

BRICK-BRK

“Brick Breaker”

Bighini Luca, Agostinelli Francesco, Tellarini Pietro

17 giugno 2023

Indice

1	Analisi	2
1.1	Requisiti	2
1.2	Analisi e modello del dominio	3
2	Design	6
2.1	Architettura	6
2.2	Design dettagliato	8
3	Sviluppo	20
3.1	Testing automatizzato	20
3.2	Metodologia di lavoro	20
3.3	Note di sviluppo	23
4	Commenti finali	26
4.1	Autovalutazione e lavori futuri	26
4.2	Difficoltà incontrate e commenti per i docenti	28
A	Guida utente	29

Capitolo 1

Analisi

Il nostro obiettivo è sviluppare un coinvolgente gioco platform 2D a livelli. Nella modalità di gioco, il compito del giocatore è ottenere il punteggio più alto possibile distruggendo mattoncini. Il giocatore avrà il controllo di una barra posizionata nella parte inferiore dello schermo, sulla quale rimbalzerà una pallina. Utilizzando la barra, il giocatore dovrà cambiare la direzione della pallina per colpire e distruggere i mattoncini presenti sullo schermo. Il livello sarà considerato completato quando tutti i mattoncini saranno stati distrutti e il giocatore avrà almeno una vita rimanente. Per rendere il gioco ancora più interessante, quando un mattoncino viene distrutto, potrebbe rilasciare un power-up. I power-up possono essere vantaggiosi o svantaggiosi e offrono al giocatore abilità speciali temporanee.

1.1 Requisiti

Requisiti funzionali

- Interfaccia per la scelta del livello di gioco, dal menù è possibile e selezionare le modalità di gioco. Selezionando la modalità a livelli sarà possibile scegliere infine il livello che da giocare.
- Il gioco si apre con la schermata iniziale che permette di selezionare il giocatore salvato precedentemente oppure crearne e rimuoverne di nuovi.
- Successivamente verrà mostrata una interfaccia per la scelta della modalità di gioco e del livello di difficoltà.
- Creazione del pattern dei brick con l'assegnazione di power-up distribuendoli in modo equilibrato in base al livello e alla relativa difficoltà.

- Il movimento della barra di gioco avverrà tramite input da tastiera.
- Corretta collisione della pallina con gli oggetti in gioco e il relativo angolo di rimbalzo.
- I power-up a cascata verso la barra e applicazione della rispettiva modifica che essi apportano.
- Scrittura del punteggio realizzato su file e visualizzazione dei loro punteggi con quelli degli altri giocatori.
- Mettere in pausa temporaneamente il gioco e riprenderlo successivamente.
- Corretta implementazione del pattern di progettazione MVC.
- Estendibilità del codice per futura aggiunta di mappe e power-up in modo da poter aggiungere eventuali modifiche o novità al gioco.

Requisiti non funzionali

- Interfaccia di gioco semplice tale da rendere immediato l'avvio di una partita.
- Prestazioni in grado di offrire una buona esperienza di gioco.
- Prestazioni in grado di offrire una buona esperienza di gioco. Disposizione di power-up in modo casuale all'interno dei bricks.
- Creazione di livelli randomici in base alla difficoltà nella relativa modalità di gioco.

1.2 Analisi e modello del dominio

Il tipo di partita che il giocatore andrà a giocare dipende dalla modalità che verrà selezionata che può essere infinita o di tipo carriera, questo porterà a diverse modalità nella creazione del mondo e il relativo posizionamento degli oggetti in gioco.

Nella modalità “carriera” il giocatore dovrà selezionare uno dei livelli preconfigurati dal menù che verranno caricati da files. In questa modalità il giocatore potrà giocare solamente in modo scalare ai livelli, vincendo un livello sarà infatti possibile sbloccare il livello successivo.

Nella modalità “infinita” al giocatore verranno presentati invece modelli generati randomicamente in base ad un fattore di difficoltà e il gioco proseguirà fintanto che il giocatore non avrà perso tutte le vite a disposizione.

Durante il gioco l’utente avrà la possibilità di muovere orizzontalmente una barra posta nella parte inferiore dello schermo per farle rimbalzare al di sopra una o più palline in movimento costante nella finestra di gioco. Una pallina che colliderà contro un ostacolo nel gioco dovrà decrementarne la vita e cambiare la sua direzione di movimento. Se una pallina collide con il limite inferiore della finestra di gioco verrà rimossa e nel caso di esaurimento delle palline sarà decrementata una vita.

