Relazione per programmazione ad oggetti

“PUYO-BLAST”

Aisja Baglioni, Beatrice Di Gregorio, Chiara De Nardi, Federica Guiducci

15 Febbraio 2025

INDICE

CAPITOLO 1

ANALISI

1.1 DESCRIZIONE E REQUISITI

Il software in sviluppo è un videogioco ispirato alla saga di "Puyo Puyo" e allo spin-off Puyo Pop Fever in "Sega Superstar Tennis". Nel gioco, un giocatore deve sparare palline su una griglia popolata da Puyo (sfere colorate). L'obiettivo è far esplodere i gruppi di Puyo, accumulare punti e ottenere un numero minimo di stelle per superare i livelli. La partita termina se la griglia si riempie senza aver raggiunto il punteggio minimo richiesto, o se si raggiunge il massimo del punteggio.

REQUISITI FUNZIONALI

- **Menù di gioco e scelta dei livelli**: Il giocatore può selezionare i livelli da un'interfaccia dedicata.
- **Gestione dei comandi e degli input**: Il cannone è controllabile dal giocatore per mirare e sparare le palline.
- **Visualizzazione dello stato della partita**: Il punteggio, la griglia di gioco e le stelle ottenute vengono aggiornati in tempo reale.
- **Gestione della partita**: Monitoraggio dello stato del gioco, con aggiornamento della griglia e calcolo del punteggio.
- **Condizioni di fine gioco**: La partita termina in due casi:
    - Vittoria: il giocatore raggiunge il punteggio corrispondente a tre stelle.
    - Sconfitta: la griglia si riempie prima di raggiungere il punteggio massimo.
- **Movimento del cannone e della pallina**: Il cannone può essere spostato per mirare in diverse direzioni e sparare le palline.
- **Esplosione dei Puyo**: Quando viene colpito un Puyo che non ha vicini dello stesso colore, esplode singolarmente, altrimenti esplodono tutti i Puyo dello stesso colore a lui collegati.
- **Gestione delle conseguenze dell'esplosione**: Dopo un'esplosione, i Puyo sovrastanti cadono e il gioco aggiorna lo stato della griglia.
- **Implementazione della pausa**: Il giocatore può mettere in pausa la partita.
- **Animazioni e transizioni fluide**: Movimenti ed esplosioni dei Puyo devono essere animati con fluidità.
- **Congelamento temporaneo dei Puyo**: I Puyo hanno una piccola probabilità di subire un "freeze", rendendo impossibile scoppiarli a meno di un colpo speciale.
- **Sistema di punteggio**: Il punteggio si basa sulle esplosioni e determina le stelle ottenute (0-3 stelle per livello).

REQUISITI NON FUNZIONALI

- **Prestazioni**: Il gioco deve funzionare senza lag o rallentamenti, garantendo transizioni fluide tra le animazioni.
- **Sicurezza**: Devono essere evitate situazioni di crash o bug che possano interrompere la partita.
- **Portabilità**: Il software deve essere compatibile con diverse piattaforme di gioco.
- **Reattività**: Gli input del giocatore devono essere immediatamente registrati e sincronizzati con la logica di gioco.
- **Usabilità**: L'interfaccia deve essere intuitiva e accessibile, con controlli semplici da apprendere.

Il gioco si propone di offrire un'esperienza di intrattenimento dinamica, sfidando i giocatori a ottenere il massimo punteggio possibile tramite strategia e precisione nei lanci delle palline.

1.2 MODELLO DEL DOMINIO

Il gioco si aprirà con un menù dedicato alla gestione dei livelli, da cui sarà possibile anche accedere alla pagina dei comandi. Sarà strutturato in livelli a difficoltà crescente, ciascuno con un sistema di punteggio basato sull'esplosione dei Puyo dello stesso colore all'interno di una griglia. I Puyo cadranno in modo casuale nelle colonne non ancora riempite. Il giocatore potrà controllare un cannone a cui sarà legato un mirino, spostandolo con la tastiera per mirare e sparare proiettili, facendo esplodere i Puyo. Inoltre, sarà possibile mettere il gioco in pausa, bloccando temporaneamente la partita.

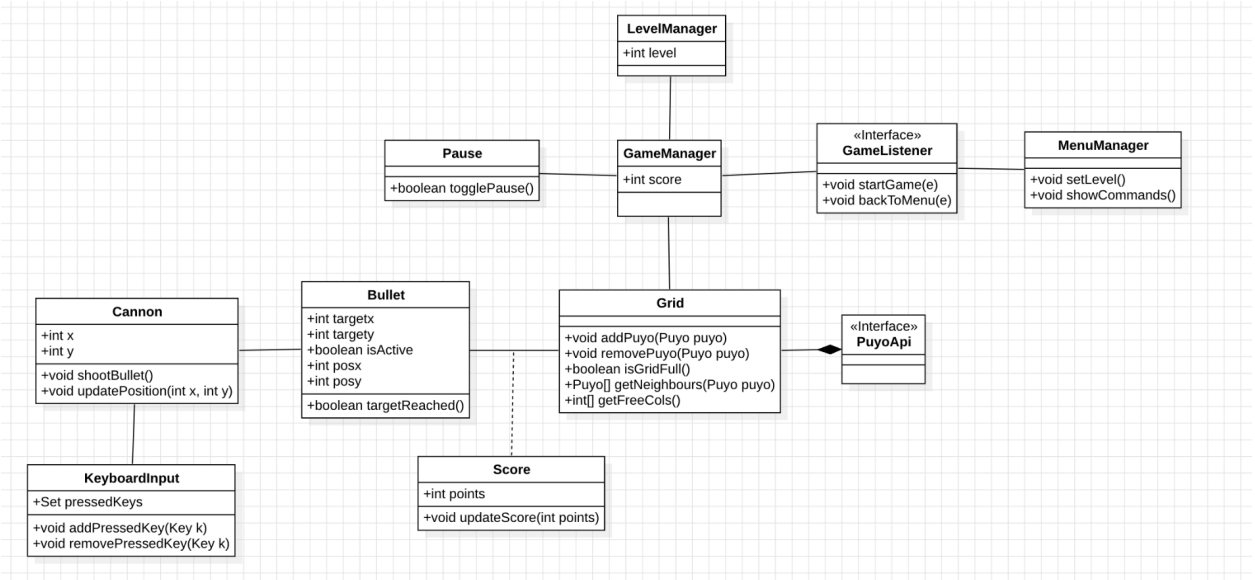


Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

CAPITOLO 2

DESIGN

2.1 ARCHITETTURA

PuyoPop è stato sviluppato seguendo un pattern architetturale con struttura MVC, con l'obiettivo di separare le preoccupazioni e migliorare la manutenibilità e l'estensibilità del sistema. In questo contesto, Model, View e Controller sono entità separate che interagiscono tra loro per gestire l'intero flusso del gioco. Abbiamo scelto di usare i Model alla base delle logiche di implementazione e aggiornamento degli attributi. I Controller hanno permesso l'aggiornamento dei modelli -delle loro posizioni assolute, i booleani- attraverso gli input dell'utente e talvolta secondo una logica basata su tick. Le View, infine, hanno renderizzato le immagini e la grafica rispettando gli aggiornamenti eseguiti dai Controller, ma generalmente utilizzando le variabili stipate nei Model.

Model: Degli esempi di model sono la classe Grid che gestisce i dati relativi alla griglia, il KeyboardModel che si occupa della raccolta degli input della tastiera e il CannonModel, che gestisce gli spostamenti del cannone e la sua logica. Queste classi sono responsabili della gestione dello stato e delle logiche interne.

View: Due esempi in questo progetto sono PuyoRenderer e ViewInterface. PuyoRenderer è una view dedicata all'aggiornamento real-time di grafica e animazione dei Puyo nella griglia, mentre la ViewInterface è l'interfaccia responsabile della visualizzazione delle immagini nelle loro corrette dimensioni e posizioni.

Controller: Il controller è composto da classi come BulletController e KeyboardController. Queste classi ricevono gli input dell'utente e aggiornano il modello (come il movimento dei Puyo o le logiche di esplosioni).

Per comodità, tutti i model sono stati raccolti da una classe ModelStorage, tutti i controller sono stati raccolti da una classe ControllerStorage, e tutte le view sono state raccolte da una classe GameView. I tre "storage" sono poi inizializzati nel costruttore del GameManager, ovvero il responsabile della gestione degli elementi specifici della partita.

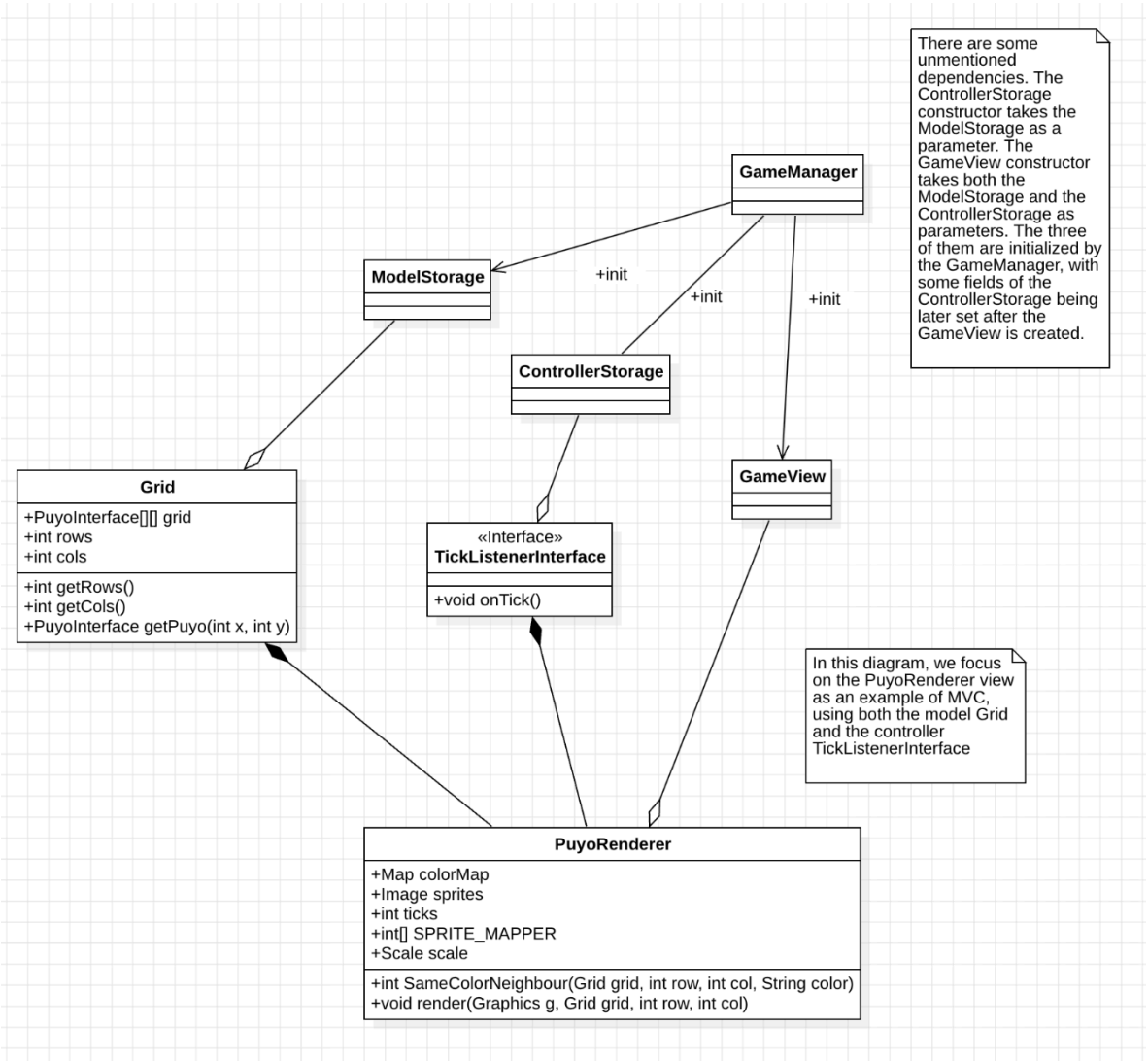


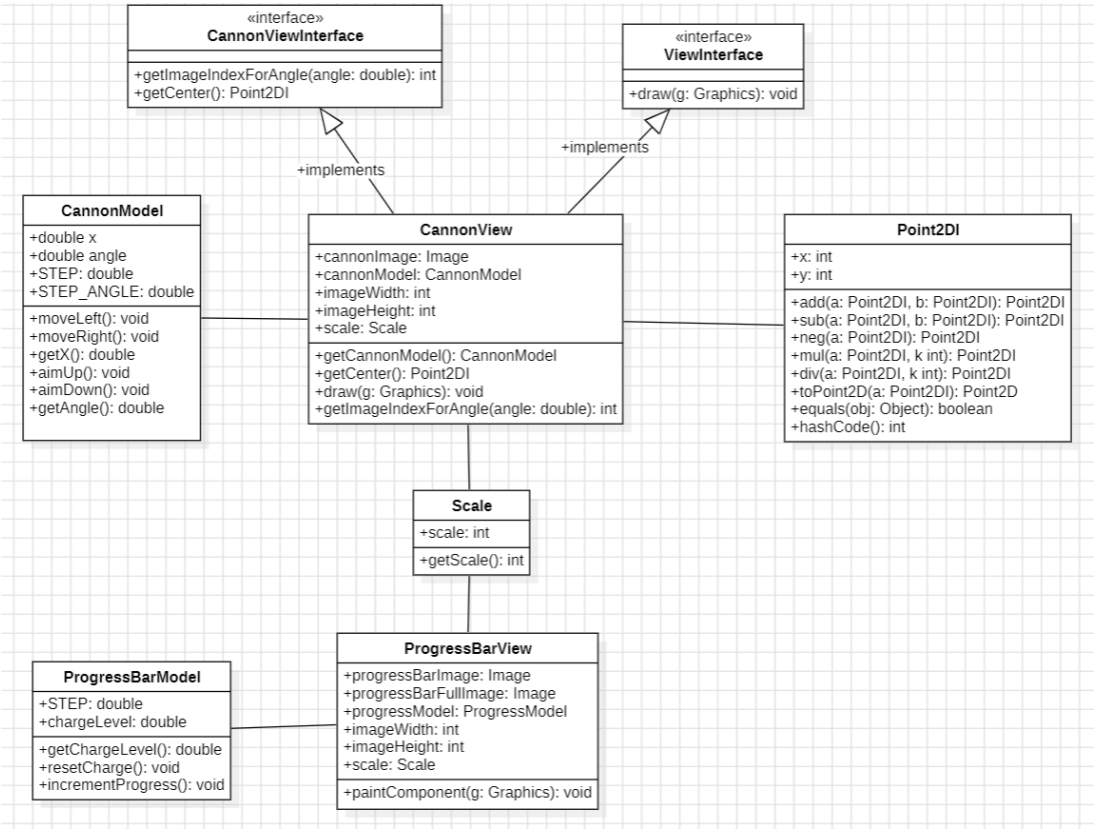
Figura 2.1: Schema UML architetturale di PuyoPop.

