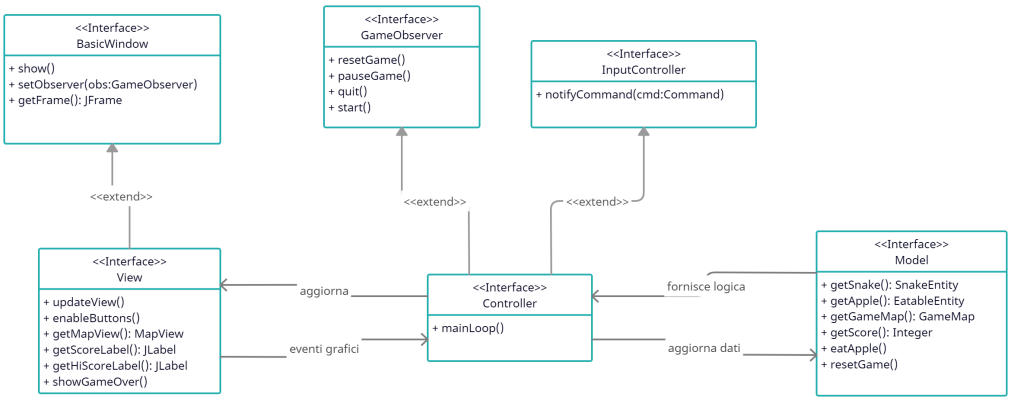

Figura 2.1: Schema UML architetturale di Snake. É mostrata l'interazione delle tre componenti Model, View e Controller rappresentate dalle tre interfacce omonime.

2.2 Design dettagliato

Model

Il Model é cosí strutturato: abbiamo una classe principale di modello che funge da suo elemento centrale nella quale sono istanziate le entitá di gioco; questa classe fornisce al Controller i metodi necessari per avere accesso a queste entitá e ai loro dati e per modificare lo stato del modello. All'interno di questa classe sono quindi istanziati come campi le tre entitá fondamentali, che sono le classi Snake, Apple e GameMapImpl. A livello di struttura delle interfacce é stato deciso di creare un'interfaccia generale GameEntity che potrebbe rappresentare una qualsiasi entitá del gioco, che viene poi specializzata in base alle esigenze delle differenti entitá che vanno create. Anche l'interfaccia EatableEntity rappresenta una qualsiasi entitá "mangiabile" dallo snake. Questa scelta rende possibile l'aggiunta in futuro di nuove entitá di gioco diverse da quelle giá presenti, estendendo e specializzando GameEntity oppure implementando con nuove classi EatableEntity.

Snake

La classe Snake implementa l'interfaccia SnakeEntity, che a sua volta estende l'interfaccia GameEntity. Per questa classe é stato deciso di implementare il pattern Builder per rendere piú semplice e leggibile la creazione dell'entitá snake visto che essa prevede l'inizializzazione di diversi parametri. Per implementare questo pattern si é definito un costruttore privato per Snake ed é stata inserita una classe SnakeBuilder innestata pubblica al suo interno che fornisce metodi ritornanti SnakeBuilder utilizzabili per inizializzare a cascata tutti i parametri necessari e un metodo build() che istanzia un oggetto di tipo Snake correttamente inizializzato. Questa classe fornisce attraverso i suoi metodi pubblici la possibilitá di accedere al corrente stato del serpente a livello logico (getDirection(), getBodyPosition()) e quella di modificarne lo stato (setDirection(), resetDirection(), setBodyPosition(), increaseLength(), die()) e inoltre anche quella di semplicemente aggiornare lo stato attuale (move()).