Inoltre, un ostacolo potrà resistere ad un numero di urti prima che venga distrutto. Alla sua distruzione verrà incrementato il punteggio e potrà generare un nuovo oggetto power-up che si muoverà verticalmente verso il margine inferiore. Se durante il tragitto incontreranno la barra verranno attivati potenziamenti o penalità al giocatore.

Dovremo ideare un’implementazione del loop di gioco basato su tre fasi: l’elaborazione dell’input dell’utente; l’aggiornamento delle posizioni degli oggetti in gioco; la visualizzazione delle modifiche a video.

Al termine di ogni partita il punteggio realizzato andrà salvato su file nel caso sia migliore del precedente. In questo modo alla riapertura dell’applicazione i punteggi realizzati in precedenza saranno ancora disponibili.

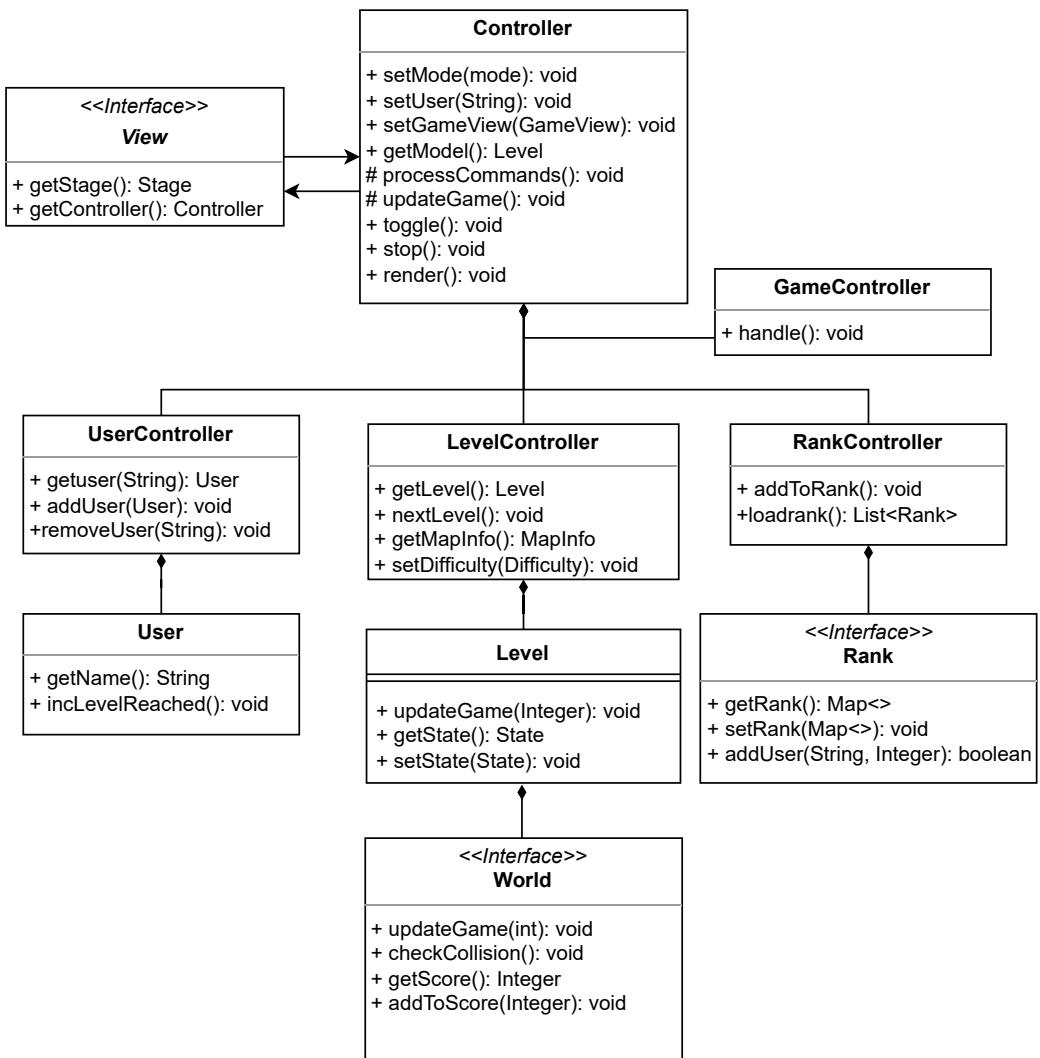


Figura 1.1: Schema UML dell’analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Il pattern architettonico Model-View-Controller (MVC) utilizzato nel gioco Brick Breaker offre una struttura organizzativa per separare i diversi componenti e gestire le interazioni tra di essi.