2.2 DESIGN DETTAGLIATO

Beatrice Di Gregorio:

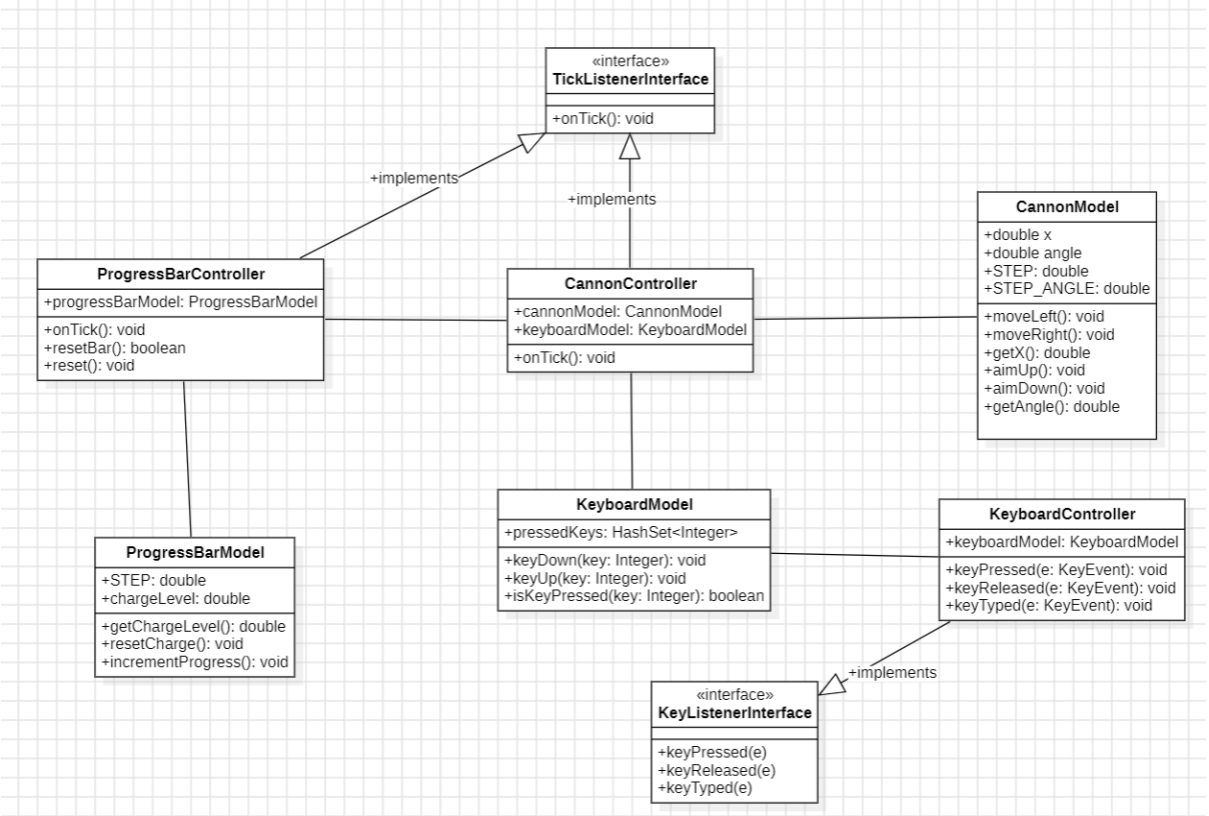
- Grafica del cannone e della barra di caricamento:
  - **Problema**: Creare una rappresentazione grafica del cannone che si adatti al gioco e fare in modo che il cannone venga visualizzato correttamente in base alla posizione e all'angolazione. Inoltre creare una rappresentazione visiva della barra di caricamento che deve essere aggiornata in tempo reale.
  - **Soluzione**: La grafica del cannone è stata gestita dalla classe CannonView, che si occupa di disegnare il cannone in base al suo stato (posizione e angolazione) all'interno della finestra di gioco. La classe CannonView carica una serie di immagini che rappresentano il cannone in diverse posizioni angolari e le seleziona dinamicamente in base all'angolo del cannone. Ogni immagine viene ridimensionata in

base alla scala della finestra in modo da garantire una corretta proporzione.
La classe ProgressBarView gestisce la visualizzazione della barra di caricamento, la quale viene disegnata tramite due immagini: una per la parte vuota e una per la parte piena. Tale classe calcola il livello di riempimento della barra in base al valore di carica del modello e disegna la parte piena della barra sovrapponendola a quella vuota.
Ogni volta che il modello del cannone o della barra di caricamento cambia, la vista corrispondente viene aggiornata per riflettere il cambiamento visivo. Sia il movimento del cannone, sia il progresso della barra di caricamento, sono sincronizzati con il passare dei tick di gioco mediante la funzione onTick implementata nei due controller.
In questo modo la grafica viene gestita in modo dinamico garantendo una visualizzazione fluida e reattiva degli elementi.



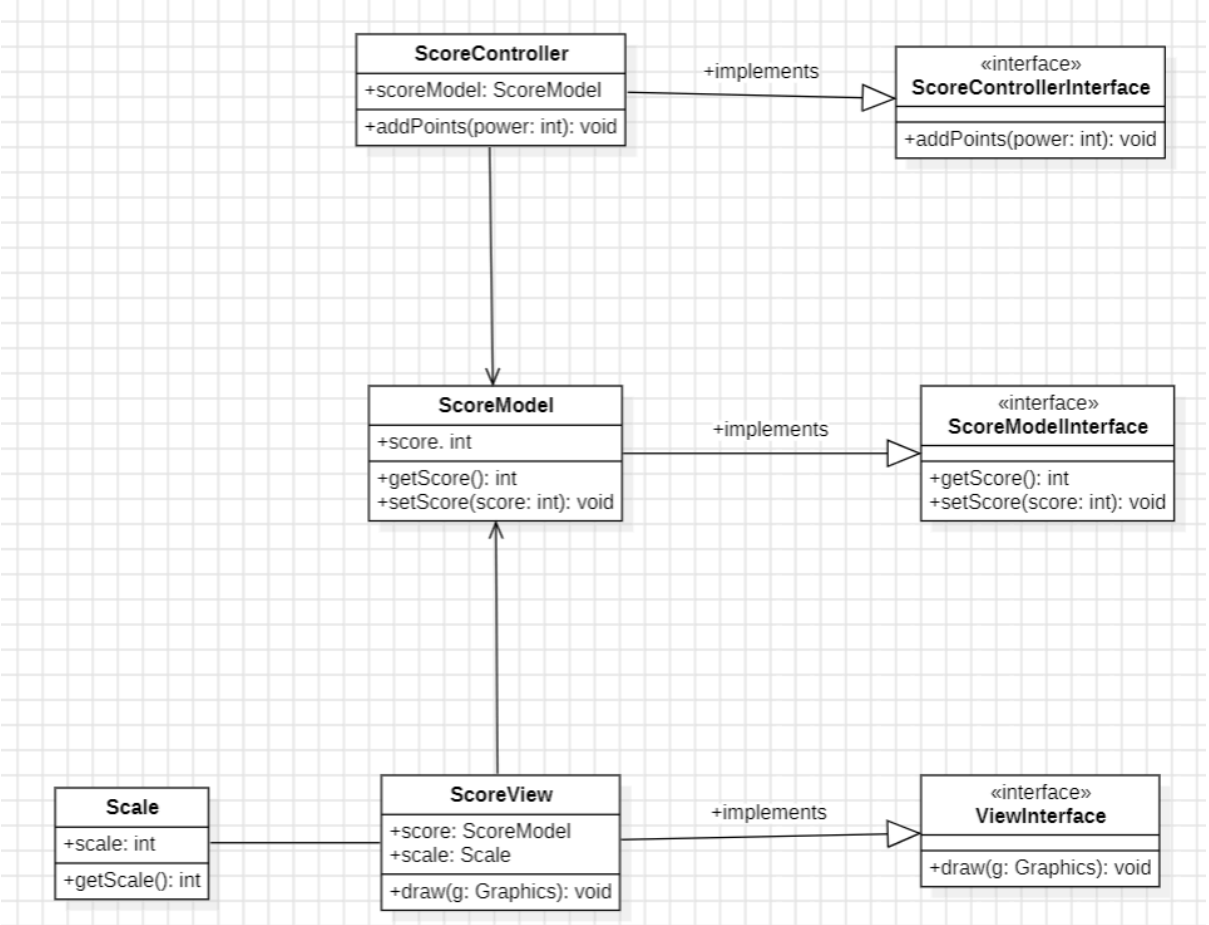
- Mappatura input del cannone:
  - **Problema**: Gestire e mappare gli input dell'utente per controllare la direzione del tiro in modo preciso così da permettere tiri strategici da parte dell'utente.
  - **Soluzione**: Per gestire gli input dell'utente, il sistema si basa su input da tastiera. La classe KeyboardController si occupa di registrare i tasti premuti dall'utente e di aggiornare lo stato del modello di tastiera (KeyboardModel), che tiene traccia dei tasti attualmente premuti. Ogni

pressione o rilascio di tasto viene registrata e tradotta in un'azione che influisce sulla direzione del cannone.
Ogni volta che l'utente interagisce con la tastiera, l'input viene processato dal CannonController, che aggiorna la logica del gioco e, successivamente, la visualizzazione.



- Visualizzazione punteggio:
  - **Problema**: Aggiornare e visualizzare il punteggio del giocatore in modo chiaro e in tempo reale, riflettendo i progressi fatti durante il gioco sulla base della rimozione dei Puyo e delle loro combo.
  - **Soluzione**: La visualizzazione del punteggio è stata gestita dalla classe ScoreView, che si occupa di disegnare il punteggio nella parte inferiore destra della finestra di gioco. La classe ScoreView utilizza un'istanza di ScoreModel per ottenere il valore attuale del punteggio e lo disegna sulla schermata in tempo reale.
    La posizione del punteggio nella finestra di gioco viene calcolata dinamicamente in base alla scala del gioco, in modo da adattarsi a diverse risoluzioni e mantenere una posizione coerente sullo schermo.
    L'aggiornamento del punteggio avviene a ogni frame, garantendo una visualizzazione fluida e sincronizzata con il gioco.
    La gestione dell'incremento del punteggio è affidata alla classe ScoreController e ogni volta che il punteggio viene modificato la Scoreview si occupa di ridisegnare il valore aggiornato sulla schermata.

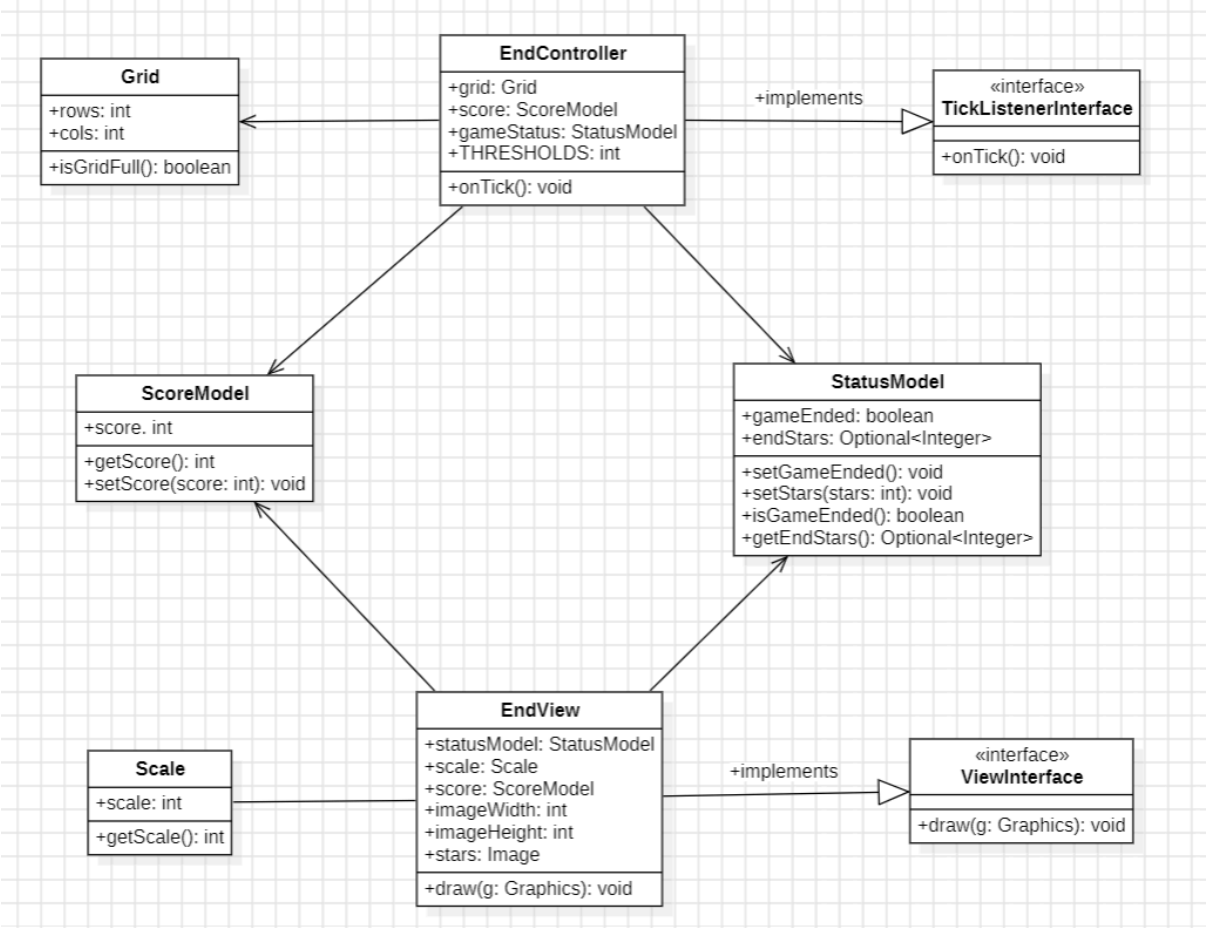
In questo modo il punteggio viene visualizzato in tempo reale e fornisce un chiaro feedback al giocatore.



