# Relazione per Programmazione ad Oggetti : "Snake Runner"

Buldrini Matteo, Galli Mattia, Ince Betul, Soufi Hiba

febbraio 2026

# Indice

# Capitolo 1

# Analisi

## 1.1 Descrizione e Requisiti

L'applicazione consiste nell'iterazione del famosissimo gioco bidimensionale Snake. Il giocatore controlla un'entità serpente, muovendolo all'interno dello spazio di gioco con l'obiettivo di raccogliere oggetti o raggiungere traguardi, evitando collisioni con ostacoli o con sé stessa.
In questa versione, il corpo dello snake non si allunga dopo aver mangiato ma può consumare power-up o malus che vanno ad influenzare il corso della partita come:

- Bombe che rimuovono tutte le vite del giocatore forzando un reset;
- Funghi che ripristinano un hp del giocatore;
- Orologi che rallentano il movimento del giocatore consentendo di eseguire manovre più intricate
- Chiavi che aprono porte per accedere a sezione chiuse del livello.

Il gioco è organizzato in livelli progressivi di incrementale difficoltà.
Il giocatore dispone di un numero limitato di vite e deve completare ogni livello entro un tempo prestabilito.
Ogni volta che il giocatore perde una vita ritorna allo spawn iniziale di quel livello, quando le perde tutte questo tornerà al menù resettando la sua partita.

**Requisiti funzionali**

- Creazione dei livelli
- Creazione degli ostacoli
- Creazione di Menù e HUD

- Gestioni delle collisioni e varie hit-box del personaggio

**Requisiti non funzionali**

- Compatibilità del gioco con sistemi operativi
- Caricamenti veloci e precisi dei livelli
- Stabilità del gioco
- Comandi reattivi

## 1.2 Modello del dominio

L'applicazione è composto dal serpente che dovrà essere in grado di muoversi nelle quattrobet direzioni e superare più livelli, ciascuno dei quali definisce una specifica configurazione dello spazio di gioco e un obiettivo per portarlo al termine.

**Entità principali del dominio**

- Sessione avrà il livello corrente, il punteggio, il numero delle vite e lo stato della partità.
- Livello rappresenta la griglia logica e i suoi elementi all'interno come oggetti raccoglibili e ostacoli.
- Snake, personaggio giocabile dal giocatore.
- Ostacoli, elemento statico che avrà una collisione che porterà a perdere una vita.
- Power-Up/Malus, può essere consumato e produce un effetto sul gioco.

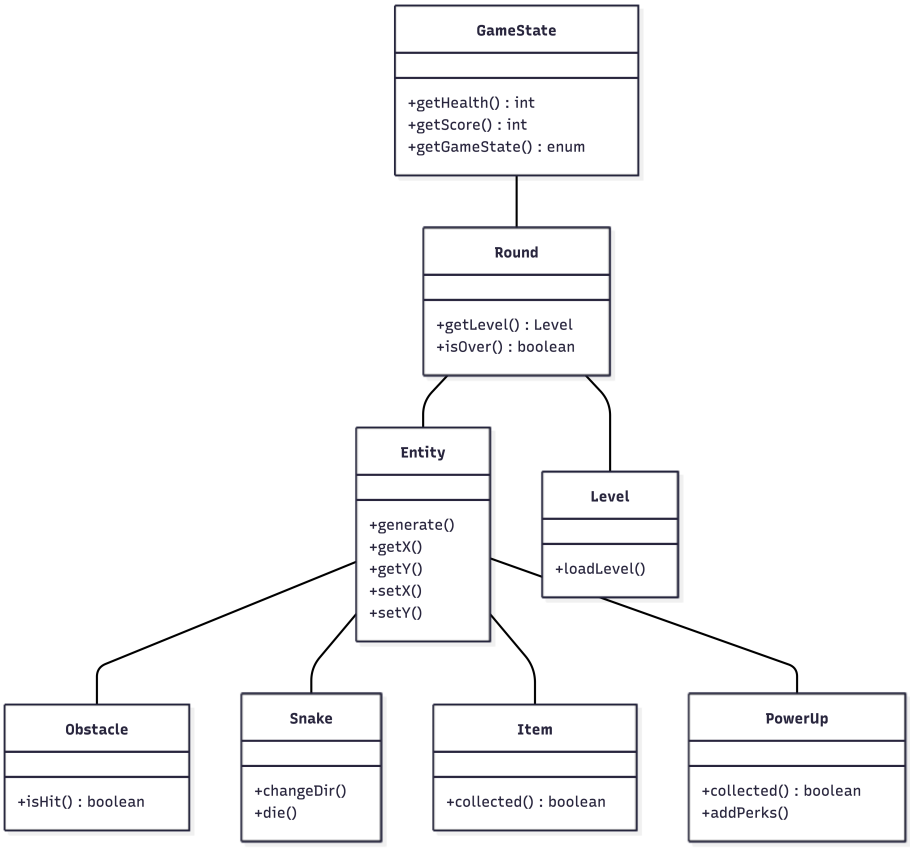


Figura 1.1: Rappresentazione UML del Modello del dominio

# Capitolo 2

# Design

## 2.1 Architettura

Il progetto *SnakeRunner* adotta il pattern architetturale **Model–View–Controller (MVC)**, al fine di garantire una chiara separazione delle responsabilità, una maggiore manutenibilità del codice e la possibilità di sostituire singole componenti senza impattare sulle altre.

## 2.2 Design dettagliato

### 2.2.1 Matteo Buldrini

Nel progetto ho contribuito principalmente allo sviluppo della componente grafica e alla componente del controller dell'applicazione, occupandomi della gestione dell'interfaccia grafica e della comunicazione tra utente e logica di gioco.

**- Problema:**
La componente grafica richiede la gestione di più schermate oltre a quella di gioco (menù, tutorial, opzioni) e ogni componente deve avere componenti comuni come bottoni, etichette e layout.

**- Soluzione:**
Per ridurre la duplicazione di codice e determinare comportamenti comuni, ho introdotto un'interfaccia *BasePanel* e una classe astratta *AbstractBasePanel*. L'interfaccia definisce metodi comuni di base a tutti i pannelli, la classe astratta implementa la logica condivisa. Questo approccio consente di centralizzare la logica comune, ridurre duplicazioni e facilitare l'estensione del sistema con nuovi pannelli.

**-- Pattern:**
Questa soluzione realizza il pattern **Template Method**:
la classe astratta *AbstractBasePanel* definisce la struttura generale dei metodi comuni ai pannelli e lascia alle classi concrete (*MenuPanel*, *OptionPanel*, *TutorialPanel* *GamePanel*) la possibilità di personalizzare dettagli specifici. In questo modo, la logica comune è centralizzata nella classe astratta, mentre le classi concrete implementano comportamenti particolari senza duplicare codice, rispettando il principio del Template Method.