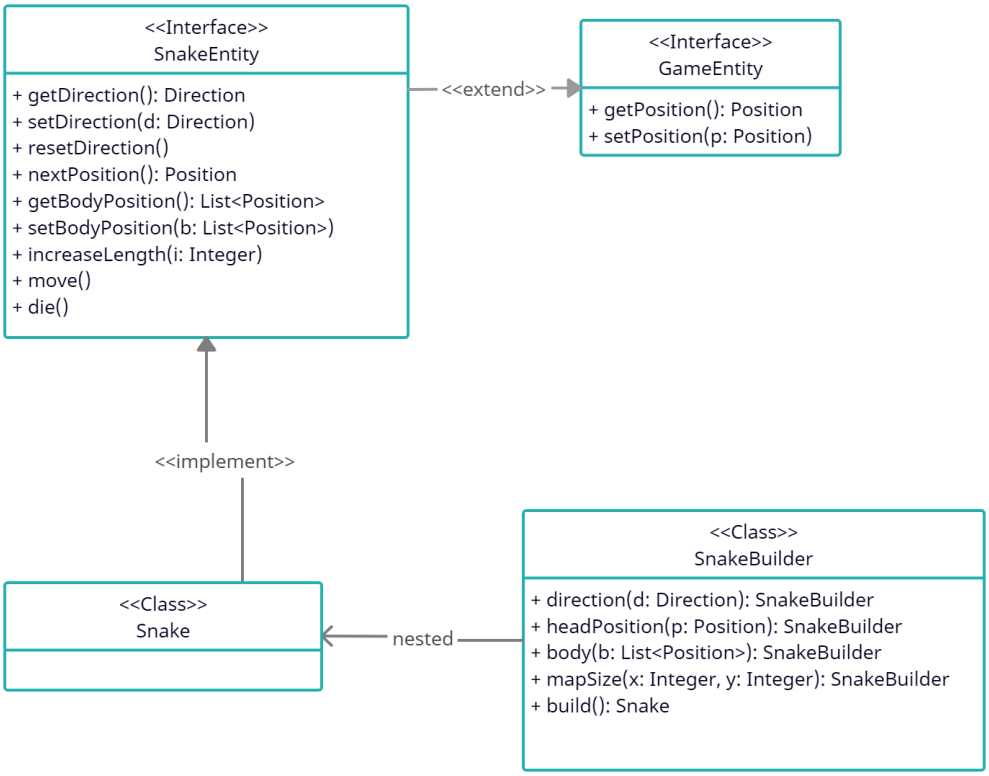

Figura 2.2: Schema UML che rappresenta l'architettura del modello dell'entitá snake, con il relativo pattern Builder

Apple

L'architettura scelta per l'entitá apple é molto semplice, infatti abbiamo un'interfaccia EatableEntity (che estende sempre GameEntity) implementata dalla classe Apple. La scelta di avere un interfaccia che modella una entitá mangiabile é stata effettuata poiché dá la possibilitá di poter aggiungere in future versioni del programma ulteriori entitá consumabili dal serpente, come ad esempio powerup o tipi diversi di mele. Anche qua i metodi pubblici permettono di accedere ai dati di questa classe (getTimesEaten(), getPointsValue()) e di modificarne lo stato (resetPoints(), incrementEatenCounter()).