- Gestione della condizione di sconfitta:
  - **Problema**: Gestire e visualizzare la condizione di sconfitta in modo che il giocatore possa essere informato correttamente e assegnare un numero di stelle sulla base del punteggio ottenuto.
  - **Soluzione**: La gestione della condizione di sconfitta è stata implementata tramite due componenti principali: la classe EndController, che monitora lo stato della partita e decide quando terminarla, e la classe EndView, che si occupa della visualizzazione della schermata di fine gioco.
    La classe EndController è responsabile della determinazione della fine del gioco. Ogni volta che avviene un aggiornamento (onTick), vengono controllate due condizioni:
    - Soglia del punteggio: se il punteggio raggiunge determinati valori (200, 350, 500 punti), vengono assegnate rispettivamente 1, 2 o 3 stelle. Se il punteggio massimo viene raggiunto, il gioco termina automaticamente.
    - Riempimento della griglia: se la griglia di gioco è piena, il gioco viene considerato terminato indipendentemente dal punteggio.

    Se una delle due condizioni è soddisfatta, il modello StatusModel viene aggiornato per segnalare la fine del gioco.

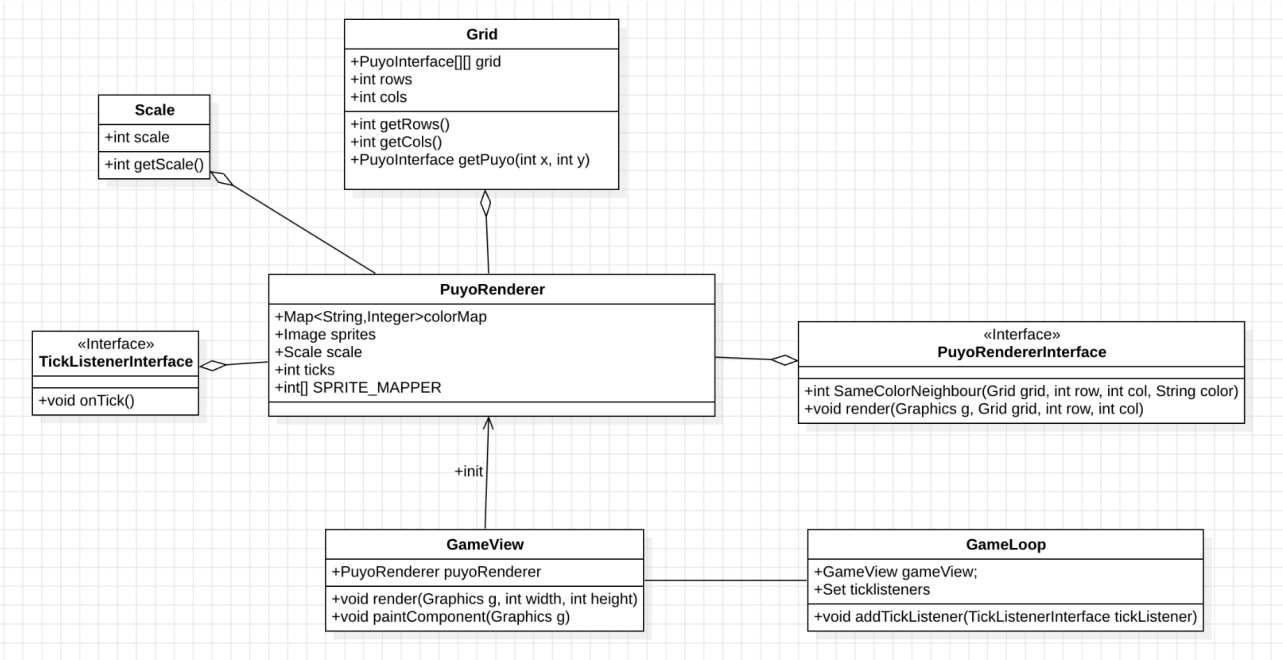
La classe EndView si occupa della rappresentazione grafica della schermata di fine gioco. Quando termina la partita, viene visualizzata una schermata con un messaggio di stato che indica se il livello è stato completato con successo o se è fallito, il punteggio del giocatore e un'immagine con le stelle corrispondenti alle performance (da 0 a 3 stelle).



Aisja Baglioni:

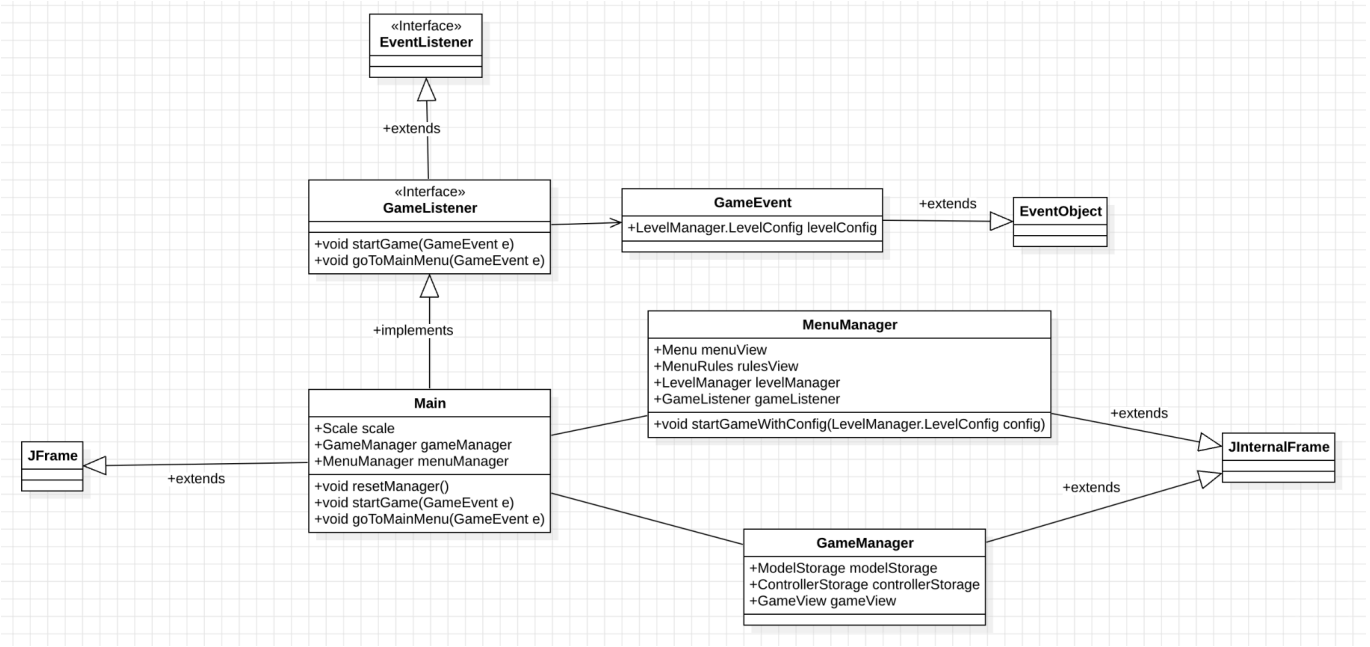
- Animazione degli sprite:
  - **Problema**: Gestire l'animazione ogni volta che un Puyo subisca un evento come l'esplosione, il freeze, o la presenza di vicini dello stesso colore.
  - **Soluzione**: L'animazione dei Puyo è gestita dalla classe view PuyoRenderer. Il PuyoRenderer aggiorna la rappresentazione grafica di ogni singolo Puyo a ogni tick del gioco, basandosi sulle sue proprietà e sul contesto in cui si trova all'interno della griglia.
    Ogni Puyo viene disegnato con uno sprite specifico, determinato innanzitutto analizzando la presenza di vicini dello stesso colore: questo avviene attraverso il metodo "sameColorNeighbour", che costruisce una maschera binaria rappresentante la configurazione dei vicini (sinistra, sopra, destra, sotto), successivamente utilizzata per

selezionare l'offset corretto di colonna nello spritesheet mediante la tabella “SPRITE_MAPPER”. La riga di riferimento viene invece scelta tramite una mappa “colorMap” che ha come chiave il colore del Puyo e come valore l’indice di riga. Se un Puyo non ha vicini dello stesso colore, gli viene applicata un'animazione casuale con una probabilità di 2/101 ogni cinque tick, simulando un effetto di movimento spontaneo, come la chiusura e apertura degli occhi, o in generale un cambio di espressione. Se un Puyo è stato colpito, il suo DeathClock sarà attivo. In questo caso, il Puyo viene renderizzato utilizzando una riga specifica dello spritesheet che mostra un'animazione a bolle prima di scomparire. Se invece il FreezeClock è attivo, il Puyo viene disegnato come intrappolato in un blocco di ghiaccio. L'animazione viene aggiornata e sincronizzata con il passare dei tick grazie alla funzione “onTick” che incrementa un contatore interno.

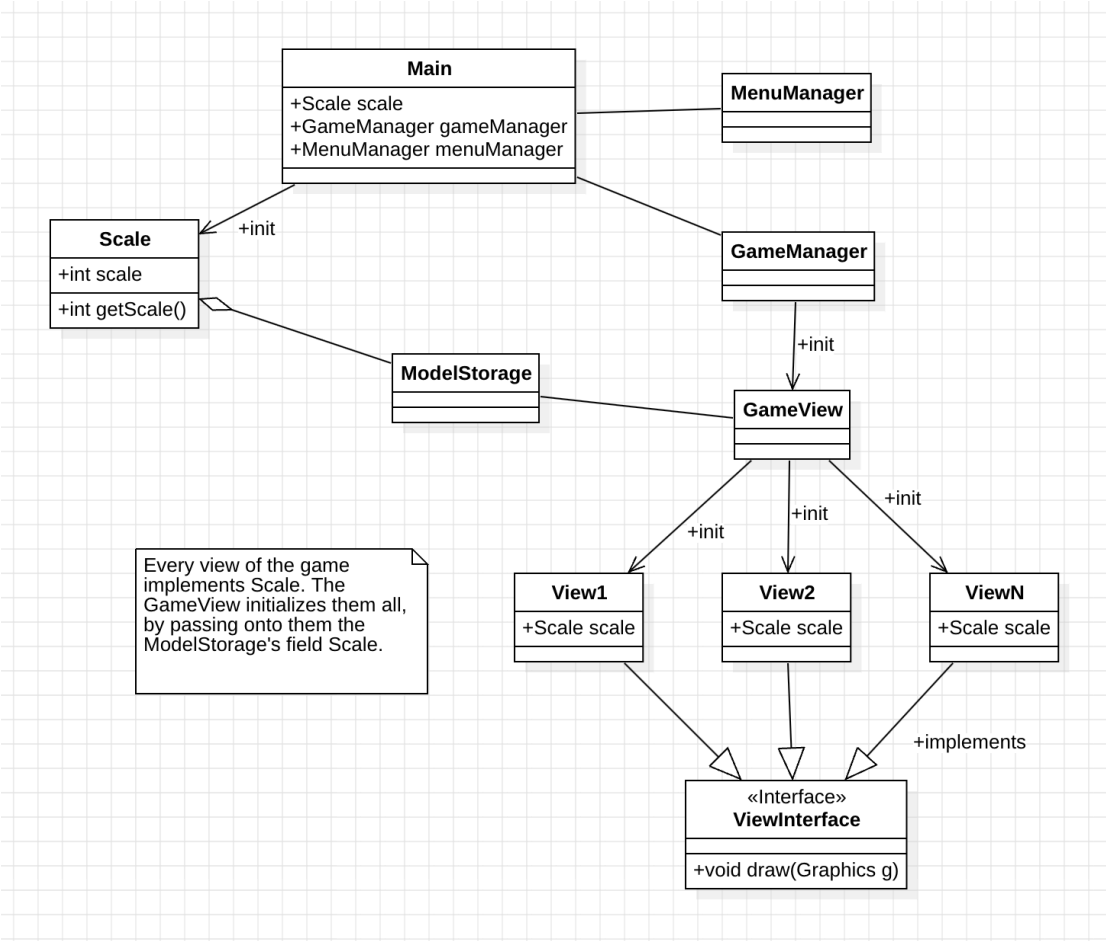


- Gestione delle transizioni:
  - **Problema**: Assicurarsi che le transizioni tra il menù principale e la schermata di gioco siano gestite correttamente.
  - **Soluzione**: Il Main collabora strettamente con l’interfaccia Game Listener, comprendente due metodi che, una volta ricevuto il GameEvent -classe rappresentante un evento nel gioco-, vengono chiamati a gestire le transizioni. Il GameEvent può trattenere o no il livello scelto prima dell’inizio della partita. Il caso in cui il livello venga mantenuto diventa utile ai fini dell’implementazione del bottone TryAgain, utilizzato per resettare il livello corrente direttamente da dentro il gioco. Decidere di non mantenere il livello è invece l’opzione di default nel caso in cui si esca verso il Main Menu.
    Entrambe le implementazioni possono essere utilizzate sia a partita ancora in corso, sia una volta che la partita è finita a causa di vittoria o

sconfitta. Lo scambio tra il menù e il gioco è separato efficientemente dalla presenza di due controller appositi, “MenuManager” e “GameManager”. Il Main possiede due metodi “startGame” e “goToMainMenu” per istanziare questi controller. Entrambi i metodi chiamano un terzo metodo “resetManager”. Quando richiesto, questo metodo rimuove lo stato del manager attualmente in funzione, così che nel passaggio tra menù e gioco non vi siano errori. Dopo il reset, startGame aggiungerà la schermata di gioco GameManager al JFrame e la renderà visibile, goToMainMenu farà lo stesso per il menù MenuManager. Questo meccanismo impedisce che il gioco mantenga dati in memoria tra un ritorno al menù e un nuovo avvio.