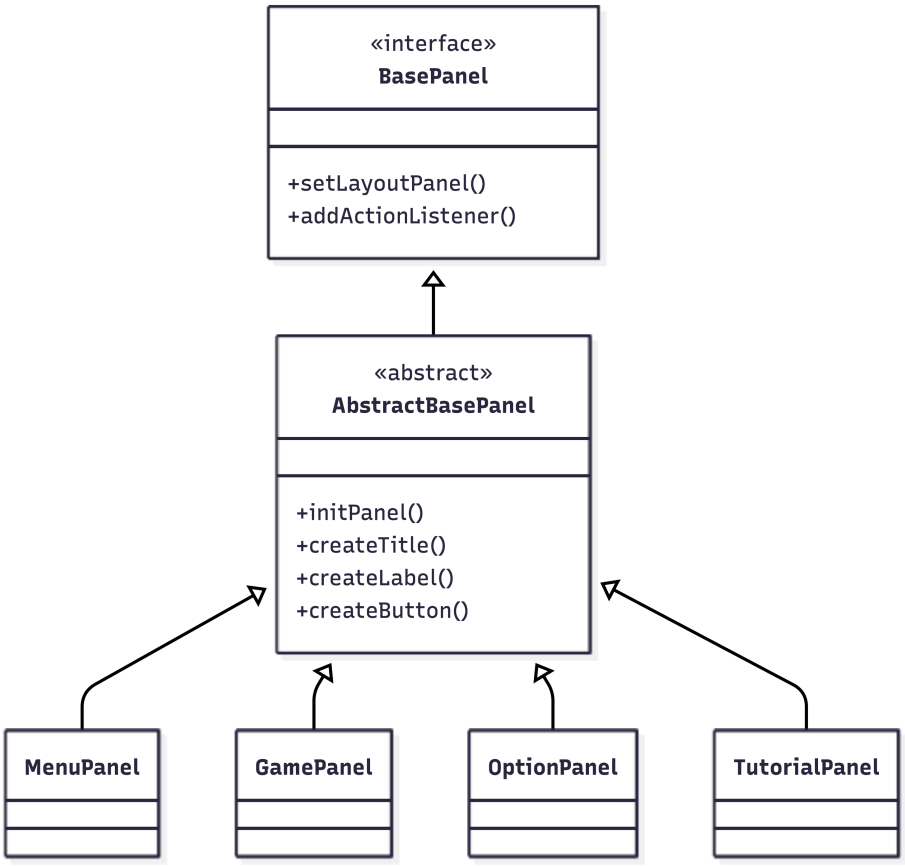


Figura 2.1: Rappresentazione UML della gerarchia dei pannelli (interfaccia BasePanel, classe astratta AbstractBasePanel e classi concrete.

**- Problema:**
Nell'applicazione ho diversi pannelli, senza un approccio centralizzato, la creazione dei pannelli sarebbe sparsa in tutto il codice, rendendo così il codice meno

manutenibile e presenterebbe duplicazioni.

**- Soluzione:**
E' stata introdotta la classe *PanelFactory*, ovvero una factory statica che centralizza la creazione dei pannelli.

**- Pattern:**
Questa soluzione realizza il pattern **Factory Pattern**, garantendo coerenza nella creazione di oggetti e centralizzando la creazione.

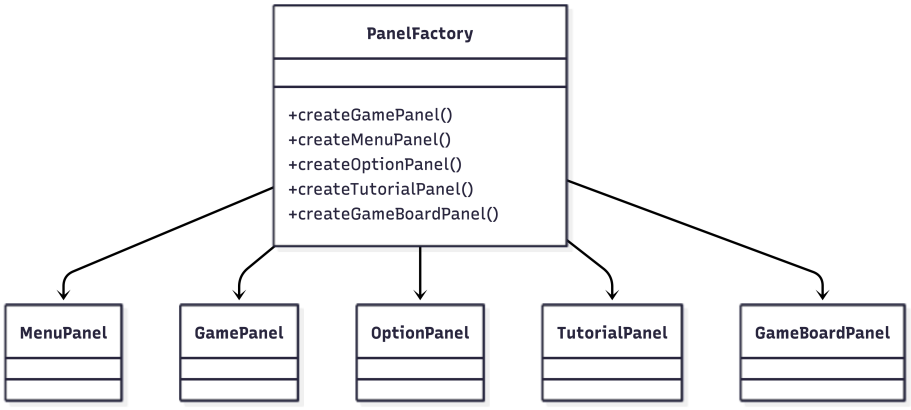


Figura 2.2: Rappresentazione UML del pattern Factory per la creazione dei pannelli

**- Problema:**
L'applicazione richiede la visualizzazione delle informazioni di stato in tempo reale per il giocatore, come:

- Punteggio
- Livello
- Tempo
- Vite

**- Soluzione:**
E' stata introdotta l'interfaccia *BaseHUD*, che definisce il metodo *setValue(int value)* per aggiornare i valori dei componenti dell'HUD. La classe astratta *AbstractBaseView* fornisce le funzionalità grafiche a tutti i componenti dell'HUD.

**- Pattern:**
Questa soluzione realizza il pattern **Template Method**.

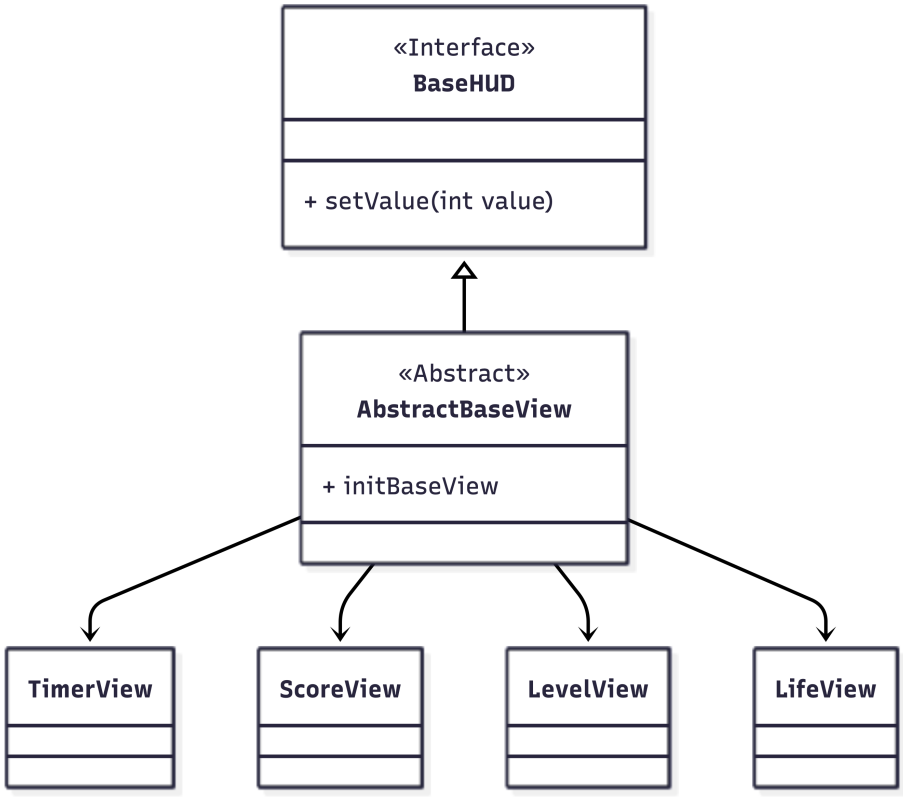


Figura 2.3: Rappresentazione UML del pattern Tempate Method per i componenti HUD.

**- Problema:**
Sempre a riguardo dei componenti HUD (vite, punteggio, timer, livello), bisogna centralizzare la creazione di questi componenti, per evitare che la creazione sia sparsa nel codice, creando duplicazioni e rendendo difficile la manutenzione.