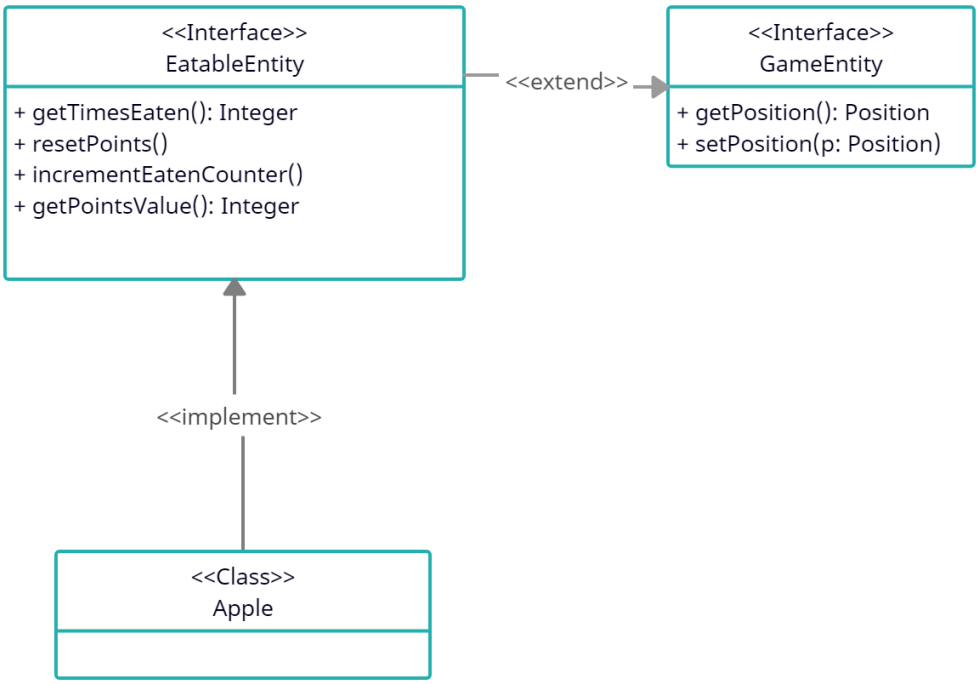

Figura 2.3: Schema UML che rappresenta l'architettura del modello dell'entitá apple

View

Per l'architettura generale della view é stato scelto di avere una classe centrale GameView in cui vengono istanziati e disegnati tutti gli elementi grafici del programma. Nella classe GameView viene quindi creato il frame che costituirá la schermata principale del gioco, nel quale vengono aggiunti nella parte superiore dei label che indicheranno il punteggio corrente e il punteggio record, nella parte centrale un panel di tipo MapView che rappresenta la mappa di gioco con le entitá e nella parte inferiore dei pulsanti utili per mettere in pausa, ricominciare la partita o terminare l'esecuzione del programma. La classe GameView implementa l'interfaccia View, che a sua volta estende BasicWindow. Questo rende possibile impostare un oggetto di tipo GameObserver che capterá gli eventi di click sui pulsanti per eseguire le relative azioni. É stato deciso di creare un'interfaccia BasicWindow in quanto rende possibile la creazione di altre finestre di gioco con le medesime caratteristiche (GameOver e GameStart). In questa classe sono presenti i metodi che permettono al controller di accedere alle rappresentazioni grafiche del-

le entitá (getMapView()), di modificare direttamente gli elementi del frame principale di gioco (enableButtons(), getScoreLabel(), getHiScoreLabel()) e di renderizzare ogni frame della view (updateView()).

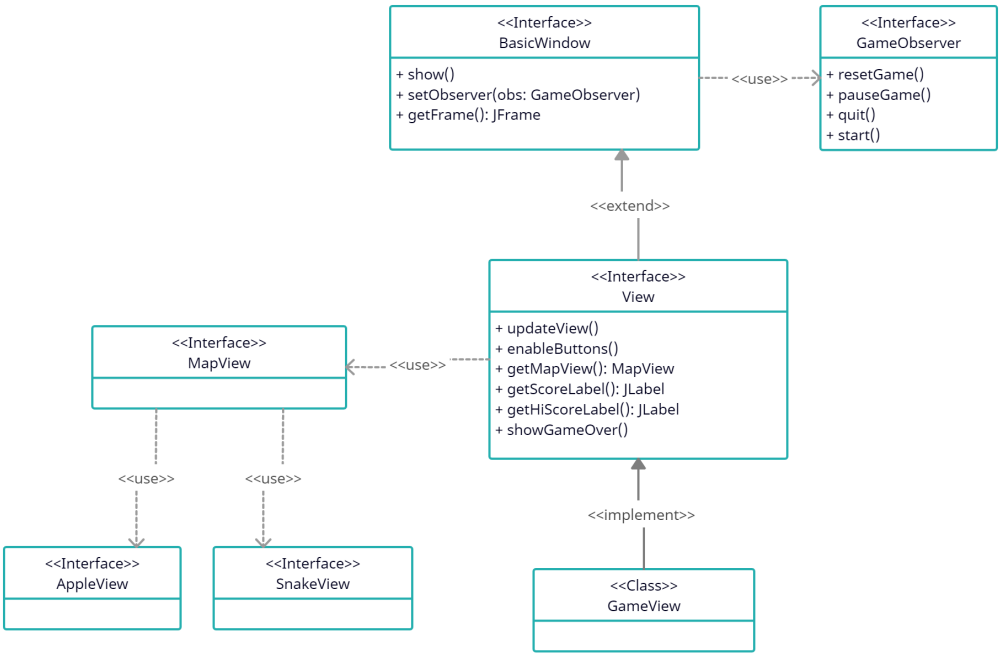

Figura 2.4: Schema UML che rappresenta l'architettura generale della view.