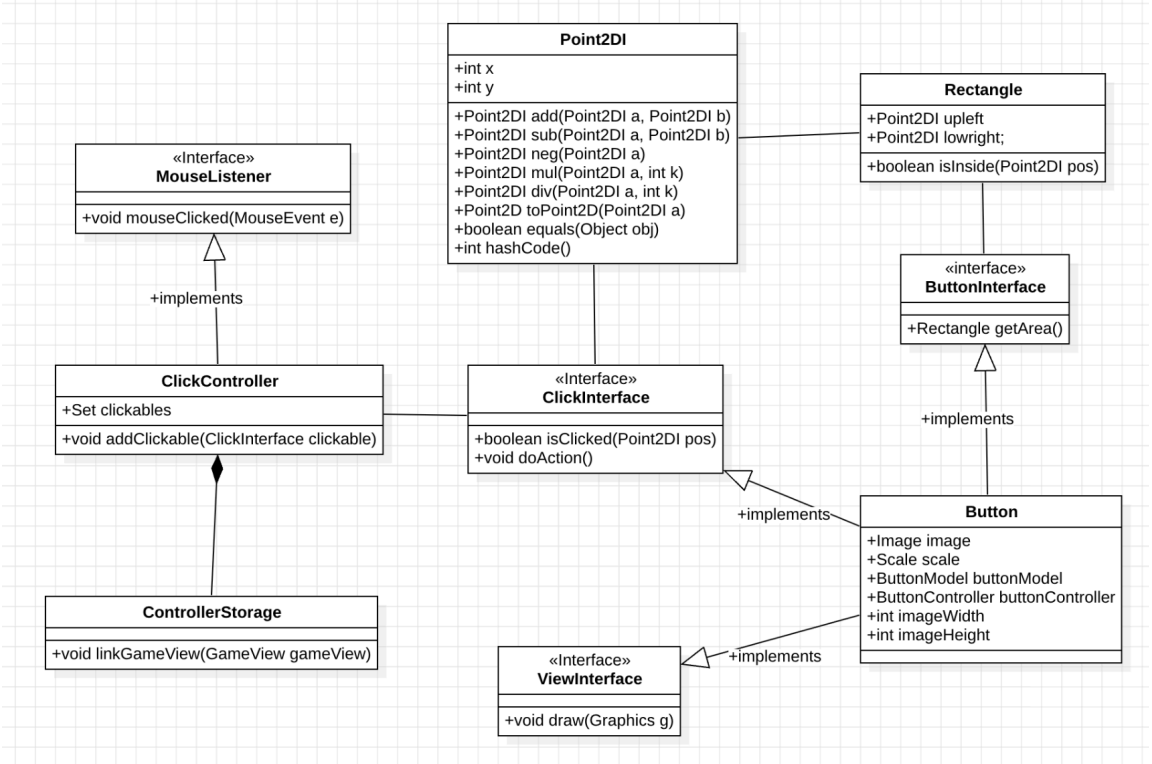


- Gestione della finestra desktop:
    - **Problema**: Visualizzare tutti gli elementi a schermo nelle corrette dimensioni e posizioni.
    - **Soluzione**: Il problema del posizionamento e dimensionamento degli elementi è stato gestito inserendo una classe Scale, che può essere inizializzata con parametro a piacere oppure con valore di default. Scale rappresenta sia la scalatura della grafica che la dimensione della finestra stessa, che nel nostro caso è una finestra quadrata, per centrare l’attenzione del giocatore sulla griglia, elemento chiave del gioco. Il Main imposta le dimensioni di Scale dinamicamente, a ¾ del minimo tra altezza e larghezza dello schermo dell’utente, ma si potrebbe anche scegliere un valore assoluto in pixel.



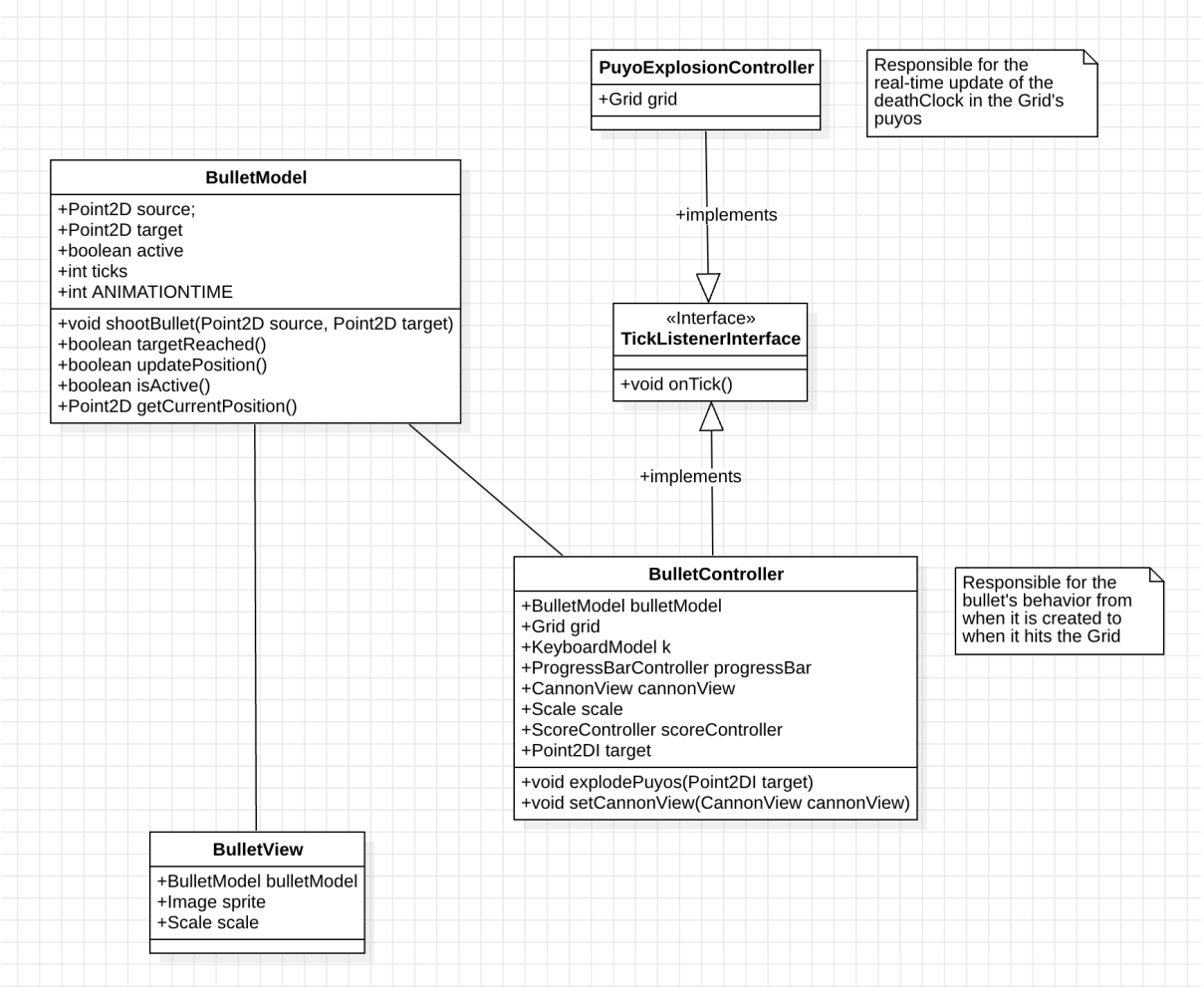
- Gestione degli input dei bottoni attraverso le loro view:
    - **Problema**: Dalla schermata di gioco deve essere possibile cliccare sull'immagine dei pulsanti Try Again, Main Menu e Pause/Resume per attivarne l'effetto. Essendo questi delle view e non dei JButton, come ne ricavo la porzione cliccabile nella finestra, e come capisco se l'utente ha cliccato in quella posizione?
    - **Soluzione**: Il problema è stato risolto grazie all'aiuto dei modelli custom di utility Rectangle e Point2DI, e di un ClickController. Queste classi lavorano insieme alle view dei bottoni per rendere l'area di un'immagine cliccabile attraverso un sistema di gestione degli input basato sul mouse. Per fare un esempio: la classe TryAgainView è responsabile della visualizzazione del bottone "Try Again", quindi carica l'immagine corrispondente e la disegna sulla schermata tramite il metodo "draw(Graphics g)", esattamente come le altre classi implementanti ViewInterface. La funzione draw di TryAgainView utilizza un oggetto Rectangle per definire l'area occupata dal bottone, per cui basta calcolare l'angolo in alto a sinistra e l'angolo in basso a destra. Il metodo isClicked(Point2DI pos) verifica poi se un dato punto (la posizione del click del mouse) si trova all'interno di quest'area, mentre doAction() invoca il metodo handleClick() del TryAgainController, il quale gestisce più concretamente la logica di ripristino del livello. La classe ClickController funge da gestore degli

eventi di input del mouse e mantiene un insieme di oggetti cliccabili (ClickInterface), tra cui TryAgainView, PauseView ed ExitView, ma renderebbe possibile anche aggiungerne degli altri inizializzandoli mediante la funzione “addClickable(ClickInterface clickable)”, chiamata dal ControllerStorage durante il setup della finestra di gioco. Quando il giocatore clicca sulla finestra, il metodo mouseClicked(MouseEvent e) converte le coordinate del clic in un Point2DI e verifica se uno degli oggetti cliccabili è stato premuto, eseguendo l'azione corrispondente. In questo modo, quando il giocatore clicca sul bottone, il gioco intercetta l'evento, riconosce che è stato premuto il pulsante "Try Again" e riavvia il livello tramite il controller.



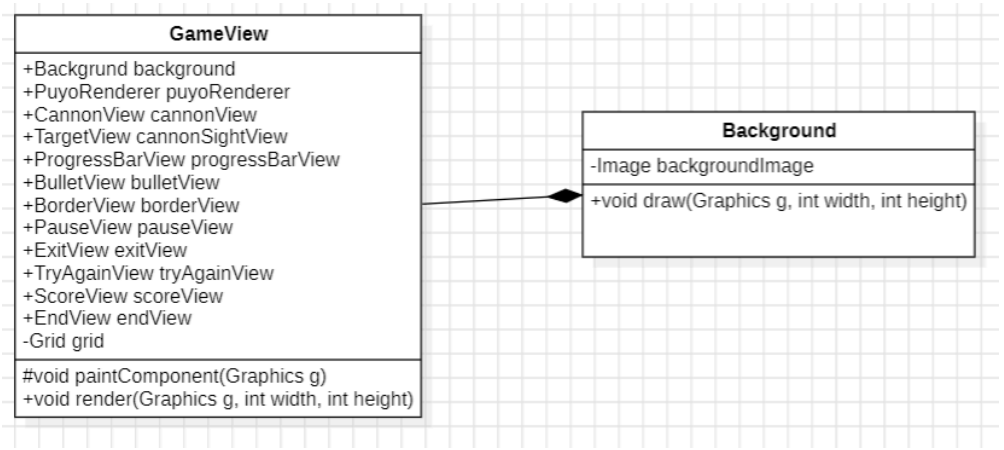
- Gestione delle logiche di esplosione e post-esplosione:
    - **Problema**: Assicurarsi che il proiettile sia animato correttamente. Garantire la corretta funzionalità del proiettile permettendo di distruggere potenziali gruppi di Puyo dello stesso colore, anche assecondando le logiche di freeze e unfreeze.
    - **Soluzione**: Il problema dell'animazione e della gestione della distruzione dei Puyo è stato risolto con l’implementazione del BulletController, assicurando che, una volta attivato, il proiettile segua la sua traiettoria fino a raggiungere un Puyo. Quando il proiettile impatta un Puyo, il metodo “explodePuyos” utilizza un algoritmo di Breadth-First Search per individuare tutti i Puyo connessi dello stesso colore, tenendo conto delle possibili logiche di freeze. In particolare, un Puyo congelato non può

essere eliminato, rendendo la distruzione di un gruppo solo parziale, nel caso se ne incontrasse uno. Se il Puyo è congelato, il ProgressBarController, quando carico, gestisce lo scongelamento. I Puyo non congelati vengono invece marcati per la distruzione con un deathClock (campo della classe Puyo) che scala in base alla distanza dall'impatto, creando un effetto di esplosione progressiva. Il PuyoExplosionController è il responsabile dell'aggiornamento del deathClock a ogni tick, decrementando il timer, impostandolo o eliminando il Puyo dalla griglia qualora il tempo della sua esplosione sia finito. L'animazione del proiettile è gestita dal BulletView, che ne aggiorna graficamente la posizione fino all'impatto, e che lo rende visibile solo se attivo. L'utilizzo della classe Point2D al posto di Point2DI avviene solo nel modello del proiettile, dove si è preferito mantenere l'astrazione delle coordinate per motivi strutturali.