**- Soluzione:**
Creazione di una classe **HUDFactory** in modo da centralizzare la creazione di tutti i componenti tramite metodi statici.

**- Pattern:**
Questa soluzione realizza il pattern **Factory Pattern**, garantendo coerenza nella creazione di oggetti e favorendo centralizzazione.

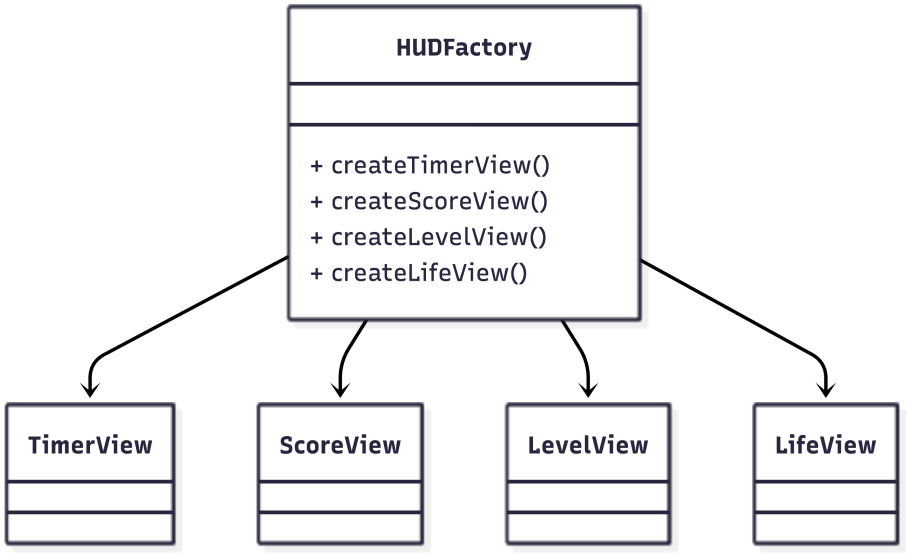


Figura 2.4: Rappresentazione UML del pattern Factory per gli elementi HUD

**- Problema:**
Come visto in precedenza, il gioco ha diverse schermate ed è necessaria :

- Gestione transizioni
- Avvio/Termine dell'applicazione

Senza un controller dedicato, la logica di navigazione sarebbe gestita nelle view rendendo il codice fragile e poco manutenibile.

**- Soluzione:**
Implementazione di un **NavigationController** che coordina le interazioni tra view e azioni dell'utente.

**- Pattern:**
Questa soluzione realizza il pattern **MVC**, occupandosi del **Controller**.

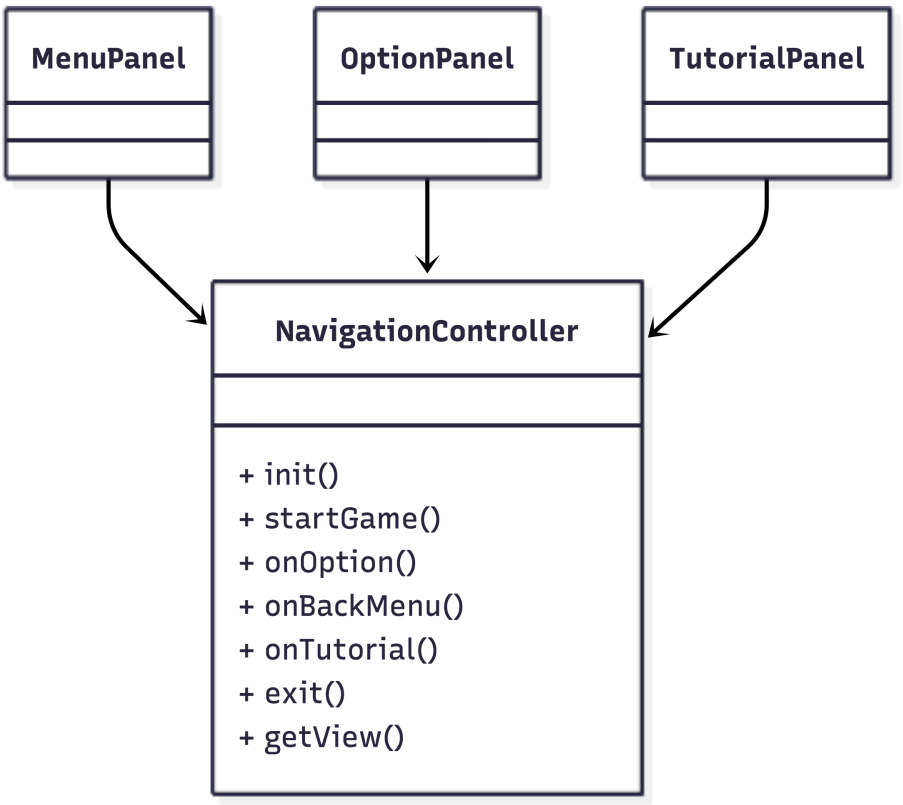


Figura 2.5: Rappresentazione UML del NavigationController

**- Problema:**
La logica di gioco (GameLoop, aggiornamento dello stato, gestioe di pause e caricamento livelli) non può essere gestita direttamente dalla view in quanto si creerebbero accoppiamenti e duplicazione di codice.

**- Soluzione:**
**GameController** che centralizza la logica di gioco, gestisce l'inizio del GameLoop, si occupa del pause/resume di gioco, si occupa di aggiornare HUD e aggiorna lo stato della partita.

**- Pattern:**
Questa soluzione realizza il pattern **MVC**, occupandosi del **Controller**.

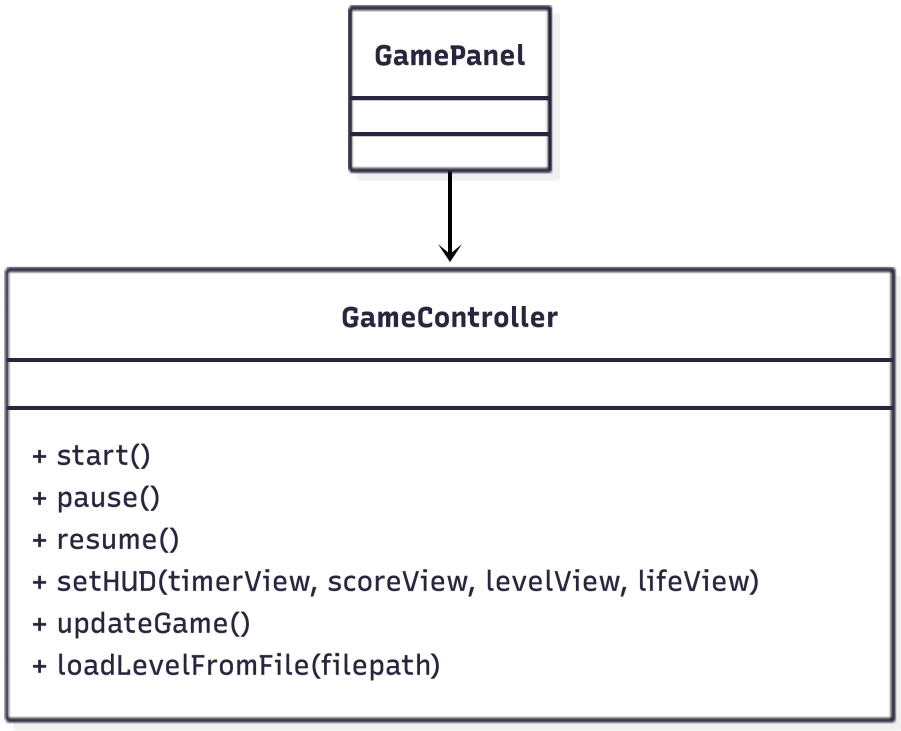


Figura 2.6: Rappresentazione UML del GameController

**- Problema:**
La view ha bisogno di conoscere lo stato del gioco : posizione dello snake, cibo, chiavi, porte. Ricordiamo che l'architettura MVC non permette alla view di accedere direttamente, quindi ho bisogno di un mediatore per non rompere l'architettura.