Entities view

Abbiamo un oggetto rappresentante la mappa di gioco (MapView) e che incorpora le due classi che corrispondono alla rappresentazione grafica delle entitá snake e apple (SnakeView e AppleView). Le interfacce SnakeView e AppleView estendono entrambe DrawableGameEntity; questa interfaccia potrá venire riutilizzata in caso di future aggiunte di entitá al gioco. MapView fornisce i metodi attraverso i quali il controller puó accedere agli oggetti SnakeView e AppleView per ottenerne lo stato e modificarlo (getAppleView(), getSnakeView()) e anche il metodo getMapBounds() che viene utlizzato nel controllo delle collisioni con i bordi della mappa dal controller. A loro volta anche AppleView e SnakeView forniscono metodi utilizzati per il controllo delle collisioni: getRect() nel primo e getHeadCenter() nel secondo.

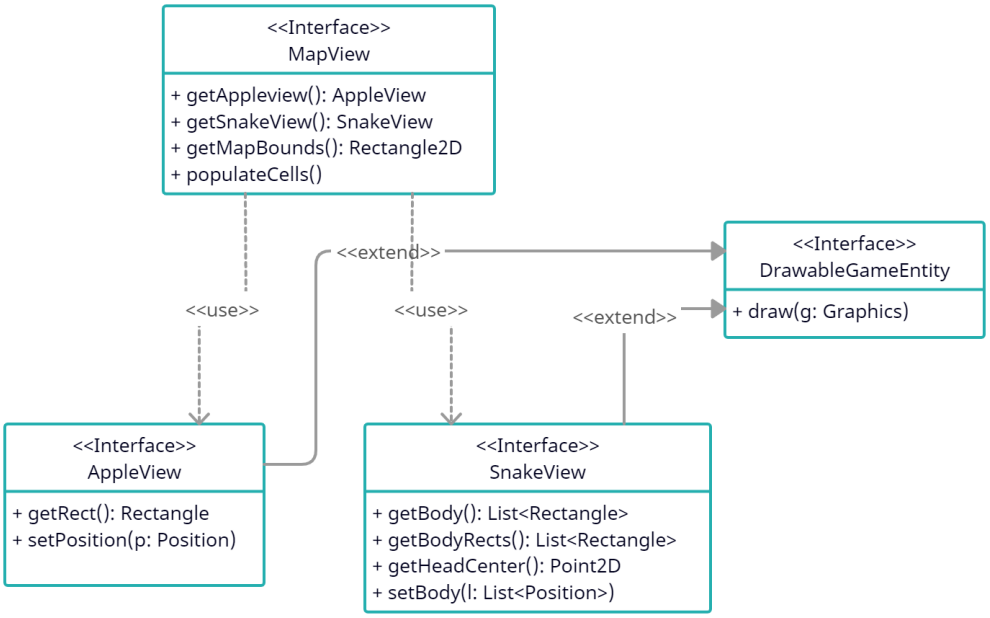

Figura 2.5: Schema UML per l'architettura delle rappresentazioni grafiche delle entitá di gioco.

Controller

Gestione input e Command pattern

Per gestire gli input da tastiera dell'utente é stato deciso di utilizzare il design pattern Command poiché tale pattern offre la possibilitá di incapsulare ogni "comando" in un oggetto, facilitandone la gestione e rendendo inoltre possibile la creazione di una coda di comandi che vengono eseguiti in ordine. Nel caso specifico di questo programma abbiamo quattro tipologie diverse di comandi (MoveUP, MoveRight, MoveDown, MoveLeft) che corrispondono alla pressione di un dei tasti per far cambiare direzione al serpente. La classe Controller implementa l'interfaccia InputController che fornisce il metodo notifyCommand, usato dalla classe KeyNotifier per notificare al Controller il giusto comando in base al tasto premuto sulla tastiera. Il Controller riceve questi comandi e li aggiunge in una coda da cui verranno poi recuperati uno alla volta ed eseguiti attraverso il metodo privato processInput.