1. L'utente interagisce con la View, che rappresenta l'interfaccia utente del gioco. Attraverso l'input fornito dall'utente, la View è in grado di navigare tra le diverse finestre e comunicare con il controller per impostare i parametri necessari per l'esecuzione del gioco.
2. Una volta che l'utente ha fatto le sue scelte, il controllo viene passato al controller. Il controller istanzia il "mondo" di gioco e genera un primo render, che rappresenta la visualizzazione iniziale del gioco. Successivamente, il controller attende che l'utente prema il pulsante di "play" per iniziare effettivamente il gioco.
3. Dopo l'avvio del gioco, viene eseguito periodicamente il ciclo gestito dall'AnimationTimer (controller):
 - processo dei comandi da parte dell'utente (view)
 - aggiorna il mondo di gioco (model)
 - esegue una nuova renderizzazione (view)
4. In qualsiasi momento, tramite l'input dell'utente, la View può comunicare al model di interrompere o riprendere il ciclo di gioco. Questo consente all'utente di mettere in pausa o riprendere il gioco durante l'esecuzione.

5. Quando il controller chiama il metodo di aggiornamento del mondo nel model, esso comunica al model di eseguire l'aggiornamento sui propri oggetti: aggiorna la posizione delle palline in base alla loro velocità; la posizione della barra di controllo se richiesto dall'utente; controlla se si verificano collisioni e, se necessario, crea o rimuove elementi nel mondo di gioco; modifica lo score in base agli eventi.
6. Ad ogni iterazione del ciclo di gioco, il controller controlla se lo stato del gioco è cambiato. Se viene rilevato un cambiamento di stato, il controller è in grado di terminare gli aggiornamenti del model e comunicare alla View di cambiare la finestra da visualizzare. Questo permette di gestire le transizioni tra le diverse schermate di gioco, ad esempio quando si passa dallo schermo di gioco al menu principale.