**- Soluzione:**
Il **WorldController** fa da mediatore tra view e model, esponendo solo metodi per leggere lo stato del gioco. La view interroga il controller per ottenere le informazioni necessarie per poi poter disegnare nella mappa di gioco i vari elementi. Questo garantisce indipendenza tra Model e View.

**- Pattern:**
Questa soluzione realizza il pattern **MVC**, occupandosi del **Controller**.

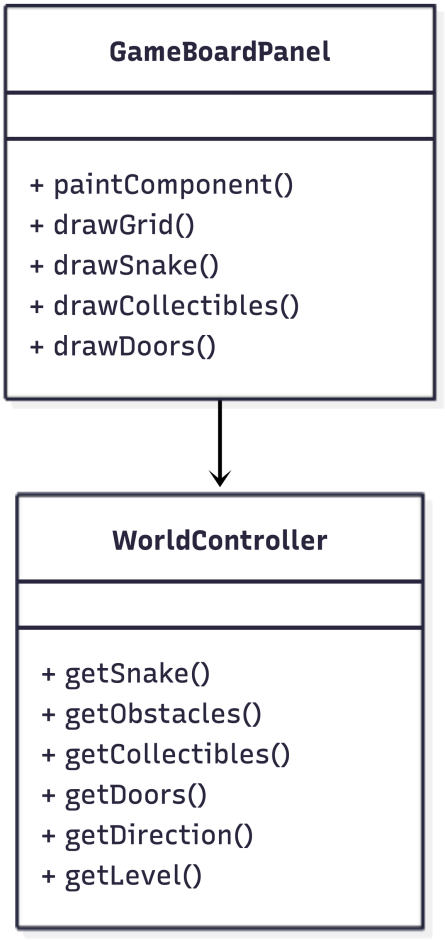


Figura 2.7: Rappresentazione UML del WorldController

### 2.2.2 Galli Mattia

Sono state implementati:

- il Level Loader per caricare i livelli da file di testo;
- la progressione e il reset dei livelli e dei stati del gameloop;
- la generazione e la gestione di item e power-up con relativi effetti sul gioco;
- gestione del gameloop dell'applicazione.

**- Problema:**
Il gioco prevede diversi oggetti collezionabili (Food, Clock, Key, Life Boost), ognuno dei quali produce un effetto differente sullo stato del gioco. Il problema

è evitare una gestione centralizzata tramite strutture condizionali complesse come if/else o switch basati sul tipo dell'oggetto.

**- Soluzione:**
Modellare ogni oggetto collezionabile come un'implementazione dell'interfaccia Collectible, ogni oggetto quando "consumato" avrà un diverso effetto nel gioco semplicemente richiamando il metodo nel collectible desiderato. La soluzione adottata consente di aggiungere nuovi collectible senza modificare codice già esistente e stimola il riuso del codice.
Il model delega il comportamento all'oggetto, evitando accoppiamenti inutili.

**-Pattern:**
Questa soluzione realizza il pattern **Strategy**:
l'interfaccia Collectible rappresenta la stategie, il metodo consume è il punto di variazione del comportamento.

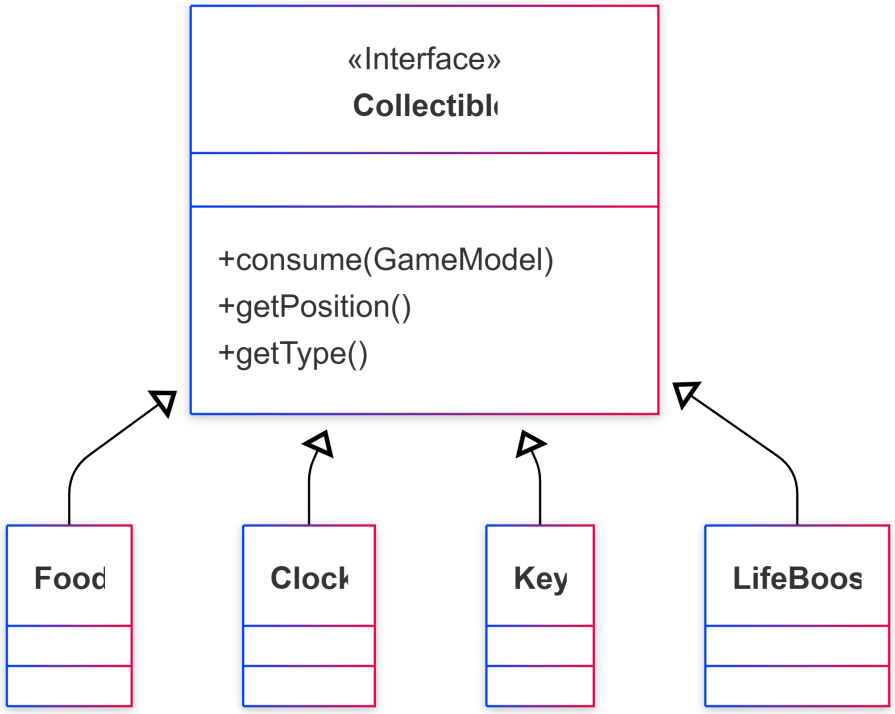


Figura 2.8: Rappresentazione UML del pattern Strategy per i Collectibles (Power-UP)

**- Problema:**
Il gioco prevede più livelli con configurazione differenti, ciascuno caratterizzato da:

- disposizione degli ostacoli;
- posizione di collectible di vario tipo;
- presenza di porte.

Il problema consiste nel definire un modo in cui si debba modificarre e aggiungere livelli senza ricompilare codice, separare la descrizione dei livelli dalla logica di gioco per tenere l'MVC il più puro possibile.

**- Soluzione:**
La soluzione consiste nell'utilizzare un file di testo per la descrizione dei livelli e una classe, LevelLoader, dedicata al loro caricamento. Il LevelLoader egge il file di testo e costruisce un oggetto LevelData contenente i dati logici del livello. Il GameModel infine riceverà solo un oggetto LevelData già pronto senza dover conoscere i formati dei file dei livelli.
La soluzione adottata consente invece di aggiungere nuovi livelli semplicemente creando nuovi file di testo, senza modificare il codice.