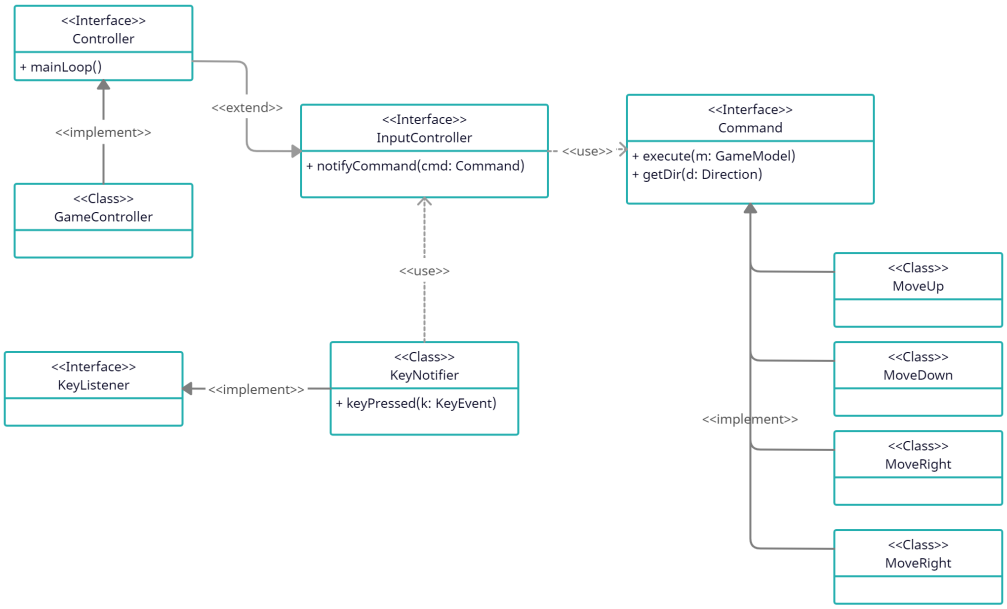

Figura 2.6: Schema UML per l'architettura della gestione degli input da tastiera con l'utilizzo del pattern Command.

Score e Collision Managers

Per la gestione del punteggio e delle collisioni abbiamo due interfacce, rispettivamente ScoreManager implementata in ScoreManagerImpl e CollisionManager implementata in CollisionManagerImpl. I metodi per gestire le collisioni in CollisionManager prendono come parametri un'istanza di GameView e un'istanza di GameModel, poiché utilizzano le informazioni ottenute dalla componente grafica per verificare se é avvenuta una collisione e, in caso positivo, accedono alle entitá del modello per aggiornarne correttamente lo stato. ScoreManager, invece, fornisce metodi usati per mantenere il punteggio della partita corrente e accedere a un file nel sistema operativo per salvare e leggere il valore del punteggio massimo ottenuto.