Federica Guiducci:

- Grafica di sfondo:
  - **Problema:** Gestire il rendering dello sfondo del gioco in modo che sia correttamente scalato alle dimensioni della finestra e caricato dinamicamente in base alle risorse disponibili. Lo sfondo deve essere facilmente sostituibile con altre immagini senza modificare il codice.
  - **Soluzione:** La grafica di sfondo è gestita dalla classe Background, che implementa l'interfaccia BackgroundInterface. Questa classe si occupa di caricare un'immagine dalla directory resources/images e di disegnarla all'interno della finestra di gioco.
    L'immagine viene caricata nel costruttore attraverso il metodo getResource, che recupera il file specificato all'interno del classpath del progetto. L'oggetto ImageIcon viene quindi utilizzato per convertire l'URL dell'immagine in un'istanza della classe Image, che viene memorizzata nella variabile backgroundImage.
    Il metodo draw(Graphics g, int width, int height) è responsabile del rendering dell'immagine di sfondo. Se l'immagine è stata caricata correttamente, viene disegnata utilizzando g.drawImage, scalata per adattarsi alle dimensioni specificate della finestra (width e height).
    Grazie a questa implementazione, la classe Background permette una gestione flessibile delle immagini di sfondo, rendendo possibile la sostituzione dell'immagine semplicemente passando un nome di file diverso al costruttore, senza dover modificare la logica di rendering.



- Avvio del gioco:
  - **Problema:** Gestire correttamente l'inizializzazione del gioco, assicurando che tutti i componenti essenziali (modelli, controller e vista) siano collegati e pronti per l'esecuzione. Il gioco deve avviarsi in modo coerente, garantendo che il loop di aggiornamento funzioni correttamente e che l'interfaccia utente venga popolata con gli elementi di gioco. Inoltre, è necessario gestire la sincronizzazione tra la logica di gioco e il rendering grafico, mantenendo un frame rate stabile e impedendo aggiornamenti non necessari quando il gioco è in pausa o terminato.
  - **Soluzione:** L'avvio del gioco è orchestrato dalla classe GameManager, che funge da punto centrale per l'inizializzazione dei componenti principali.

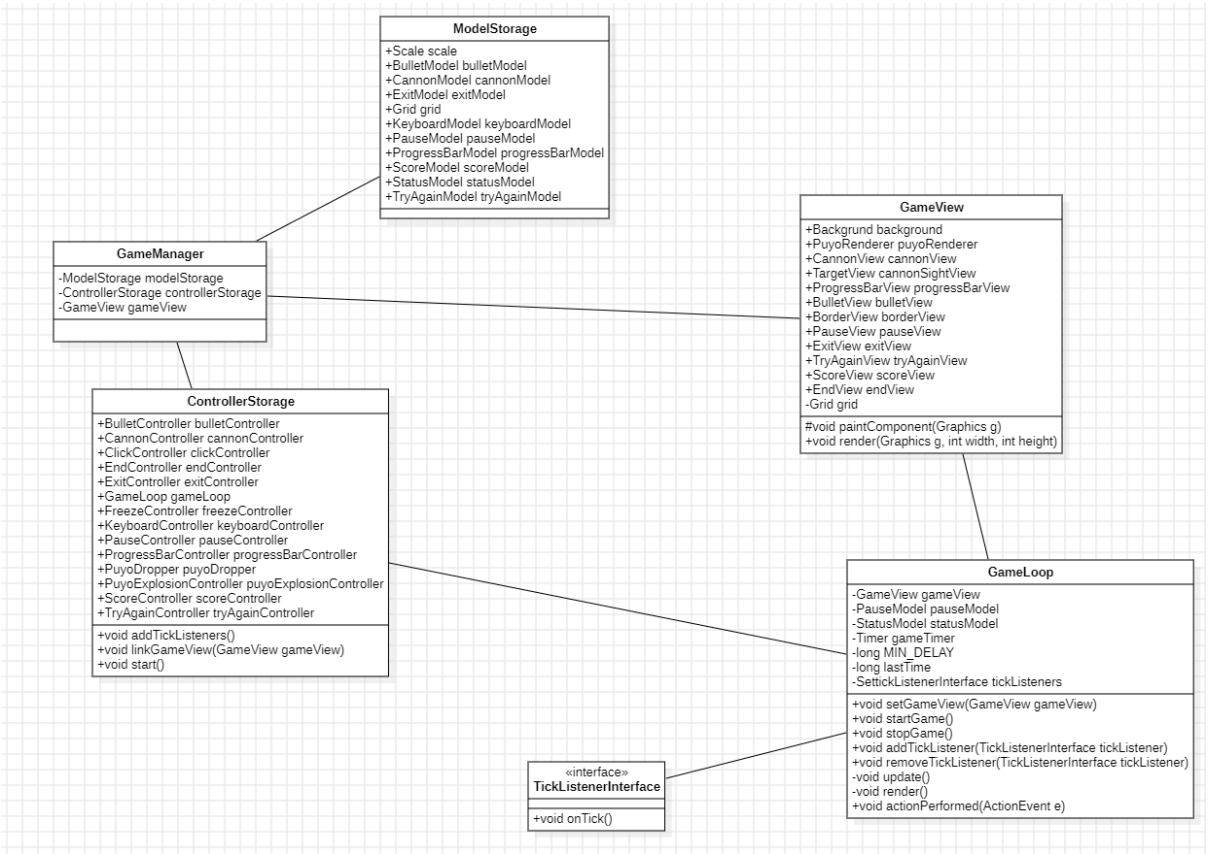
Questa classe estende JInternalFrame per integrarsi con l'interfaccia grafica e rimuove la barra del titolo per una visualizzazione più pulita.
Alla creazione di un'istanza di GameManager, vengono inizializzati i seguenti elementi:

- ModelStorage: responsabile della gestione dello stato di gioco, incluso il punteggio, la posizione degli elementi e la progressione dei livelli.
- ControllerStorage: gestisce le interazioni dell'utente e le logiche di gioco, collegandosi al ModelStorage e al GameListener per rispondere agli eventi.
- GameView: rappresenta graficamente lo stato del gioco, aggiornando l'interfaccia utente in base alle modifiche nei modelli.

Una volta creati questi componenti, controllerStorage viene collegato alla GameView per garantire la sincronizzazione tra la logica di gioco e la grafica. Infine, il metodo start() viene chiamato per avviare il loop di gioco.
La classe GameLoop è responsabile della gestione del ciclo principale del gioco. Utilizza un javax.swing.Timer per eseguire aggiornamenti regolari a un frame rate fisso di 30 FPS. Il metodo actionPerformed si occupa di:

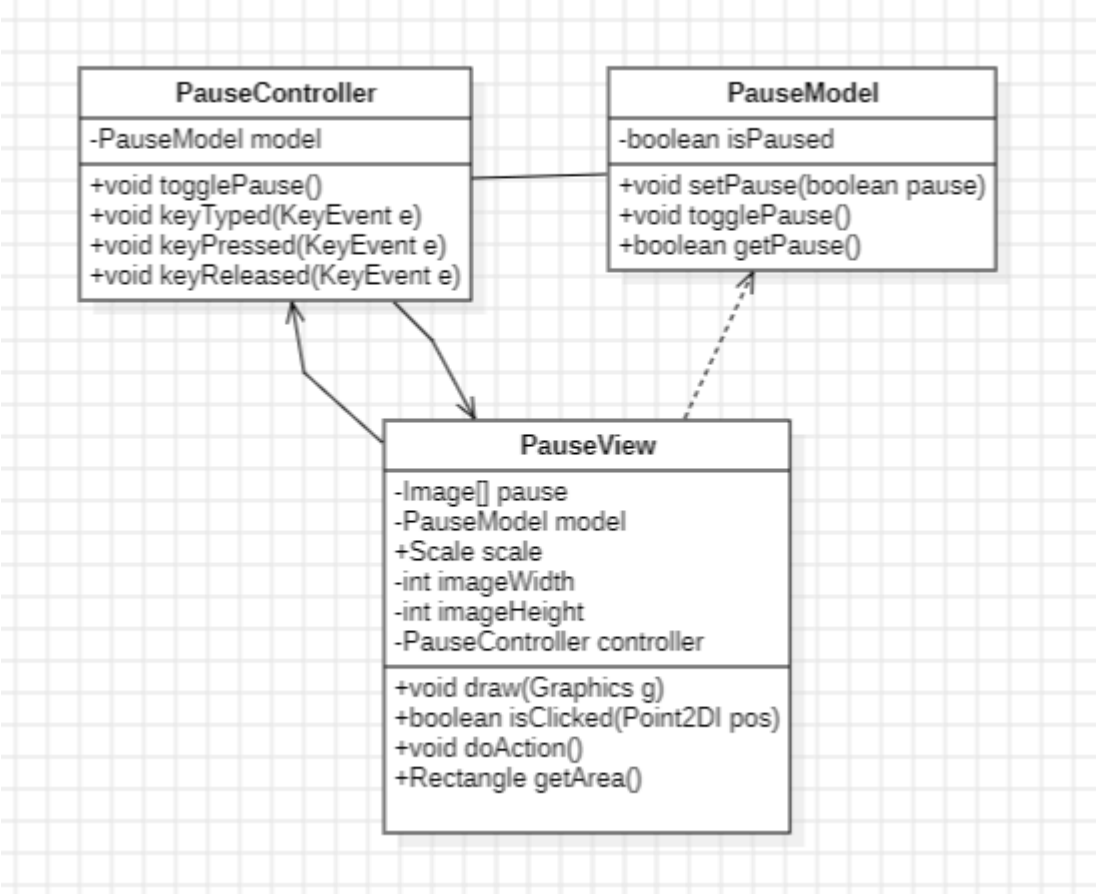
1. Aggiornare lo stato di gioco chiamando update(), che notifica tutti i TickListenerInterface registrati, a meno che il gioco non sia in pausa o terminato.
2. Renderizzare la grafica attraverso il metodo render(), che chiama repaint() su GameView.

Il loop può essere avviato con startGame(), che inizializza il timer e imposta il tempo iniziale, e fermato con stopGame(), interrompendo il timer. Questo approccio garantisce una gestione efficiente del ciclo di aggiornamento, minimizzando il consumo di risorse quando il gioco è in pausa o terminato.
Grazie a questa implementazione modulare, il processo di avvio del gioco è ben strutturato, consentendo una facile estensione delle funzionalità senza compromettere la stabilità del sistema.



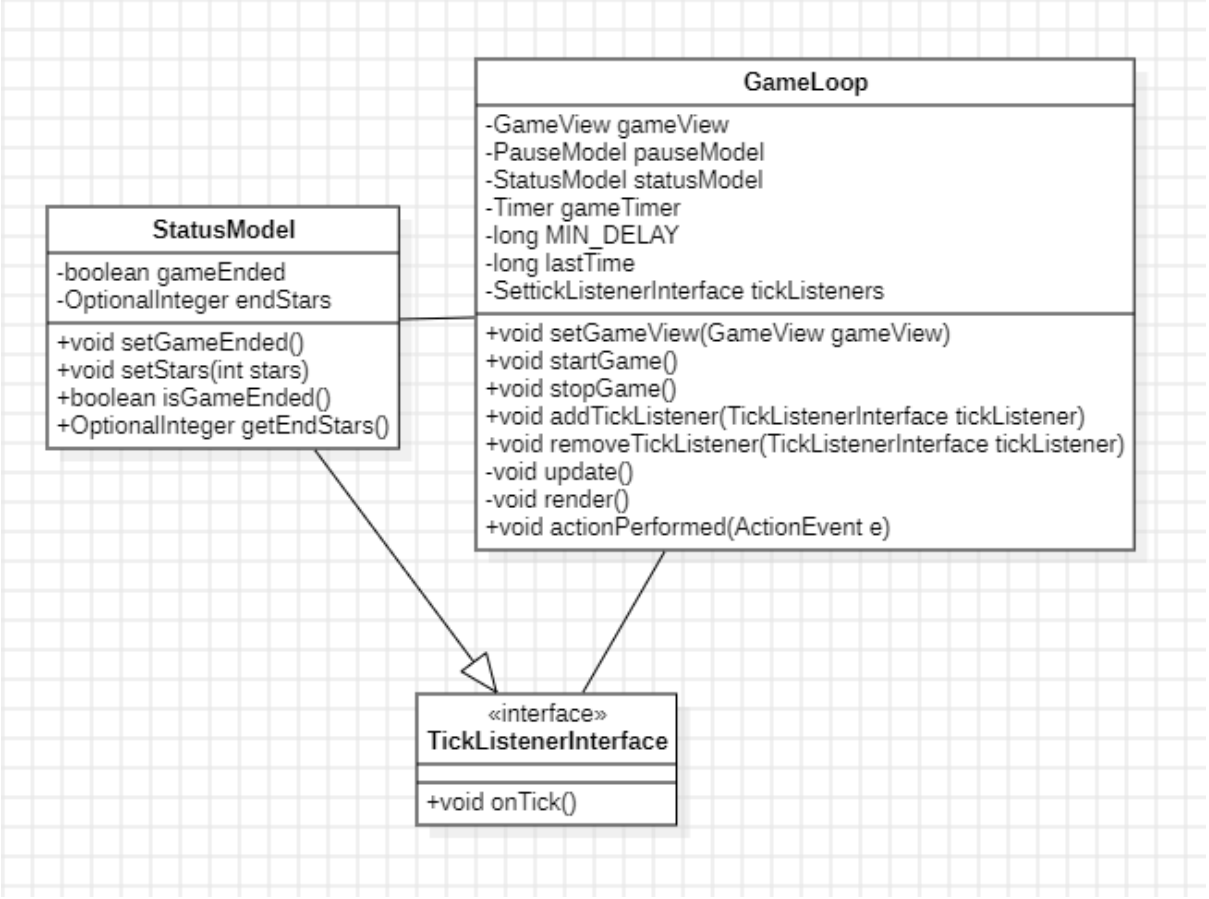
- Gestione della pausa:
    - **Problema:** Durante il gioco, l'utente deve poter mettere in pausa e riprendere la partita in modo intuitivo. Questo deve funzionare sia attraverso l'input da tastiera (es. pressione del tasto "P") sia tramite un'interfaccia grafica con un pulsante di pausa. Inoltre, il gioco deve visivamente segnalare lo stato di pausa, ad esempio oscurando lo schermo.
    - **Soluzione:** La gestione della pausa è implementata attraverso tre componenti principali:
        - PauseModel (Modello): responsabile della gestione dello stato di pausa.
          Contiene un booleano (isPaused) che indica se il gioco è in pausa.
          Fornisce metodi per impostare (setPause()), alternare (togglePause()) e ottenere (getPause()) lo stato di pausa.
        - PauseController (Controller): ascolta gli eventi da tastiera attraverso l'implementazione di KeyListener.
          Quando l'utente preme il tasto "P", richiama togglePause() su PauseModel, cambiando lo stato del gioco.
        - PauseView (Vista): mostra un pulsante per mettere in pausa/riprendere il gioco.
          Carica e visualizza le immagini del pulsante di pausa e ripresa.
          Applica un'overlay trasparente sullo schermo quando il gioco è in pausa.
          Rileva i click dell'utente sul pulsante e richiama il metodo togglePause() del controller.