**-Pattern:**
Questa soluzione realizza il pattern **Factory**:
La classe LevelLoader agisce come una Factory, incapsulando la logica di creazione di LevelData.
Il GameModel riceve un oggetto già costruito e pronto all'utilizzo

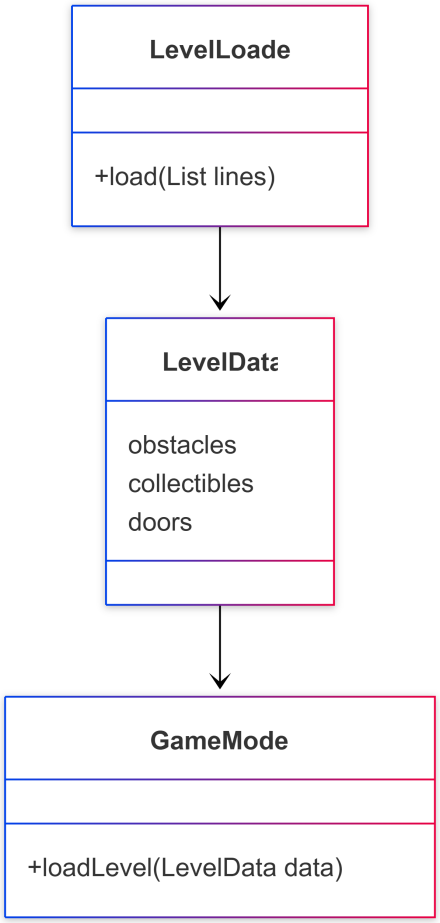


Figura 2.9: Rappresentazione UML di LevelLoader

**- Problema:**
Un videogioco richiede un meccanismo che aggiorni periodicamente lo stato del gioco, permettendo:

- il movimento continuo del serpente;
- il controllo delle collisioni;
- la presenza di effetti dopo il consumo dei collectibles;
- la gestione del passaggio di livello.

Il problema principale è decidere dove collocare il game loop evitando dipendenze che andrebbero ad intaccare l'MVC

- indipendenza del model dalla gestione del tempo;

- una chiara separazione delle responsabilità;
- semplicità nella gestione dello stato di gioco.

**- Soluzione:**
La soluzione adottata prevede l'implementazione del game loop nel controller, tramite un timer.
La scelta di collocare il game loop nel controller permette di mantenere il model completamente indipendente dalla gestione del tempo e di rendere la view una componente puramente passiva, facilitando la sostituzione della rappresentazione grafica senza modifiche alla logica di gioco.

**-Pattern:**
Questa soluzione implementa il **Controller Attivo**, in cui esso non si limita a reagire all'input, ma governa anche il ciclo di vita dell'applicazione.

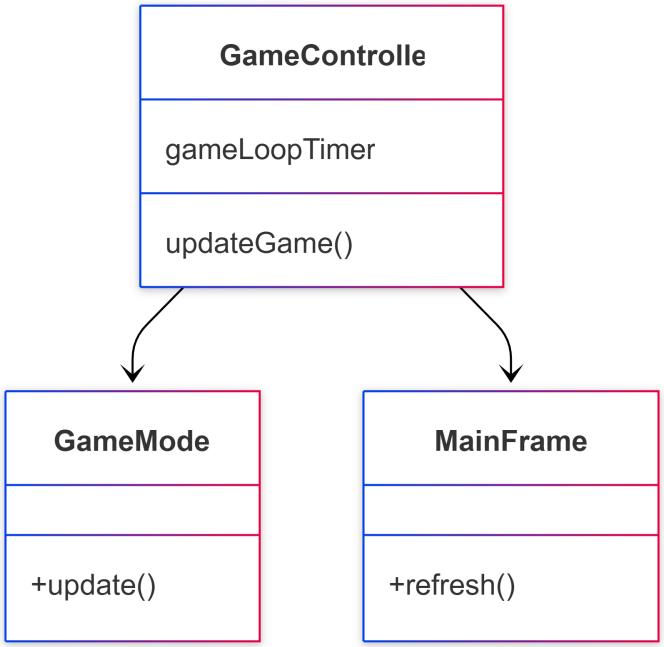


Figura 2.10: Rappresentazione UML del GameLoop

### 2.2.3 Soufi Hiba

I miei contributi all'interno del progetto sono stati :

- Modello del serpente
- Movimenti del serpente

- Gestione degli input da tastiera
- Vite e Game Over

**- Problema:**
Rappresentare il movimento fluido del corpo del serpente. Il sistema non solo deve gestire la posizione spaziale del serpente, ma anche essere in grado di fornire alla View le giuste informazioni grafiche su come rappresentare l'oggetto.

**- Soluzione:**
Il serpente è modellato tramite la classe Snake che gestisce una lista di oggetti. Il movimento del serpente segue la stessa logica di un "treno" : la nuova posizione della testa viene calcolata in base alla direzione attuale e inserita all'indice 0 della lista, mentre l'ultimo elemento viene rimosso per mantenere la dimensione fissa del serpente. Grazie al metodo updateLogic() ogni segmento analizza la posizione relativa del pezzo precedente e successivo. Attraverso il metodo RelativeDirection(), il sistema assegna a ogni elemento un SegmentType e le direzioni toHead e toTail. Permettendo alla View di distinguere tra segmenti dritti, curvi e orientando corretamente la grafica. Ogni segmento mantiene così uno stato grafico (tipo e orientamenti) ricalcolato in updateLogic(), così la View non deve fare calcoli geometrici.

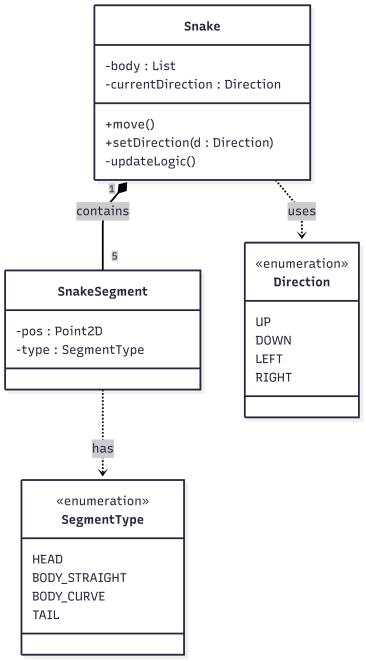


Figura 2.11: UML semplificato del modello Snake: composizione con SnakeSegment e uso di Direction/SegmentType.

**- Problema:**
Fornire dati sufficienti per il rendering senza appesantire il calcolo della View.
**- Soluzione:**
Si è fatto modo che ogni pezzo del serpente (SnakeSegment) contenga già le informazioni utili per essere disegnato : la posizione, il tipo di pezzo. Inoltre ogni segmento è a conoscenza della direzione dei pezzi vicini. La View legge questi dati e sceglie subito lo sprite giusto e l'orientamento corretto.

**- Problema:**
Gestione degli input da tastiera. All'inizio del progetto la gestione dei tasti era stata affidata ad un KeyListener. Questa funzione però presentava due limiti, ovvero un problema di focus (il keylistener funziona solo se la finestra è attiva e in primo piano, il serpente smetteva di rispondere ai comandi) e soprattutto per poterla usare inizialmente era stato utilizzato un downcast che andava contro i principi Model-View-Controller.
**- Soluzione:**
Ho sostituito i KeyListener con i Key Bindings, implementandoli nel GamePanel tramite il metodo setupKeyBindings(), separando la logica di gioco dal rendering. Tramite l'utilizzo delle InputMap e delle ActionMap(es. con condizione WHEN_IN_FOCUSED_WINDOW) di Java Swing, è possibile catturare i tasti a prescindere da quale componente abbia il focus in quel momento e soprattutto rispetta l'MVC, la View si limita ad intercettare il tasto e inviare un comando, senza dover conoscere i dettagli interni del Controller tramite downcast. Command Pattern :associa a ciascun tasto un "comando" incapsulato in un'azione.