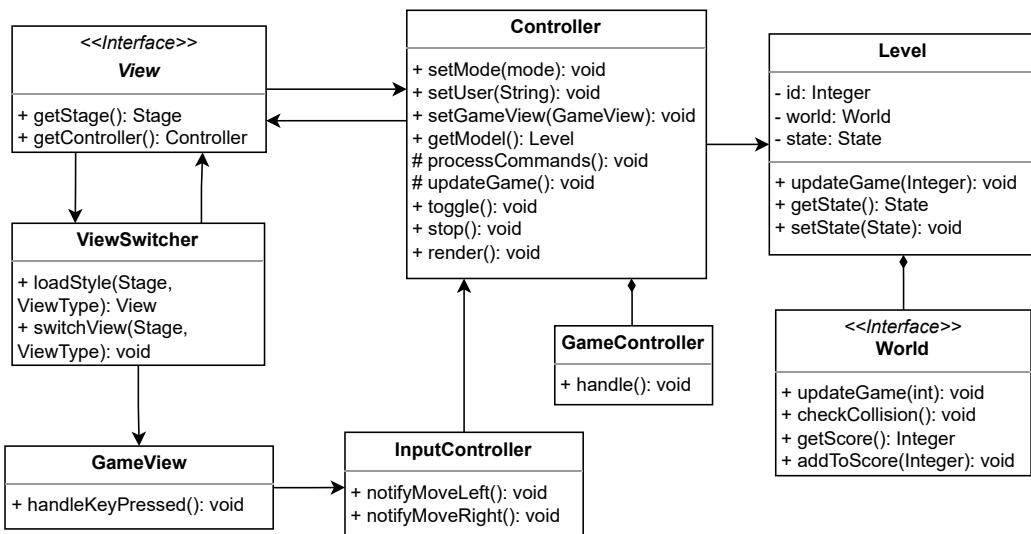


Figura 2.1: Schema UML architetturale di BRICK-BRK

2.2 Design dettagliato

Bighini Luca

World relation

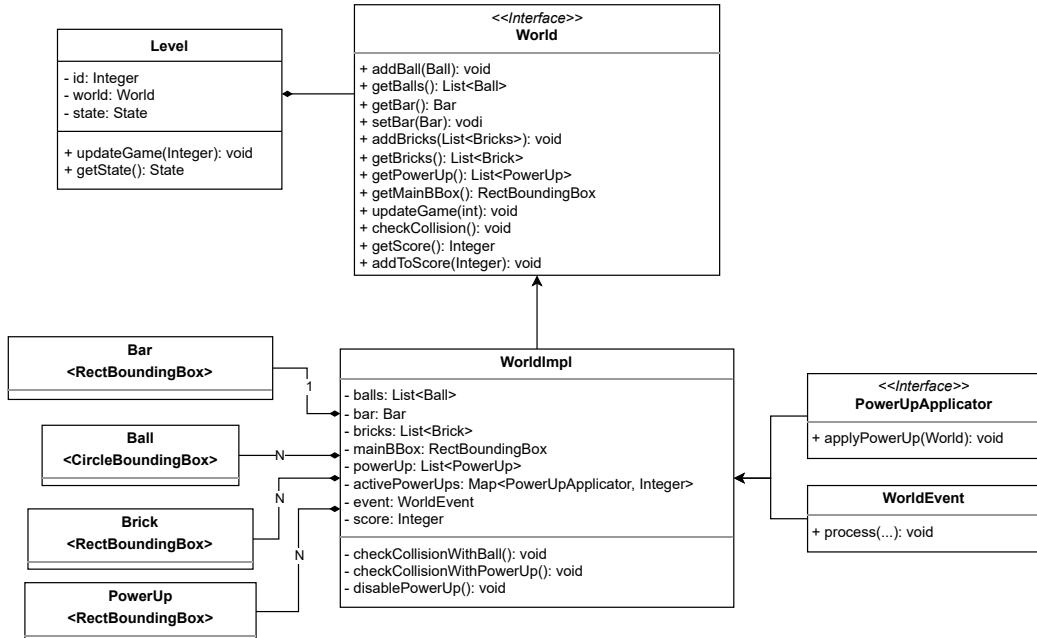


Figura 2.2: Schema UML architetturale World

Problema: Il world deve rappresentare un'immagine di tutti gli oggetti presenti in gioco (vedi Figura 2.2).

Soluzione: Per mantenere la visione più snella del model è stato creato un oggetto Level che rappresenta il model e separa lo stato di gioco da i singoli oggetti. Il world espone tutti i metodi per prendere o aggiungere GameObjects e mantenerne traccia in campi diversi. Inoltre, il mondo dispone del WorldEvent e del PowerUpApplicator per applicare modifiche agli oggetti presenti a seguito di collisioni.