Questa architettura separa chiaramente la logica (modello), la gestione degli input (controller) e l'interfaccia grafica (vista), garantendo modularità e facilità di estensione.



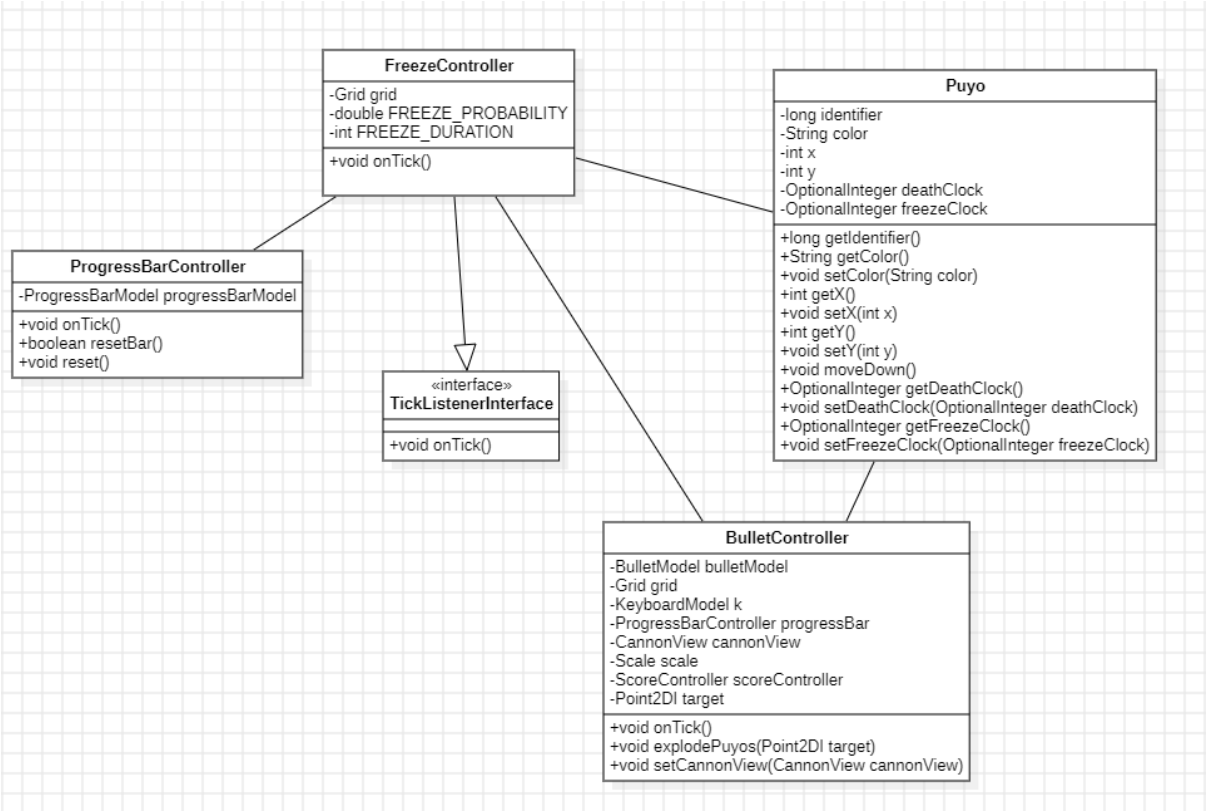
- Gestione dello stato della partita:
    - **Problema:** La partita deve gestire il proprio stato, determinando quando il gioco è terminato e registrando il punteggio finale (espresso in stelle). È essenziale che il gioco possa: segnalare quando la partita è finita, memorizzare il numero di stelle guadagnate alla fine del gioco e consentire ad altri componenti del gioco (es. interfaccia utente, logica di gioco) di recuperare queste informazioni per aggiornare la schermata di fine partita o salvare il progresso del giocatore.
      Il sistema deve inoltre integrarsi con il ciclo di gioco, assicurando che lo stato della partita sia aggiornato a ogni "tick" (unità di aggiornamento del gioco).
    - **Soluzione:** Per gestire lo stato della partita, è stato introdotto il modello StatusModel, il quale mantiene informazioni sulla fine del gioco e sul punteggio finale, rappresentato dal numero di stelle guadagnate. Il modello utilizza una variabile booleana gameEnded per indicare se la partita è terminata e un Optional<Integer> endStars per memorizzare il numero di stelle assegnate al termine. All'inizio della partita, il gioco è impostato come non terminato e il numero di stelle è assente. Quando la partita giunge alla fine, il sistema invoca setGameEnded() per segnare la conclusione e, se necessario, assegna il punteggio con setStars(int stars). Il metodo isGameEnded() consente di verificare se il gioco è finito, mentre

getEndStars() restituisce il numero di stelle eventualmente guadagnate. Per garantire che lo stato venga aggiornato in modo coerente durante l'esecuzione del gioco, è stata definita l'interfaccia TickListenerInterface, che prevede il metodo onTick(), il quale viene eseguito a ogni aggiornamento del ciclo di gioco. Questo approccio permette di monitorare costantemente le condizioni della partita e di aggiornare lo stato in tempo reale, assicurando una chiara separazione tra logica di gioco e interfaccia grafica. L'uso di Optional<Integer> garantisce inoltre una gestione più sicura dell'assenza di un valore prima della conclusione della partita, evitando il rischio di valori nulli o incoerenti. In questo modo, il sistema fornisce un meccanismo flessibile e modulare per determinare quando la partita termina e quale punteggio è stato ottenuto, rendendo semplice l'integrazione con altri componenti come la schermata di fine gioco, i salvataggi e le classifiche.



- Gestione del freeze:
    - **Problema:** Nel gioco, è necessario introdurre una meccanica di congelamento (freeze) che possa bloccare temporaneamente i Puyo sulla griglia, impedendone il movimento e l'esplosione fino al termine dell'effetto. Questa funzione deve essere gestita in modo probabilistico e aggiornata a ogni tick del gioco. Inoltre, è necessario che il giocatore possa interagire con i Puyo congelati, ad esempio sbloccandoli attraverso l'uso di un'azione specifica.
    - **Soluzione:** Per implementare questa meccanica, è stato creato il FreezeController, che implementa l'interfaccia TickListenerInterface,

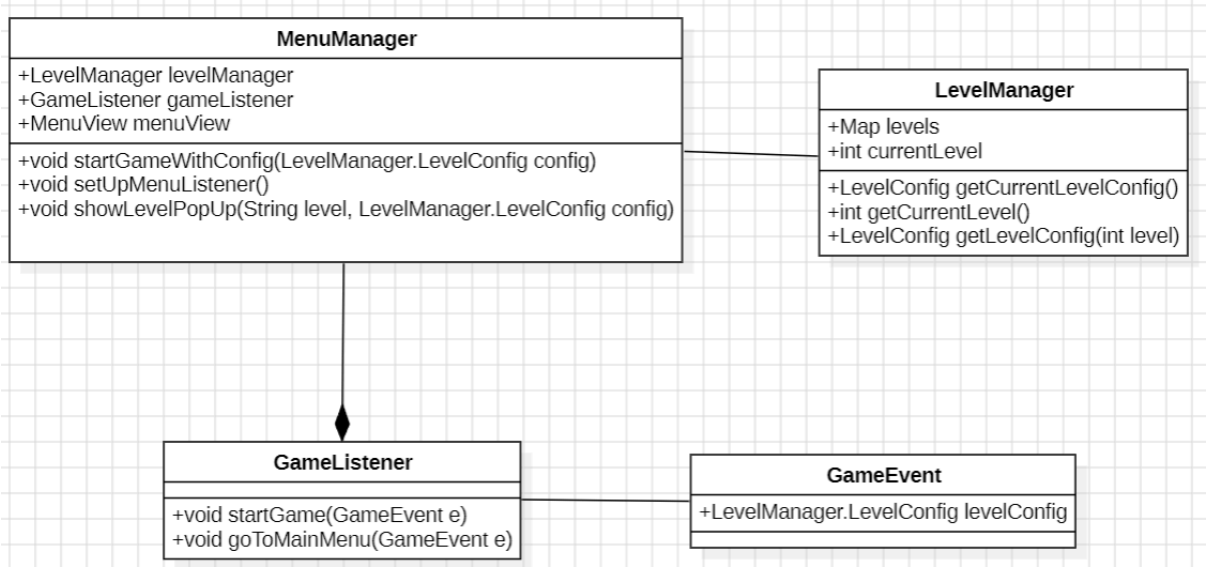
permettendo così di aggiornare lo stato dei Puyo congelati a ogni tick di gioco. Il controller scorre l'intera griglia e, per ogni Puyo, verifica se è già congelato: in tal caso, decrementa il suo timer di congelamento fino a quando l'effetto scompare. Se invece il Puyo non è congelato, esiste una piccola probabilità definita (FREEZE_PROBABILITY) che venga congelato per una durata fissa (FREEZE_DURATION). Parallelamente, il BulletController gestisce l'interazione tra i proiettili e i Puyo. Quando un proiettile colpisce un Puyo, si attiva l'algoritmo di esplosione basato su una ricerca in ampiezza (BFS), che individua e marca per la distruzione i Puyo adiacenti dello stesso colore. Tuttavia, se il Puyo colpito è congelato, l'esplosione non avviene immediatamente: al contrario, si verifica se la barra di avanzamento (ProgressBarController) è carica. Se lo è, il Puyo viene scongelato e la barra si resetta, aggiungendo un livello di strategia in cui il giocatore deve gestire quando e come scongelare i Puyo. Questo sistema assicura che la meccanica di congelamento si integri in modo naturale con il flusso del gioco, senza risultare eccessivamente punitiva o imprevedibile.