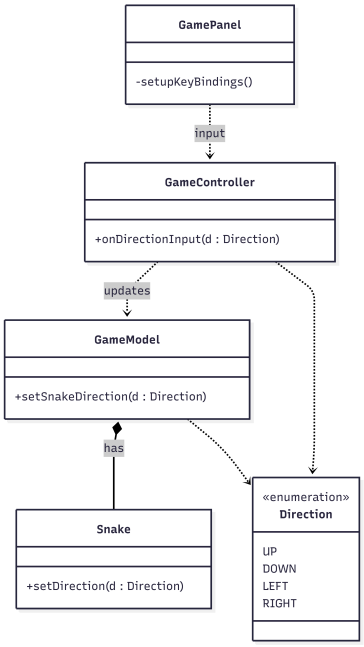


Figura 2.12: UML semplificato del della gestione input: GamePanel cattura i tasti (KeyBindings) e inoltra i comandi al GameController, che aggiorna GameModel e la direzione dello Snake.

**- Problema:**
Gestire vite e fine partita in modo consistente e centralizzato, evitando che Controller o View debbano controllare direttamente variabili interne o replicare la condizione di Game Over in più punti.
**- Soluzione:**
In GameModelImpl è stato introdotto lo stato lives e sono stati implementati i metodi getLives() e isGameOver(). getLives() espone il numero di vite rimanenti per HUD/View, mentre isGameOver() incapsula la regola di fine partita (lives $<= 0$), così le altre componenti devono solo interrogare il Model senza inserire logica di controllo duplicata.

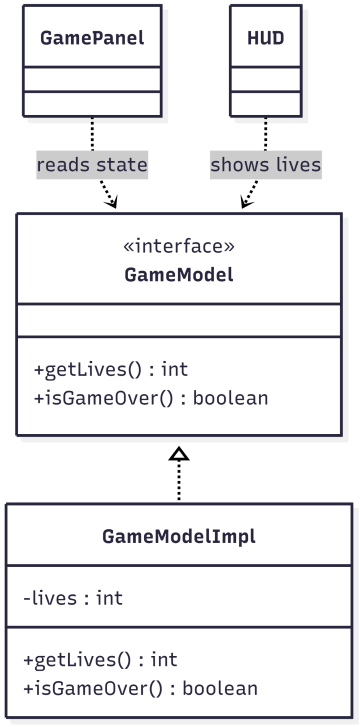


Figura 2.13: UML semplificato della gestione "vite e game over": GameModelImpl mantiene lo stato lives ed espone getLives() e isGameOver(), letti dalla View (HUD/GamePanel) per mostrare le vite e gestire la fine partita.

### 2.2.4 Ince Betul

All'interno del progetto mi sono occupata dello sviluppo della logica fisica e delle regole di terminazione dei livelli. In particolare, ho implementato:

- Gerarchia delle entità *Entity, Obstacle, Door, Flag*
- Gestione delle collisioni
- Gestione delle condizioni di vittoria

**- Problema:**
Il gioco richiede la gestione di diversi oggetti sulla griglia, di cui alcuni semplici punti, altri invece occupano aree rettangolari oppure possiedono degli stati variabili. Definire classi separate senza una base comune porterebbe a una forte ridondanza del codice.

**- Soluzione:**
Ho adottato una gerarchia basandomi sull'ereditarietà. La classe *Entity* fornisce la logica di posizionamento. La sottoclasse *Obstacle* estende questa logica introducendo il concetto di area occupata.

**- Pattern:**
Ho utilizzato il principio del Template Method attraverso l'ereditarietà. La classe base Entity definisce la struttura comune, mentre le sottoclassi specializzano il comportamento specifico.

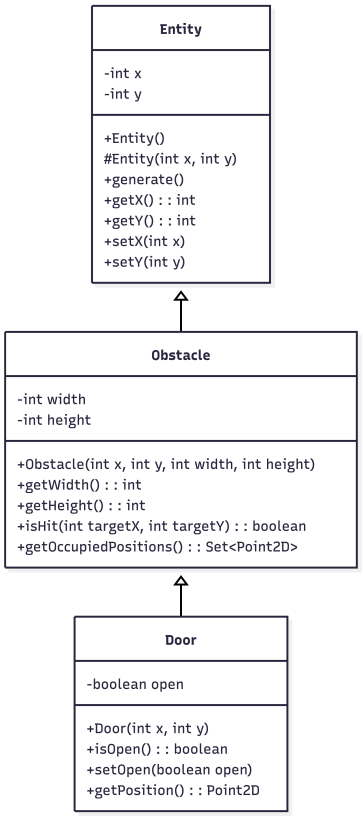


Figura 2.14: Gerarchia delle classi Entity Obstacle e Door

**- Problema:**
Il sistema deve verificare costantemente se lo Snake collide con tre tipologie di elementi distinti: se stesso, gli ostacoli e le porte, che possono essere chiuse o aperte. Gestire tutti questi controlli in punti diversi del codice renderebbe il codice disorganizzato e difficile da gestire.

**- Soluzione:**
Nel GameModelimpl ho centralizzato la logica in un metodo *checkCollisions()* dove controllo in sequenza il corpo dello snake, gli ostacoli e infine le porte. La collisione con la classe *Door* è stata resa condizionale, infatti la porta viene considerato un ostacolo solo se lo stato della porta non è "open".

**- Pattern:**
In questo punto ho applicato il pattern Mediator, infatti invece di far controllare

le collisioni ai singoli oggetti, il tutto è stato centralizzato nel *GameModelImpl*.

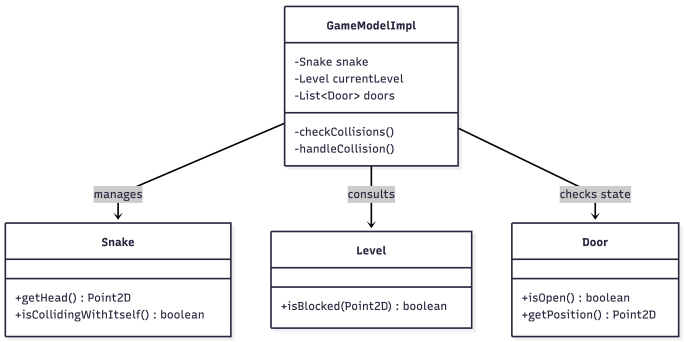


Figura 2.15: Rappresentazione UML del sistema di gestione delle collisioni

**- Problema:**
Le condizioni di vittoria variano a seconda del livello di gioco: nel primo, l'obiettivo è la raccolta di tutti i *Collectibles*, ovvero mele, mentre nei successivi è il raggiungimento di un traguardo fisico, identificato con una bandierina. Nonostante ci sia questa disuguaglianza tra gli obiettivi, il nucleo del gioco deve rimanere invariato.

**-Soluzione:**
Per risolvere questa question, ho implementato l'enumerazione *VictoryCondition* all'interno del codice. Nel metodo *checkCollectibles()*, viene verificata la condizone attiva per il livello corrente, se questa è impostata su *COLLECT_ALL_FOOD*, allora viene effettuata una scansione dei *Collectibles*. Nel caso in cui la lista risulta vuota, confermando dunque la raccolta di tutti gli elementi della mappa, viene stabilita la vittoria.