Generic Bounding Box

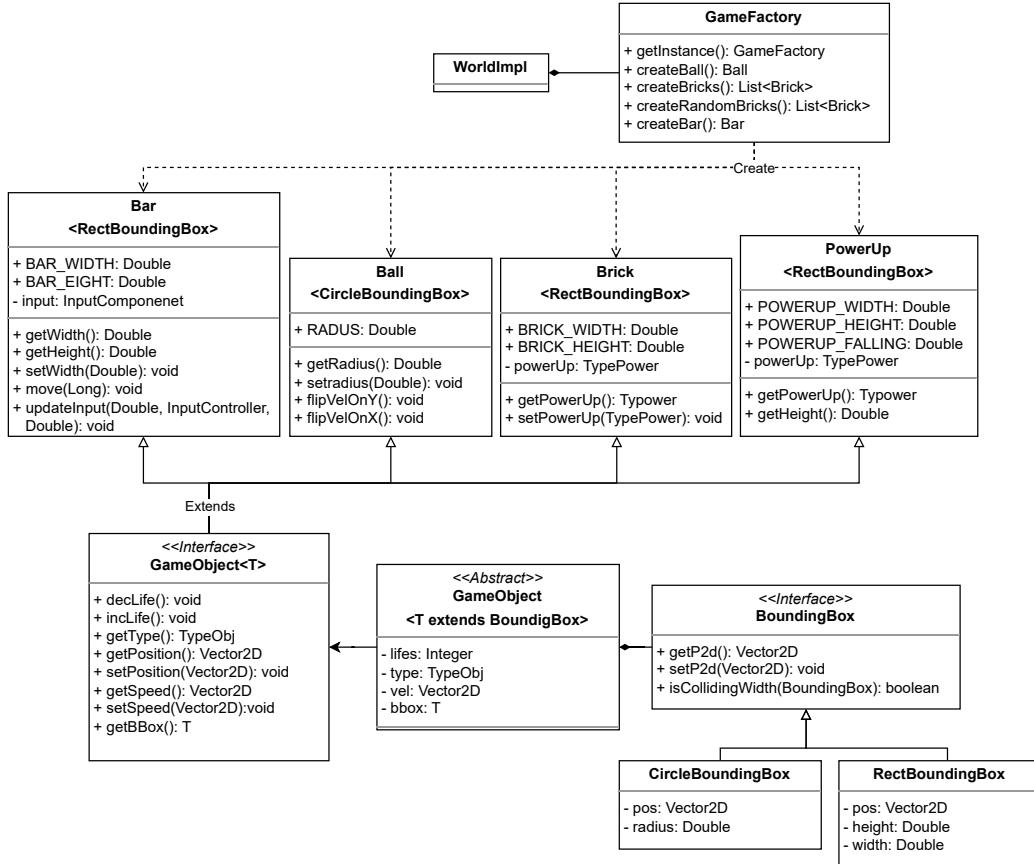


Figura 2.3: Schema UML architetturale BoundingBox

Problema: Implementare un metodo unico che mi ritornasse le Bounding box degli oggetti, ma di classi diverse in quanto una un cerchio e una un rettangolo (vedi Figura 2.3).

Soluzione: Si è optato per un metodo che ritorna un classe generica che estende l’interfaccia Bounding Box, in questo modo quando andremo a creare i diversi oggetti potremo decidere quale tipo di Bounding box dare all’oggetto così da poter creare una interfaccia comune **GameObject**. La classe **GameObject** poi viene estesa da tutti gli oggetti fisici utilizzando il **Template Method pattern** con la possibilità di aggiungere ulteriori metodi: Bar, metodi per moversi tramite input utenete; Ball, metodi per l’inversione della velocità in caso di collisione. Questi oggetti vengono creati tramite una classe **GameFactory** che utilizzando il **Singleton pattern** è in grado di mostrare in ogni punto del codice i metodi per la creazione dei singoli oggetti.

Power-up Applicator

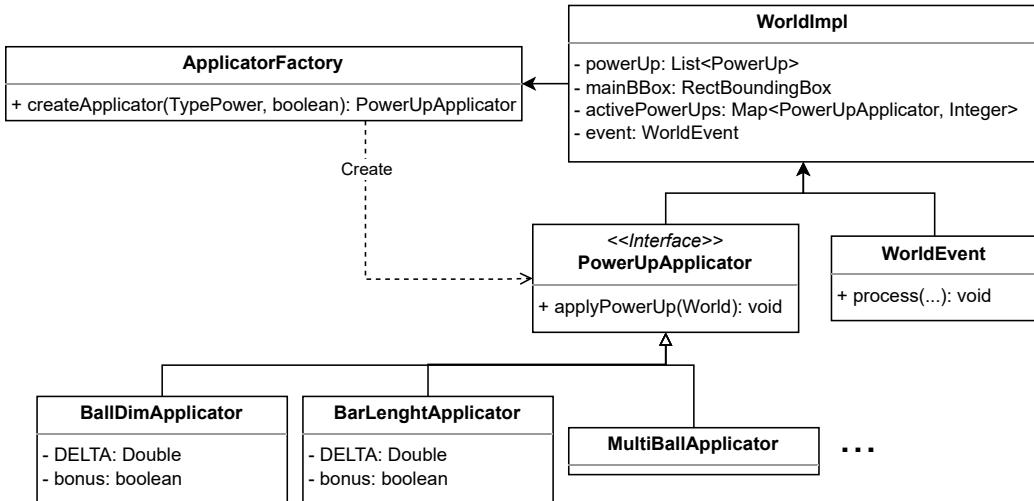


Figura 2.4: Schema UML architetturale Power-up Applicator

Problema: Creare un metodo univoco per l'applicazione di tutti i power-up (vedi Figura 2.4).

Soluzione: Per risolvere questo problema viene utilizzato lo **Strategy pattern** in modo che tutte le classi dei diversi power-up implementino un metodo comune (`applyPowerUp`) che applichi al mondo i cambiamenti necessari. Inoltre, nel world viene utilizzato il **Factory pattern** per creare all'evenienza le classi dei power-up. Nel world quando un brick viene distrutto se al suo interno vi è presente un TypePower non NULL, viene aggiunto alla lista powerUp un oggetto PowerUp di quel tipo. Quando questo oggetto viene in collisione con la Bar vengono applicate le relative modifiche agli oggetti e se il PowerUp ha una durata viene aggiunto alla lista degli activePowerUp, con il proprio contatore che viene decrementato ad ogni frame e al suo termine il power-up viene disabilitato.