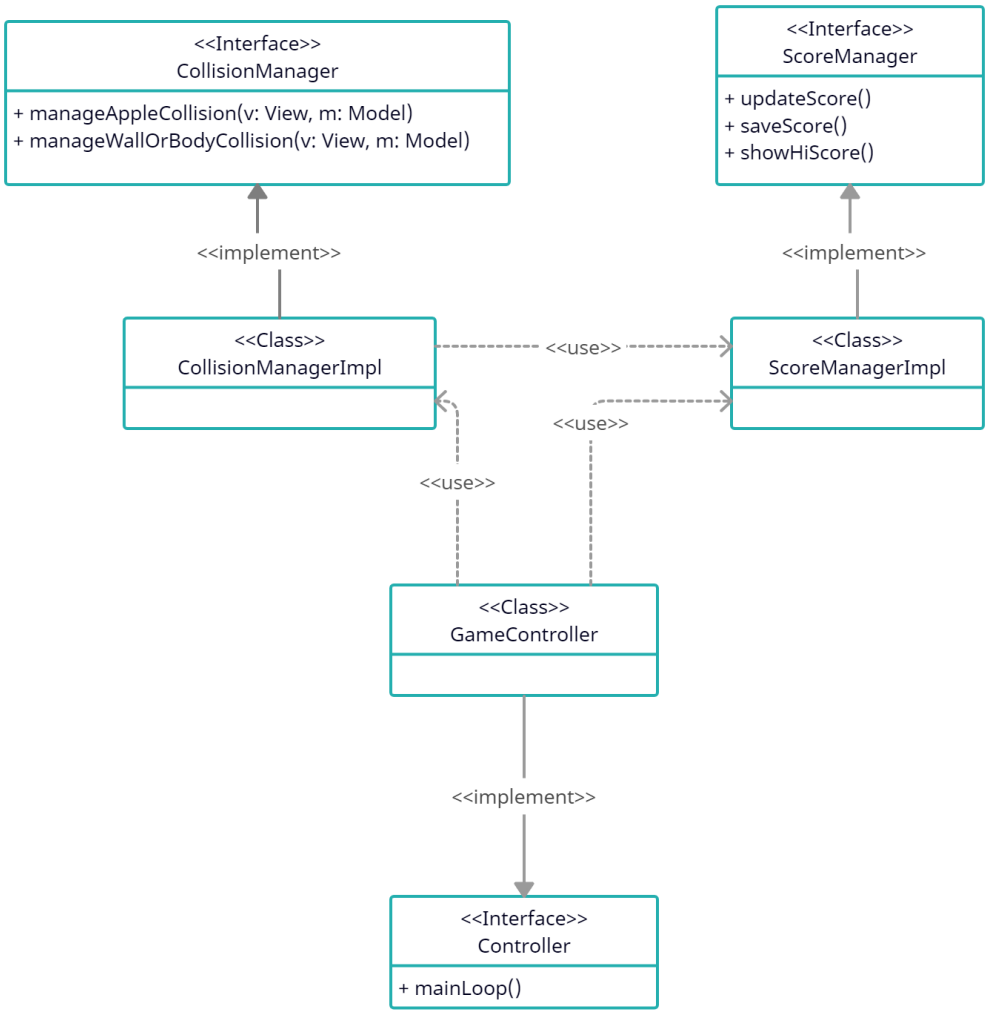

Figura 2.7: Schema UML per l'architettura dei componenti che si occupano della gestione del punteggio e delle collisioni.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per effettuare dei test automatizzati é stata utilizzata la libreria JUnit5, che fornisce metodi specifici per il testing, quali assertThrows per verificare che l'esecuzione di una porzione di codice lanci una determinata eccezione, assertEquals per verificare che due determinati oggetti corrispondano e assertTrue e assertFalse per verificare che una determinata espressione sia vera o falsa. Le classi di test sono due:

- **GameModelTest** - Qui vi sono tre test automatizzati riferiti alla classe GameModel che verificano il corretto funzionamento del suo costruttore e dei due metodi eatApple() e resetGame().
- **SnakeTest** - Qui vi sono 5 test riferiti all'entitá di gioco Snake. Questi test verificano il corretto funzionamento del builder per creare un ogetto di tipo Snake e dei metodi legati alla modifica della direzione e del movimento di Snake.

Altri test, soprattutto relativi alla corretta visualizzazione della componente view sullo schermo, sono stati eseguiti manualmente a video. Inoltre l'applicativo é stato testato anche sui sitemi operativi MacOS e Linux per verificarne la corretta esecuzione e funzionamento.

3.2 Metodologia di lavoro

Trattandosi di un progetto sviluppato singolarmente tutta la progettazione e l'implementazione é stata svolta unicamente da me per cui non vi é stato alcun lavoro di integrazione. Tuttavia ho deciso lo stesso di utilizzare il DVCS

come se si fosse trattato di un progetto cooperativo, eseguendo commit frequenti ogni volta che introducevo modifiche e aggiunte sostanziali al codice e effettuando delle push sul repository mantenuto su GitHub. Il primo commit in cui viene creato il progetto Eclipse e settate tutte le relative impostazioni é stato effettuato sul branch main, mentre tutti i seguenti commit in cui il progetto veniva man mano implementato sono stati eseguiti sul branch develop. Solo quando il codice ha raggiunto la sua versione definitiva é stata fatta una merge del branch develop nel branch main per eseguirne la release.