**- Pattern:**
Per questo problema ho utilizzato una versione semplificata del pattern Strategy. Utilizzando l'enumerazione *VictoryCondition* è possibile evitare di alterare il *GameLoop*

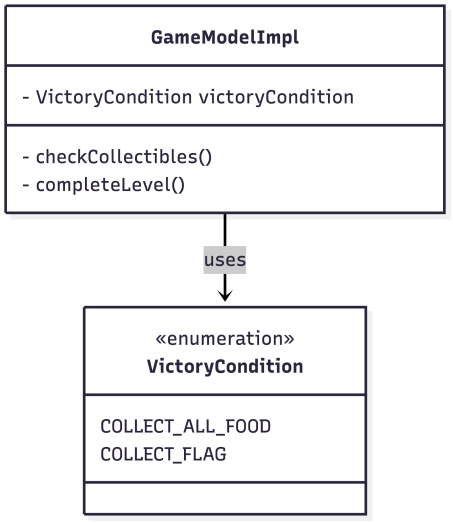


Figura 2.16: Rappresentazione UML della gestione delle condizioni di vittoria tramite l'enumerazione VictoryCondition

# Capitolo 3

# Sviluppo

## 3.1 Testing automatizzato

Sono stati realizzati test automatici utilizzando JUnit 5, focalizzati a garantire il corretto funzionamento del software dopo un aggiornamento, evitando degressioni.

-Test che controlla che power up, item, trappole non combaciano tra di loro
-Test che verifica il corretto funzionamento delle collision
-Test che verifica che la testa del serpente sia nella posizione iniziale
-Test che verifica che il movimento funzioni correttamente
-Test che verifica che lo snake non si giri di 180 gradi

### 3.1.1 GameModel

Sulla logica del Model, in particolare, sono stati testati il corretto aggiornamento dello score, l'applicazione degli effetti di gioco (come il rallentamento temporaneo) e il reset dello stato del modello.

### 3.1.2 CollisionTest

Sulla logica delle collisioni e del posizionamento degli elementi sono stati eseguiti test controllando il comportamento dello Snake al contatto con diverse entità, verificando la perdita di vite contro ostacoli o porte chiuse e la corretta raccolta dei collezionabili.

## 3.2 Note di sviluppo

### 3.2.1 Galli Mattia

- **Lambda expressions:**
  Utilizzata nel gameloop. Permalink: `https://github.com/buldra2000/snake-runner/blob/d1bdf53a946ef36772d609b3a3289cf1aa10f606/src/main/java/snakerunner/controller/impl/GameControllerImpl.java#L203`

- **Progettazione e utilizzo di tipi generici personalizzati**
  `https://github.com/buldra2000/snake-runner/blob/f347eeccd26d19a65729684995d3f973e601a0e`
  `src/main/java/snakerunner/commons/Point2D.java#L1`

# Capitolo 4

# Commenti Finali

## 4.1 Autovalutazione e lavori futuri

### 4.1.1 Buldrini Matteo

**Lavoro svolto:**
Nel progetto ho contribuito con la creazione e l'implementazione dell'interfaccia grafica, dei vari pannelli, degli elementi HUD, del suono riprodotto da azioni del serpente, l'implementazione dei controller del programma, degli stati di gioco e dell'implementazione del collectible "bomb".

- **Punti di forza**

  1. Collaborazione: tutti i colleghi sono stati disponibili sia per quanto riguarda i vari colloqui, sia per quanto riguardo la tempestività di modifiche al codice.
  2. Puntualità: Ho cercato di essere più presente possibile per i miei colleghi, in modo da non dover far attendere mie modifiche che avrebbero poi rallentato il lavoro e, dunque, i tempi di consegna.

- **Punti di Debolezza**

  1. Testing per la componente grafica: Non sono stati realizzati test automatici per la parte grafica dell'applicazione. La GUI è stata testata principalmente in modo manuale durante lo sviluppo. In futuro sarebbe utile introdurre strumenti specifici per il testing delle interfacce grafiche per aumentare l'affidabilità del sistema.
  2. Organizzazione iniziale del codice: Durante le prime fasi di sviluppo alcune componenti non erano perfettamente modulari e presentavano accoppiamenti non ottimali. Successivamente è stato necessario refactoring per migliorare la separazione delle responsabilità e rispettare pienamente l'architettura MVC.

- **Possibili lavori futuri**

    1. Un possibile sviluppo futuro consiste nell'integrazione di librerie grafiche più evolute rispetto a Swing, che consentirebbero una gestione più flessibile dell'interfaccia utente, animazioni più fluide e un miglioramento complessivo dell'esperienza visiva.
    2. Introduzione di diverse modalità di gioco.
    3. l'implementazione di un sistema di salvataggio dei progressi.

### 4.1.2 Galli Mattia

**Lavoro Svolto**:
Nel progetto ho contribuito con la creazione e l'implementazione dei power-up, la gestione del gameloop dell'applicazione e la generazione e il caricamento dei livello e la loro gestione da parte del game model.

- **Punti di Forza**

    1. Collaborazione: Collaborazione attiva con tutti i componenti del gruppo, per riuscire a risolvere al meglio le difficoltà incontrate.
    2. Puntualità: Ho lavorato al meglio per garantire le tempistiche di consegna concordate.

- **Punti di Debolezza**

    1. Testing: Potevo utilizzare i testing in maniera migliore all'inizio del progetto per garantire un altro livello di sicurezza durante l'update del codice.

- **Possibili lavori futuri**

    1. Aggiunta di Boss di fine livello.
    2. Aggiunta di nuove funzionalità o nuovi ostacoli magari dinamici.
    3. Generazione di livelli proceduralmente.

### 4.1.3 Ince Betul

**Lavoro Svolto**
Il mio contributo al progetto si è focalizzato sulla progettazione degli elementi come ostacoli, porte e il flag, quest'ultimo necessario per il completamento dei livelli successivi al primo. Inoltre, mi sono occupata della gestione del sistema di collisioni tra lo snake e i componenti come l'ostacolo e la porta, definendo così le logiche d'interazione fondamentali per le condizioni di vittoria e il superamento dei livelli.

- **Punti di Forza**

1. Sinergia di gruppo. Durante lo sviluppo, il gruppo ha mantenuto una comunicazione costante, garantendo disponibilità e in casi di necessità, ci siamo aiutati a vicenda.

- **Punti di Debolezza**

  1. Testing. Nel testing non sono compresi alcuni casi estremi.
  2. Vincoli geometrici degli ostacoli. Al momento gli ostacoli possono avere solo una forma rettangolare, rendendo il design dei livelli un po' ripetitivo.

- **Possibili lavori futuri**

  1. Implementazione di ostacoli dinamici che reagiscono alle azioni dello Snake e che si generano in modo randomizzato.
  2. Aggiunta di ostacoli di altre forme e non solo rettangolari

### 4.1.4 Soufi Hiba

**Lavoro Svolto** Mi sono occupata dello sviluppo del Modello del gioco, progettando la logica del serpente e il sistema di movimento "a treno". Ho implementato i vincoli direzionali per impedire la rotazione non ammessa di 180 gradi e ho gestito le vite del serpente e la logica di Game Over. Infine, ho ottimizzato la ricezione dei comandi sostituendo i comuni KeyListener con i Key Bindings, garantendo una gestione degli input più fluida, robusta e coerente con l'architettura MVC.