Input della barra

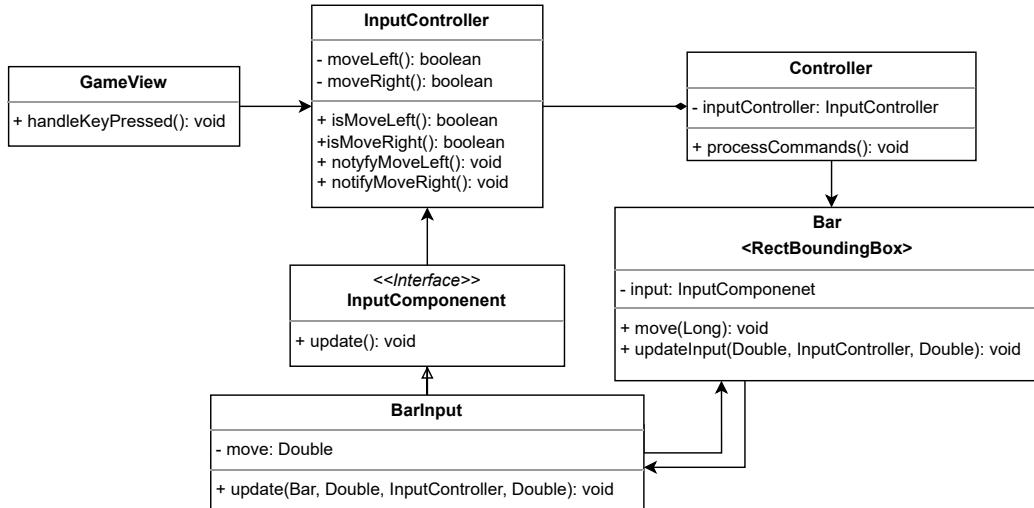


Figura 2.5: Schema UML architetturale Input Observer

Problema: Creare un metodo il estendibile che potesse relazionare l'input preso dalla view con l'aggiornamento della posizione della barra nel model (vedi Figura 2.5).

Soluzione: E' stato utilizzato l'**Observer pattern**: la View funge da oggetto osservabile, mentre il Controller agisce come osservatore. Quando avviene un evento nella View che richiede un'azione, la View chiama il metodo "notifyMove()" sul Controller. Il Controller, a sua volta, controlla ad ogni iterazione del loop se la variabile è cambiata e, se necessario, chiama i metodi per la modifica degli oggetti del Model. Ciò favorisce la separazione delle responsabilità e permette di mantenere un codice modulare ed estensibile.

Francesco Agostinelli

Creazione del mondo e degli oggetti di gioco

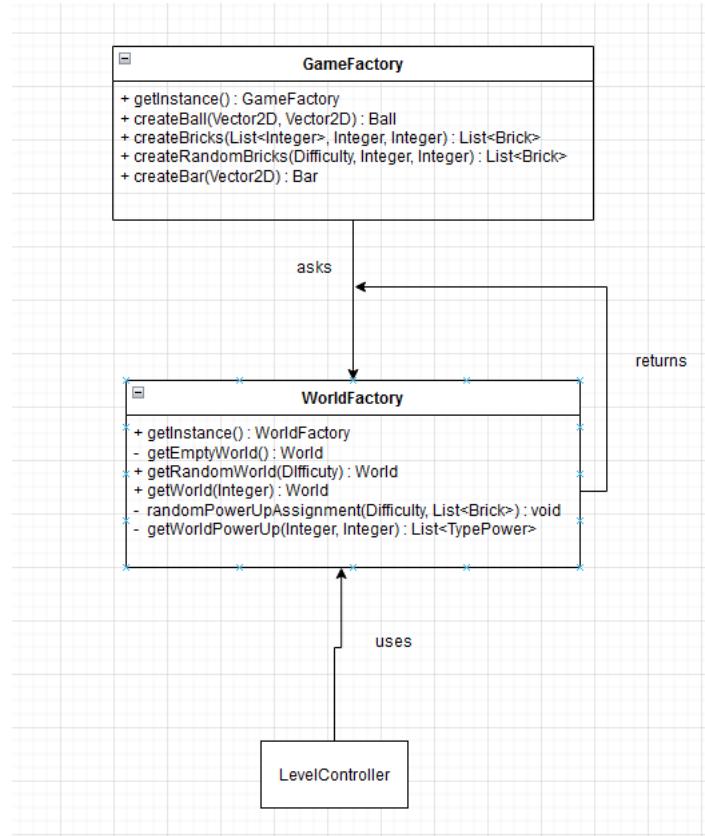


Figura 2.6: Schema UML dell'interazione e utilizzo delle Factory

Mi sono occupato della creazione degli oggetti di gioco creando due factory principali, cioè la **GameFactory** e la **WorldFactory**.