3.3 Note di sviluppo

- Utilizzo di **Optional** come campi della classe SnakeBuilder.
- Utilizzo di **Lambda Expressions** in particolare per aggiungere gli ActionListener ai JButton nella view e, in combinazione con l'utilizzo di stream per rilevare le collisioni tra la testa del serpente e il suo corpo in CollisionManagerImpl, per disegnare i bordi dell'area di gioco in MapViewImpl, per disegnare tutti i rettangoli che compongono Snake in SnakeViewImpl.
- Utilizzo di **Stream** nei casi giá citati nel punto precedente.

Va evidenziato che l'implementazione del metodo privato confirmDialog, definito a riga 122 della classe GameView, é stata copiata dalla slide numero 55 della lezione 15-GUI del corso di programmazione a oggetti a.a. 2020/2021, in particolare nelle righe 5-8.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Premetto che mi dispiace molto non essere riuscito a sviluppare questo progetto insieme a un gruppo, in quanto ritengo che sarebbe stata un'esperienza molto utile a livello formativo, anche se penso che l'abilitá di gestire il lavoro in gruppo non sia innata in tutti e per alcuni non é ideale affrontare un progetto di questa portata come prima esperienza collaborativa. A questo riguardo credo di essere stato un po' sfortunato, visto che per ben due volte ho visto fallire proposte di progetto in quanto risultava molto ardua la collaborazione tra i membri e addirittura alcuni non partecipavano attivamente allo sviluppo nonostante le loro promesse, ed é per questo che mi sono visto costretto a richiedere di poter affrontare questa prova autonomamente.
Detto ció, sono soddisfatto di aver portato a compimento questo progetto non tanto per la qualitá del risultato, visto che credo di aver commesso alcuni errori e molte cose le avrei potute svolgere meglio (design complessivo forse un po' troppo banale, alcune scelte implementative probabilmente non ottimali), ma piú che altro per il fatto che sia per i precedenti fallimenti che per una mia inesistente esperienza nello sviluppo da zero di un progetto di questa portata, per di piú da solo, all'inizio ero molto scoraggiato e non fiducioso sulla riuscita. Per questo posso dirmi soddisfatto in quanto il programma funziona e fa quello che mi ero prefissato che facesse all'inizio e anche l'aver portato a compimento questa cosa completamente da solo mi da grande soddisfazione.
Voglio aggiungere che probabilmente in futuro proveró ad implementare delle nuove feature aggiuntive per questo gioco, come magari delle mele bonus/malus o eventuali ostacoli sulla mappa oppure la possibilitá di aggiungere una modalitá multiplayer in stile Tron, come citato tra le funzionalitá

opzionali nella proposta di progetto, piú per piacere ed esperienza personale che per altro.

Appendice A

Guida utente

L'utilizzo del software é piuttosto semplice ed intuitivo.
All'esecuzione del programma si apre la schermata di gioco con un grande pulsante START centrale, sará sufficiente cliccarlo per far partire la partita. Durante la partita il serpente puó essere controllato sia attraverso le freccette direzionali sia attraverso i tasti WASD.
Durante la partita vi é inoltre la possibilitá di cliccare su tre pulsanti situati in basso a destra nella schermata. Questi bottoni si chiamano Pause, Reset e Quit e il loro effetto é intuitivo. Va peró sottolineata la possibilitá di attivare questi pulsanti attraverso combinazioni mnemoniche sulla tastiera, in particolare:

- **ALT+Z** corrisponde alla pressione del tasto Pause.
- **ALT+X** corrisponde alla pressione del tasto Reset.
- **ALT+C** corrisponde alla pressione del tasto Quit.

Nel caso in cui la partita terminasse a seguito di una collisione tra il serpente e il bordo della mappa o un'altra parte del suo corpo, il programma mostrerá una schermata di Game Over da cui attraverso l'utilizzo di due pulsanti sará possibile ricominciare una nuova partita o terminare l'esecuzione del programma.