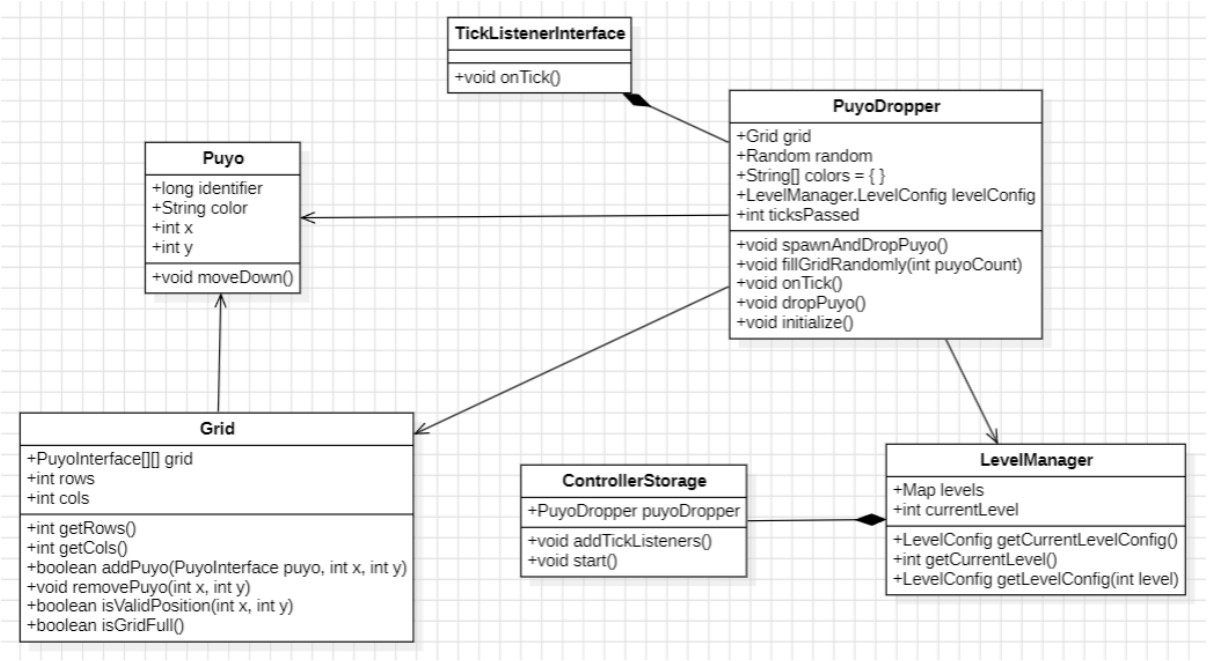
Chiara De Nardi:

- Gestione caricamento livelli:
    - **Problema**: Gestire il caricamento di un livello scelto dal giocatore.
    - **Soluzione**: Per risolvere il problema si parte dal MenuManager dove l'utente seleziona il livello e clicca su "Start". Il MenuManager recupera la configurazione del livello tramite LevelManager. Viene mostrato un popup di conferma e, al click su "OK", richiama startGameWithConfig (del MenuManager) per inviare l'evento al

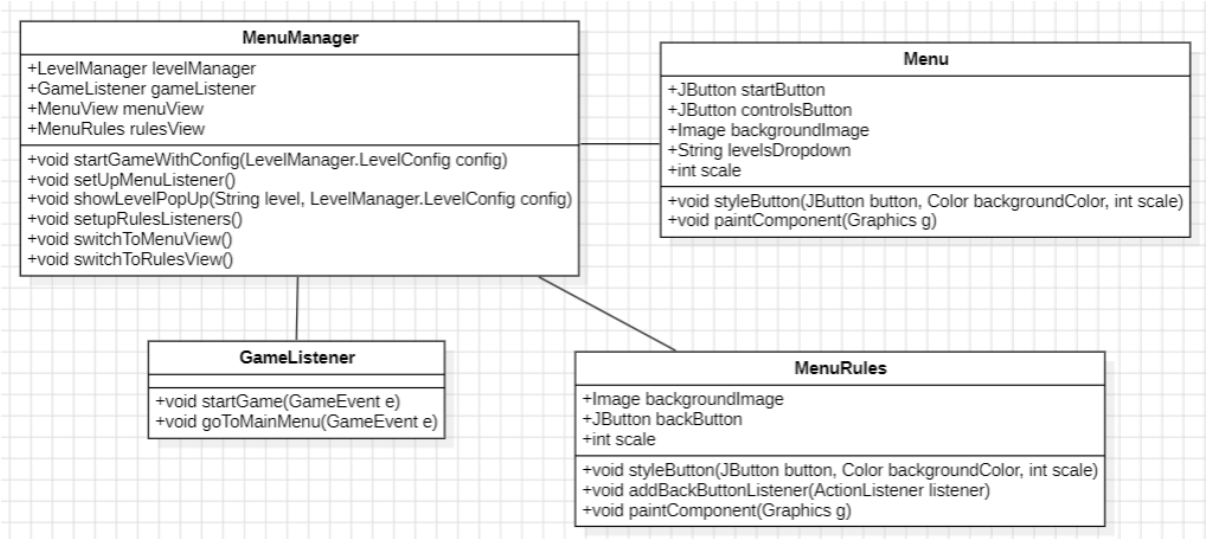
GameListener. Il GameListener avvia il gioco tramite startGame() con la configurazione del livello selezionato tramite GameEvent.



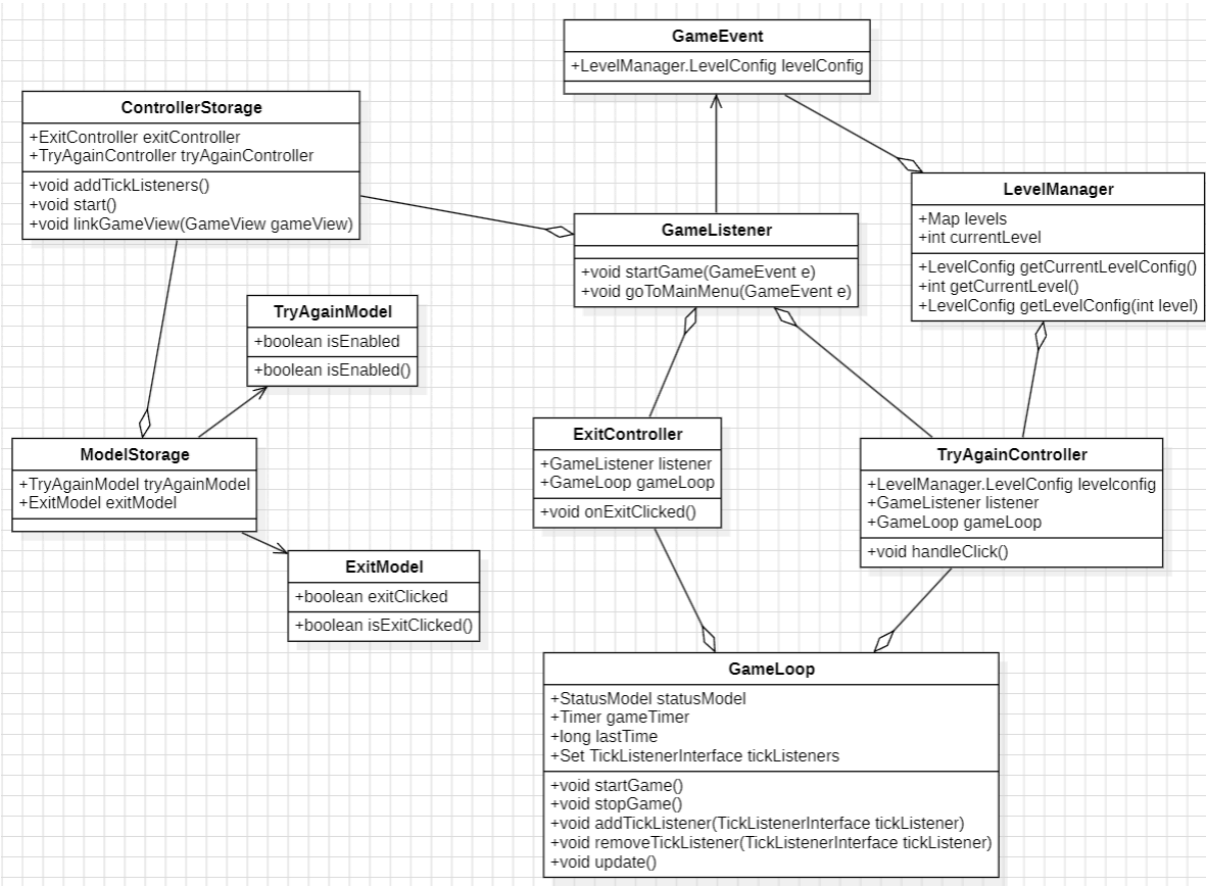
- Gestione caduta puyo nella griglia:
    - **Problema**: Gestire la caduta dei puyo nella griglia in maniera fluida.
    - **Soluzione:** La gestione della caduta dei Puyo nel gioco coinvolge principalmente la classe PuyoDropper, che si occupa di generare e far cadere i Puyo nella griglia in base alle configurazioni dei livelli, della classe Puyo. Ad ogni tick del gioco, PuyoDropper verifica se è il momento di far cadere nuovi Puyo (in base al ritardo definito dal livello) e sposta i Puyo già presenti verso il basso. La griglia è gestita dalla classe Grid, che assicura che ogni Puyo venga posizionato in una posizione valida. Infine, ControllerStorage centralizza il flusso di gioco, aggiungendo il PuyoDropper al ciclo di aggiornamento e chiamando il metodo onTick() per gestire la caduta.



- Gestione del menu:
    - **Problema**: Gestire in maniera flessibile tutti i componenti del menù.
    - **Soluzione**: Nel contesto del pattern MVC, il menù del gioco è suddiviso in tre componenti principali: il MenuManager (Controller), le classi Menu e MenuRules (View), e il GameListener (Model). Il MenuManager, come controller, gestisce la logica di navigazione del menù, coordinando le azioni dell'utente, come l'inizio del gioco o la visualizzazione delle regole. Interagisce con il GameListener, che funge da modello, ascoltando gli eventi del gioco e aggiornando lo stato del menù o avviando la partita. La View è composta dalle classi Menu e MenuRules, che si occupano dell'aspetto visivo e dell'interazione con l'utente, mostrando le schermate principali e le regole. La separazione tra controller, vista e modello offre una struttura modulare e flessibile, permettendo modifiche e ampliamenti senza compromettere la funzionalità del sistema.



- Gestione dei pulsanti tryAgain e ExitToMenu:
    - **Problema**: La necessità di separare la logica di controllo (controller) dallo stato dell'applicazione (model) per mantenere il codice organizzato e testabile.
    - **Soluzione**: Il modello è responsabile della gestione dello stato dell'applicazione, senza occuparsi direttamente della logica di gioco o dell'interfaccia utente. Il TryAgainModel Contiene un flag isEnabled che indica se il pulsante "Try Again" è attivo. Permette di abilitare/disabilitare questa funzionalità in base allo stato del gioco. ExitModel, invece contiene un flag exitClicked che tiene traccia se il pulsante "Exit" è stato premuto. Questo stato può essere utilizzato per determinare se il gioco deve tornare al menu principale. Questi modelli sono semplici e servono principalmente per separare la logica di gestione dell'UI dalla logica del gioco. Vengono poi chiamati dal ModelStorage che viene usato sia nella GameView che nel GameManager.
      Il controller (Controller) gestisce gli eventi generati dall'utente, interagendo con il modello e notificando eventuali cambiamenti. In questo caso, i controller si occupano delle azioni associate ai pulsanti "Try Again" e "Exit". Il primo memorizza la configurazione del livello (LevelConfig) tramite il GameEvent, il GameListener e il GameLoop. Quando viene premuto il pulsante "Try Again", interrompe la partita attuale (gameLoop.stopGame()) e ne avvia una nuova (listener.startGame(event)). Anche exitController interagisce con GameListener e GameLoop.Quando l'utente preme "Exit", il metodo onExitClicked() interrompe il gioco (gameLoop.stopGame()) e comunica al listener di tornare al menu principale (listener.goToMainMenu(event)). In entrambi i casi, i controller non interagiscono direttamente con il rendering grafico, ma si limitano a gestire eventi e comunicare con i listener. Il controller storage poi è quello che collega il ModelStorage col GameListener.


GameEvent
+LevelManager.LevelConfig levelConfig
ControllerStorage
+ExitController exitController
+TryAgainController tryAgainController
+void addTickListeners()
+void start()
+void linkGameView(GameView gameView)
GameListener
+void startGame(GameEvent e)
+void goToMainMenu(GameEvent e)
LevelManager
+Map levels
+int currentLevel
+LevelConfig getCurrentLevelConfig()
+int getCurrentLevel()
+LevelConfig getLevelConfig(int level)
TryAgainModel
+boolean isEnabled
+boolean isEnabled()
ModelStorage
+TryAgainModel tryAgainModel
+ExitModel exitModel
ExitModel
+boolean exitClicked
+boolean isExitClicked()
ExitController
+GameListener listener
+GameLoop gameLoop
+void onExitClicked()
TryAgainController
+LevelManager.LevelConfig levelconfig
+GameListener listener
+GameLoop gameLoop
+void handleClick()
GameLoop
+StatusModel statusModel
+Timer gameTimer
+long lastTime
+Set TickListenerInterface tickListeners
+void startGame()
+void stopGame()
+void addTickListener(TickListenerInterface tickListener)
+void removeTickListener(TickListenerInterface tickListener)
+void update()

CAPITOLO 3

SVILUPPO

3.1 TESTING AUTOMATIZZATO

Per effettuare i test abbiamo usato Junit 5.
I test che abbiamo effettuato sono:

- **GridTest**: controlla la funzionalità della griglia, inclusa la corretta aggiunta e rimozione dei puyo e la pulizia completa della griglia.
- **FreezeControllerTest**: controlla che i meccanismi della feature freeze vengono correttamente applicati ai puyo della griglia.
- **PauseViewTest**: controlla che venga disegnato correttamente il bottone pausa.
- **StatusModelTest**: controlla il corretto comportamento dello stato del gioco.
- **LevelManagerTest**: controlla la configurazione corretta del livello corrente, di un livello casuale e delle caratteristiche di ogni livello.
- **PuyoDropperTest**: testa il corretto funzionamento della caduta dei puyo, dell'inizializzazione dei puyo nella griglia e il meccanismo di caduta.
- **TryAgainModelTest**: controlla il funzionamento e il comportamento del bottone tryagain, in particolare la sua funzionalità di abilitare/disabilitare.
- **ExitModelTest**: controlla il comportamento corretto del bottone exit, se è stato schiacciato o meno
- **TryAgainControllerTest**: controlla che quando viene schiacciato il pulsante tryagain il gioco venga stoppato e ricominciato correttamente
- **ExitControllerTest**: controlla che il bottone, una volta schiacciato, fermi il gioco e torna al menu.
- **BulletModelTest**: assicura che un proiettile nasca inattivo, possa essere sparato, muoversi verso un bersaglio e disattivarsi una volta raggiunta la destinazione.
- **BulletControllerTest**: testa il proiettile su una versione ridotta della griglia con un gruppo di puyo dello stesso colore e un puyo isolato. Verifica che una volta colpito il gruppo, il deathClock sia attivato correttamente solo una volta finita l'animazione del proiettile.
- **ClickControllerTest:** verifica il comportamento del ClickController, assicurandosi che gestisca correttamente gli eventi del mouse e che le interazioni con gli oggetti cliccabili funzionino.
- **PuyoExplosionControllerTest**: simula la riduzione del DeathClock dei Puyo, la rimozione di un Puyo dalla griglia quando il DeathClock è a zero.
- **UtilityTest**: verifica il corretto funzionamento di varie classi utilitarie e componenti di supporto nel sistema. Le classi testate sono Scale, Point2D, Point2DI, Rectangle, ModelStorage e ControllerStorage.
- **ScoreModelTest**: si occupa di verificare che il punteggio venga inizializzato correttamente a 0 e che venga aggiornato in modo corretto durante la partita.
- **ScoreControlletTest**: si occupa di verificare che il punteggio venga aggiornato in modo corretto in base alla formula di calcolo. Verifica il corretto

aggiornamento nel caso in cui deve essere effettua la potenza di 0, 1 o di un altro valore.