In particolare:

- **GameFactory** si occupa della creazione di tutti gli oggetti di gioco, cioè della barra, delle palline e dei bricks.
- **WorldFactory** si occupa della creazione del mondo di gioco, utilizzando quindi la GameFactory per creare gli oggetti sopracitati ed integrarli in un oggetto World che sarà poi ritornato da ogni metodo esposto.

Ho reso entrambe le factory degli oggetti ad **accesso unificato**, utilizzando il singleton pattern, poichè diverse istanze delle stesse sarebbe stato solo un temporaneo spreco di memoria, infatti basta un'istanza utilizzabile in ogni classe, per poter eseguire efficientemente il lavoro per cui sono state implementate.

Difficoltà e modalità di gioco

In questo progetto era necessario implementare delle difficoltà di gioco, cioè le classiche, ovvero : **facile**, **medio** e **difficile** ed anche delle modalità : **endless** e **levels**.

Nella modalità **endless**, l'utente dovrà giocare fin tanto che non andrà incontro ad una sconfitta, ciò significa che l'utente dopo ogni partita vinta, dovrà giocarne una nuova.

Ovviamente non esiste un limite massimo di livelli giocabili.

L'utente, potrà anche scegliere la difficoltà con cui giocare tale modalità, che oltre a quelle descritte prima, c'è la possibilità di giocare con difficoltà "random", cioè che ogni livello avrà una difficoltà scelta casualmente.

Nella modalità **levels**, cioè a livelli, l'utente sceglierà uno specifico livello con una propria difficoltà e mappa.

Le difficoltà, specificate in un apposito enumeratore, vengono poi utilizzate in modo specifico dalla **WorldFactory**, la quale crea un mondo sulla base della difficoltà scelta dall'utente. Questo significa che la factory crea un mondo in base a determinati parametri che insieme rappresentano la difficoltà scelta, cioè :

Sulla base di questi valori, è possibile creare un possibile mondo.

WorldFactory

Problema: Implementare nel model sia la creazione di un livello predefinito che randomico.

Soluzione: Dato che si parla di creazione di oggetti World, mi è sembrato sia ovvio che di estrema importanza, implementare la classe **WorldFactory**, che appunto espone i due metodi più importanti **getWorld(Integer)** e

getRandomWorld(Difficulty), che rispettivamente creano un mondo basato su uno dei livelli del gioco e uno randomico, scegliendo casualmente i parametri dinamici prima citati.

Avendo utilizzato il pattern factory, la manutenzione viene semplificata, in quanto eventuali correzioni o estensioni, andranno ad influire solamente la classe stessa.

Difficulty

Problema; Implementare delle difficoltà, riusabili in entrambe le modalità.

Soluzione: Ho immaginato di dover definire, a livello di dati, degli insiemi di mondi, cioè dividere idealmente i mondi facili, da quelli medi e dai difficili. La scelta giusta, oltre a creare un enumeratore era assegnargli dei dati specifici, cioè:

1. **Numero massimo di bricks** che il livello può avere (salendo di difficoltà, aumenta il suo valore)
2. **Numero di power up bonus**, cioè quanti power up vantaggiosi per l'utente sono presenti nella mappa (all'aumentare della difficoltà, tale valore diminuirà).
3. Numero di bricks massimo con **più di una vita** che il livello può avere (anch'esso aumenterà, all'aumentare della difficoltà).

Questi valori, fungono da parametri durante il processo di creazione di un mondo casuale.

Livelli di gioco e leaderboard

Nella modalità a livelli, l'utente può scegliere e giocare ad un livello predefinito. Tale livello non è niente di meno che un oggetto Level, sviluppato con il contributo dei colleghi.

In particolare, i livelli sono salvati su un file json, che appunto funge da sorgente dati.

Ogni livello, appena scelto, viene caricato da file e ne viene creato un corrispondente mondo, quindi mi sono dovuto occupare di una classe che carica tali dati.

RankController, GameRank e RanksView

Problema: Creare un sistema di classifiche globali e locali ad ogni livello.

Soluzione:

Ho creato una classe chiamata GameRank, che potesse rappresentare i dati di una classifica, cioè nome del giocatore e punteggio. Tale oggetto viene manipolato dal **RankController** che si occupa di leggere i dati da file utilizzando il **ResourceLoader**. La lettura di questi dati viene fatta esclusivamente all'apertura ed interazione con la RanksView, raggiungibile dalla HomeView.

Ogni giocatore, una volta terminata una partita, che come detto può essere uno o una serie di livelli, verrà aggiunto alla classifica tramite ..., che quindi scrive i dati del giocatore sul file Json, nella classifica corrispondente.