- **Punti di Forza**

  1. Il valore aggiunto di questa esperienza è stato il rapporto con i miei colleghi. L'ambiente di lavoro, basato sulla chiarezza e sul rispetto, ha permesso di affrontare le sfide tecniche con serenità. Questo clima di fiducia reciproca ha reso ogni fase della progettazione un'esperienza formativa e gratificante.

- **Punti di Debolezza**

  1. Limitazione nel level design.Il sistema di caricamento dei livelli da file di testo, pur essendo molto flessibile per mappe statiche, non supporta attualmente entità dinamiche o eventi scriptati, limitando la complessità del gameplay alla sola disposizione degli ostacoli fissi.

- **Possibili lavori futuri**

  1. Salvataggio dei punteggi
  2. Mappa dinamica

## 4.2 Difficoltà incontrate e commenti per i docenti

**Mattia Galli** Il corso di OOP molto probabilmente è uno dei più completi della triennale, offre un lavoro quasi più professionale che didattico; crea una forte connessione con il gruppo e aiuta a spronare il problem solving.

**Ince Betul** La realizzazione del progetto non è stato del tutto immediato e a volte ci ha chiesto una risoluzione di alcuni bug, sia logici che tecnici. Tuttavia, risolvere questi problemi è stato molto interessante perchè ci ha spinto a ragionare utilizzando una logica diversa da quella abituale.

**Matteo Buldrini** Il corso di Programmazione ad Oggetti (OOP) si è rivelato estremamente stimolante e ben strutturato. La trattazione dei concetti fondamentali, quali incapsulamento, ereditarietà e polimorfismo, è stata chiara e approfondita, con numerosi esempi pratici che ne hanno facilitato la comprensione. Nonostante l'efficacia didattica, il corso presenta un livello di difficoltà elevato, soprattutto per quanto riguarda la gestione del progetto finale, che richiede una solida comprensione dei concetti teorici e una notevole capacità di applicarli in contesti complessi.

**Soufi Hiba** Il corso di Programmazione ad oggetti è sicuramente tra i più impegnativi, ma la realizzazione del progetto mi ha fatto vivere un esperienza di lavoro in team che mi ha arricchito molto.

## 4.3 Guida Utente

Il gioco è un imitazione del famoso "Snake". Una volta aperto verrà mostrato un menù con 3 bottoni : gioca, opzioni e esci.

- Gioca : avvia il gioco.
- Opzioni : si può selezione se riprodurre o meno i suoi, inoltre è possibile vedere il tutorial di gioco.
- Esci : esci dal gioco.

Una volta avviato il gioco, comparirà a schermo la griglia, il serpente e i vari item di gioco, in più verranno visualizzati gli elementi HUD come punteggio, vite, livello e tempo. Il serpente si muove tramite le frecce della tastiera. Durante il gioco saranno presenti due bottoni : Pause e Riprendi. Buona Fortuna!


Figura 4.1: Rappresentazione del menù

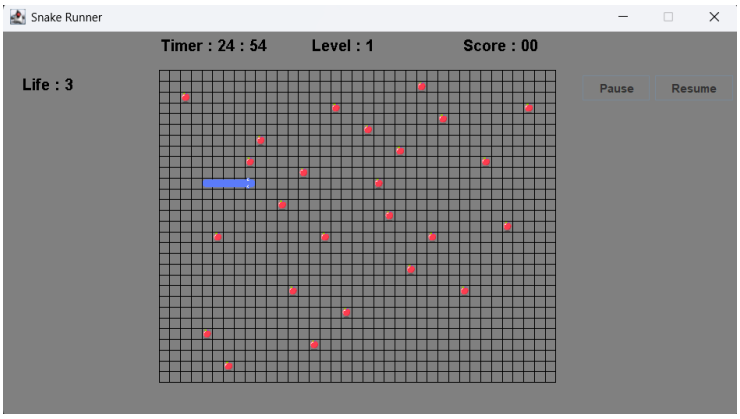


Figura 4.2: Rappresentazione Livello 1

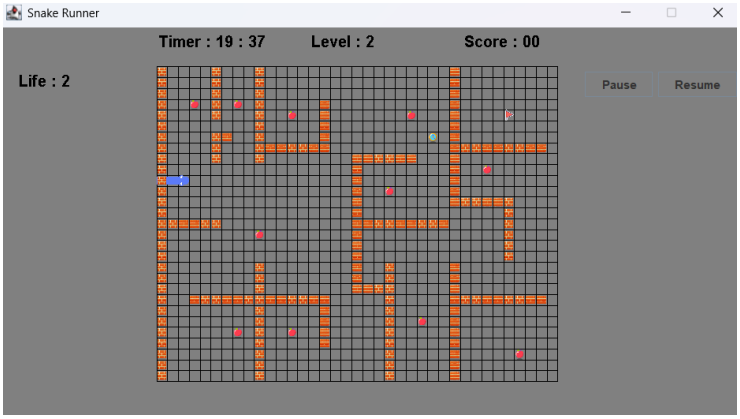


Figura 4.3: Rappresentazione del livello 2

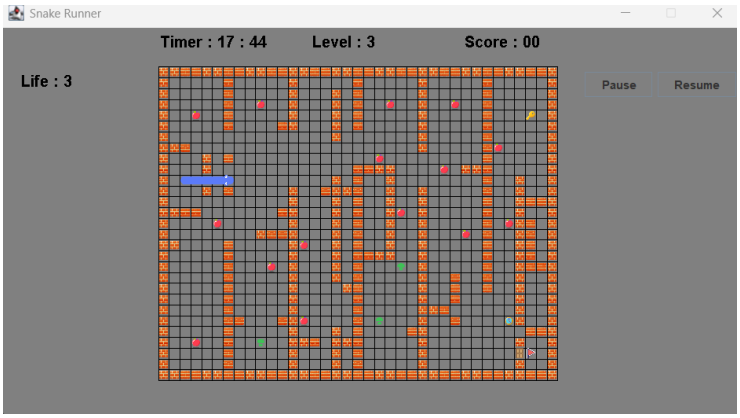


Figura 4.4: Rappresentazione del livello 3

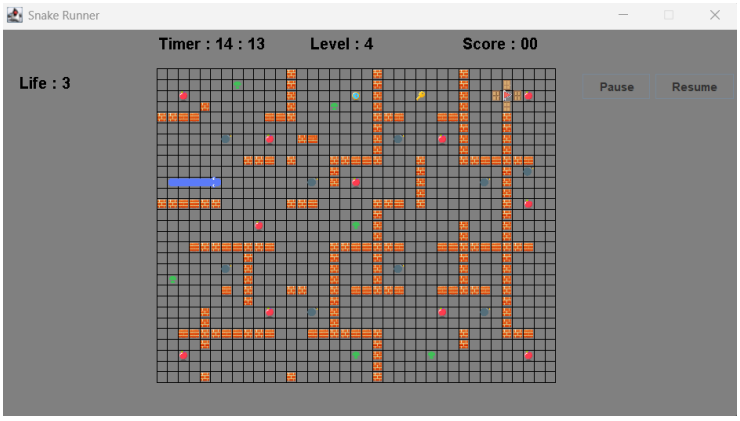


Figura 4.5: Rappresentazione del livello 4