- **ProgressBarModelTest**: controlla se la barra di caricamento viene inizializzata e incrementata in modo corretto. Verifica che il valore del caricamento non superi la soglia massima.
- **ProgressBarControllerTest**: verifica l'incremento della barra di caricamento sulla base di onTick e verifica il reset sia nel caso in cui la barra è piena, sia nel caso in cui non è piena. Infine controlla se viene resettata correttamente.
- **KeyboardModelTest**: verifica se l'input da tastiera viene correttamente interpretato simulando una pressione singola e una pressione multipla e simulando quando un tasto è tenuto premuto e poi rilasciato.
- **KeyboardControllerTest**: controlla se viene invocato il metodo corretto sulla base del l'input ricevuto.
- **EndControllerTest**: verifica le condizioni di sconfitta della partita. In particolare controlla, sulla base del punteggio ottenuto o del riempimento della griglia, se viene assegnato il numero corretto di stelle.
- **CannonModelTest**: controlla se l'inizializzazione del cannone, i movimenti laterali e i movimenti dell'angolo del tiro, vengono modificati in modo corretto rispettando i limiti.
- **CannonControllerTest**: testa i movimenti del cannone sulla base dell'input inserito e controlla che il cannone non si muova se non vengono premuti i tasti selezionati per il movimento all'interno del gioco.

3.2 NOTE DI SVILUPPO

Beatrice Di Gregorio:

- **Libreria org.mockito.Mockito**: è una libreria utilizzata nei test unitari con JUnit per creare oggetti mock. Questi oggetti permettono di simulare il comportamento di classi reali in modo controllato, facilitando il testing di componenti isolati. https://github.com/odiugfedefguido/OOP25puyo-blast/blob/73af1cfffa658a2865446e71fb08984fa8a35e48/src/test/java/it/unibo/CannonControllerTest.java
- **Optional**: nella classe EndView viene utilizzato Optional<Integer> per rappresentare il numero di stelle ottenute al termine del gioco. L'uso di Optional permette di gestire in modo più sicuro la presenza o assenza del valore evitando NullPointerException. https://github.com/odiugfedefguido/OOP25puyo-blast/blob/73af1cfffa658a2865446e71fb08984fa8a35e48/src/main/java/it/unibo/view/EndView.java
- **HashSet**: nella classe KeyboardModel viene utilizzato java.util.HashSet per gestire in modo efficiente un insieme di tasti premuti. https://github.com/odiugfedefguido/OOP25puyo-blast/blob/73af1cfffa658a2865446e71fb08984fa8a35e48/src/main/java/it/unibo/model/KeyboardModel.java

Aisja Baglioni:

- **Libreria org.mockito.Mockito**: Mockito è una libreria utilizzata nei test unitari con JUnit per creare oggetti mock. Questi oggetti fantoccio permettono di simulare il comportamento di oggetti reali in modo controllato e prevedibile. Qui un esempio sullo UtilityTest.

Permalink:
https://github.com/odiugfedefguido/OOP25puyo-blast/blob/cc93fc1d3a371ff0dd6159dd88652ac95281f355/src/test/java/it/unibo/UtilityTest.java#L99-L107

- **Algoritmo Breadth-First Search:** Nel contesto del BulletController, una BFS viene utilizzata per trovare tutti i Puyos collegati dello stesso colore, in modo da poterne formare un gruppo per un'esplosione.
  Permalink:
  https://github.com/odiugfedefguido/OOP25puyo-blast/blob/cc93fc1d3a371ff0dd6159dd88652ac95281f355/src/main/java/it/unibo/controller/BulletController.java#L120-L184

- **Hash Code:** Insieme al metodo equals, il metodo hashCode di Point2DI viene utilizzato per ottimizzare l'operazione di ricerca durante l'esplorazione della BFS. L'hashCode permette di mappare in modo efficiente gli oggetti in una struttura dati, come una hashset o una hashmap, riducendo il tempo di ricerca di un oggetto nella collezione. Viene usato un numero primo nella sua implementazione per migliorare la distribuzione degli hash e ridurre la probabilità di collisioni.
  Permalink:
  https://github.com/odiugfedefguido/OOP25puyo-blast/blob/cc93fc1d3a371ff0dd6159dd88652ac95281f355/src/main/java/it/unibo/model/Point2DI.java#L101-L106

- **Uso di operatori bitwise:** Nel Puyo Renderer, si mappano gli sprites sulla base dei vicini dello stesso colore. Il mapping è effettuato grazie all'operatore bitwise OR (|=), che imposta a 1 il bit più significativo nella variabile mask se l'espressione a destra è vera. L''operatore di shift a sinistra (<<=), sposta tutti i bit di mask di una posizione a sinistra.
  Permalink:
  https://github.com/odiugfedefguido/OOP25puyo-blast/blob/cc93fc1d3a371ff0dd6159dd88652ac95281f355/src/main/java/it/unibo/view/PuyoRenderer.java#L141-L163

Federica Guiducci:

- **Libreria org.mockito.Mockito**: Mockito è una libreria utilizzata nei test unitari con JUnit per creare oggetti mock. Viene usata in FreezeControllerTest, PauseViewTest.
  Permalink:
  https://github.com/odiugfedefguido/OOP25puyo-blast/blob/73af1cfffa658a2865446e71fb08984fa8a35e48/src/test/java/it/unibo/FreezeControllerTest.java#L3

  https://github.com/odiugfedefguido/OOP25puyo-blast/blob/73af1cfffa658a2865446e71fb08984fa8a35e48/src/test/java/it/unibo/PauseViewTest.java#L6

- **Observe Pattern:** La classe GameLoop lo utilizza per notificare gli oggetti registrati quando avviene un aggiornamento di gioco.
  Permalink:
  https://github.com/odiugfedefguido/OOP25puyo-blast/blob/73af1cfffa658a2865446e71fb08984fa8a35e48/src/main/java/it/unibo/controller/GameLoop.java#L111

  La classe GameManager lo utilizza per notificare gli eventi di gioco.
  Permalink:

https://github.com/odiugfedefguido/OOP25puyo-blast/blob/73af1cfffa658a2865446e71fb08984fa8a35e48/src/main/java/it/unibo/controller/GameManager.java#L50

La classe PauseController lo utilizza per ascoltare gli eventi di gioco (tastiera)
Permalink:
https://github.com/odiugfedefguido/OOP25puyo-blast/blob/73af1cfffa658a2865446e71fb08984fa8a35e48/src/main/java/it/unibo/controller/PauseController.java#L57-L63

- **Facade Pattern:** game manager funge da facciata per inizializzare e collegare model, view e controller.
  Permalink:
  https://github.com/odiugfedefguido/OOP25puyo-blast/blob/73af1cfffa658a2865446e71fb08984fa8a35e48/src/main/java/it/unibo/controller/GameManager.java#L17-L55

- **Optional:** Nella classe StatusModel
  Permalink:
  https://github.com/odiugfedefguido/OOP25puyo-blast/blob/73af1cfffa658a2865446e71fb08984fa8a35e48/src/main/java/it/unibo/model/StatusModel.java#L45

Chiara De Nardi:

- **Optional**: Nella classe Puyo vengono usati gli Optional per le variabili DeathClock e FreezeClock.
  Permalink:
  https://github.com/odiugfedefguido/OOP25puyo-blast/blob/e6afb9ba375162ca2466c5d24e62ff1da0f7bd9d/src/main/java/it/unibo/model/Puyo.java#L40-L49

- **ThreadLocalRandom**: Nella classe Puyo viene usato al posto di Math.random() per codice più compatto e per prestazioni migliori.
  Permalink:
  https://github.com/odiugfedefguido/OOP25puyo-blast/blob/e6afb9ba375162ca2466c5d24e62ff1da0f7bd9d/src/main/java/it/unibo/model/Puyo.java#L40-L49

- **Libreria java.awt.geom.AffineTransform**: Viene usata nella classe MenuRules.
  Permalink:
  https://github.com/odiugfedefguido/OOP25puyo-blast/blob/e6afb9ba375162ca2466c5d24e62ff1da0f7bd9d/src/main/java/it/unibo/view/MenuRules.java#L156-L168

- **Libreria org.mockito.Mockito**: Mockito è una libreria utilizzata nei test unitari con JUnit per creare oggetti mock. Viene utilizzata nella classe TryAgainControllerTest.
  Permalink:
  https://github.com/odiugfedefguido/OOP25puyo-blast/blob/a493dea2ca5ad40b0167f757133da51cd1ee09a3/src/test/java/it/unibo/TryAgainControllerTest.java#L33-L36

- **Lambda**: Nella classe MenuManager vengono usate per definire gli actionlistener dei pulsanti, rendendo il codice più conciso ed eliminando la necessità di dichiarare classi anonime separate per ciascun ascoltatore.

Permalink:
https://github.com/odiugfedefguido/OOP25puyo-blast/blob/e6afb9ba375162ca2466c5d24e62ff1da0f7bd9d/src/main/java/it/unibo/controller/MenuManager.java#L65-L76

- **Interfaccia funzionale**: Nelle classi MenuManager e MenuRules viene usata per definire gli ActionListener per i pulsanti. In questo caso l'interfaccia funzionale è "ActionListener", che è un'interfaccia predefinita in Java ed è usata per definire il comportamento da eseguire quando un pulsante viene premuto.
  Permalink:
  https://github.com/odiugfedefguido/OOP25puyo-blast/blob/e6afb9ba375162ca2466c5d24e62ff1da0f7bd9d/src/main/java/it/unibo/view/MenuRules.java#L124-L127

  https://github.com/odiugfedefguido/OOP25puyo-blast/blob/e6afb9ba375162ca2466c5d24e62ff1da0f7bd9d/src/main/java/it/unibo/controller/MenuManager.java#L187-L190

CAPITOLO 4

COMMENTI FINALI

4.1 AUTOVALUTAZIONE E LAVORI FUTURI

Beatrice Di Gregorio:
Lo sviluppo di questo progetto è stata una sfida che mi ha messo a dura prova, ma che complessivamente ritengo un'esperienza stimolante e di accrescimento. Pur non essendo la prima esperienza di lavoro di gruppo, ho potuto imparare da questo progetto a coordinarmi meglio con gli altri e a lavorare sia da sola che in team. Inoltre, grazie a questa esperienza, ho avuto la possibilità di approfondire il linguaggio Java e le sue funzionalità e di acquisire una maggiore familiarità con Git. Inizialmente ho avuto difficoltà nel comprendere la struttura del progetto e ho cercato spesso il confronto con il team che mi ha supportato e mi ha indirizzato verso una prima implementazione dei miei obiettivi. Anche se non sono particolarmente appassionata di videogiochi, ho trovato questo progetto davvero coinvolgente e mi ritengo soddisfatta del lavoro che abbiamo portato a termine.

Aisja Baglioni:

Mi sono iscritta a questo corso di laurea con l'intenzione di sviluppare videogiochi, ma purtroppo ho avuto poche opportunità di lavorare in team per realizzarne uno. Non conoscevo molto Java, ma è stato interessante esplorare le sue librerie, affrontare problemi di problem solving e risolvere dipendenze non banali, a volte ristrutturando intere sezioni di codice. La mia conclusione è che il linguaggio mi piace e la programmazione orientata agli oggetti offre molte opportunità nello sviluppo di applicazioni. Puyo Pop è stato una parte importante della mia infanzia, e poter animare i Puyo mi ha dato un grande senso di soddisfazione. È stato inoltre stimolante alternare la coordinazione del lavoro secondo gli obiettivi fissati, di volta in volta, e tutte abbiamo cercato di rendere agevole lo sviluppo ponderando e accogliendo i suggerimenti e le proposte che venivano riportate.

Federica Guiducci:
Lo sviluppo di questo progetto è stato impegnativo, dato il carico di lavoro e soprattutto il doversi coordinare in gruppo; nonostante ciò posso ritenermi soddisfatta in quanto anche nonostante le difficoltà, ho appreso nuovi concetti e ampliato il mio bagaglio formativo.

Chiara De Nardi:
Questa esperienza è stata stimolante e gratificante, permettendomi di lavorare in gruppo e di affinare sia le mie competenze tecniche che le mie capacità di collaborazione. Il progetto, seppur impegnativo, si è rivelato altamente formativo, spingendomi a confrontarmi con nuove sfide e a trovare soluzioni ai problemi incontrati lungo il percorso. Ogni ostacolo

superato ha contribuito alla mia crescita, rafforzando la mia determinazione e il mio spirito di squadra.

APPENDICE A

GUIDA UTENTE

Quando viene fatto partire l'applicativo ci si ritroverà in una scena con 3 bottoni:

- Comandi: schermata dove ci sono le istruzioni del gioco.
- Scelta livello: menù a tendina dove selezionare il livello su cui giocare.
- Inizia il gioco: fa partire il livello selezionato.

I comandi di gioco sono i seguenti:

- P per la pausa.
- Barra spaziatrice per sparare.
- ← (Left Arrow): per muovere il cannone a sinistra.
- → (Right Arrow): per muovere il cannone a destra.