Ogni classifica, indipendentemente dalla modalità, viene categorizzata dal suo indice, infatti le ranks locali dei livelli saranno divise per indice e stesso le ranks globali della modalità **"ENDLESS"** divise per difficoltà.

In caso si volesse aggiungere un nuovo livello basterà modificare la sorgente dati, invece se si volesse aggiungere una nuova modalità, nell'ottica del ranking, basterà creare un apposito file ed ovviamente, modificare soltanto il controller apposito.

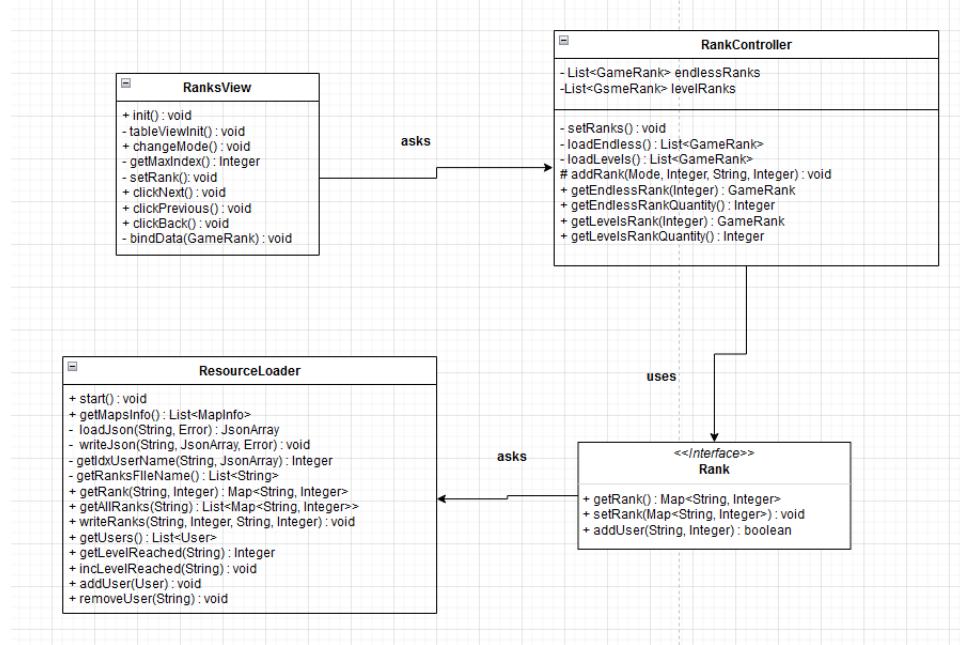


Figura 2.7: Schema UML del sistema di ranking

Tellarini Pietro

Abstract Controller

Problema: Il controller deve essere istanziabile una sola volta dalla view e poter essere fruibile in modo univoco senza dover fare riferimento ai controller più specifici che si occupano direttamente della gestione del modello (vedi Figura 2.8).

Soluzione: Essendo controller specifici relativi alla manipolazione di determinate parti del modello si è presentato il problema di poter fornire un’istanza unica e comune che potesse essere implementata dalla parte di grafica del gioco e che raccogliesse tutte le istanze dei controller specifici necessari al settaggio iniziale del gioco per poi poter avviare una sessione di gioco vera e propria della partita. Nello specifico si è giunti alla risoluzione di tale problema creando una classe astratta denominata AbstractController che contenesse le istanze delle classi visibili anche in figura (Vedi Figura 2.8) e i relativi getter e setter, in modo tale da rendere fruibili tali controller specifici (InputController, UserController, RankController, LevelController, ErrorListener) in tutti nelle classi che avrebbero esteso la classe astratta. Nel caso di questo applicativo rende fruibile le parti del modello in maniera

distinta e ordinata e permette una ristrutturazione più agevole del Controller vero e proprio nel caso si debbano apportare delle modifiche agli scenari di uscita, di pause e di avvio del ciclo di gioco o dell’interazione con l’interfaccia grafica. La classe astratta di fatto viene estesa quindi dal controller vero e proprio e costituendo un Template Pattern utile nel caso si voglia creare un nuovo Controller di gioco mandenendo invariato il modello al quale fa riferimento.

In particolare la classe Controller determina infatti tutta la logica di gioco necessaria a creare una corretta istanza del modello, a gestirne l’update e ad effettuarne il successivo render. Tale istanza contiene di fatto le informazioni relative alla sessione di gioco che l’utente sta giocando e le informazioni relative alla view della vera e propria finestra di gioco.

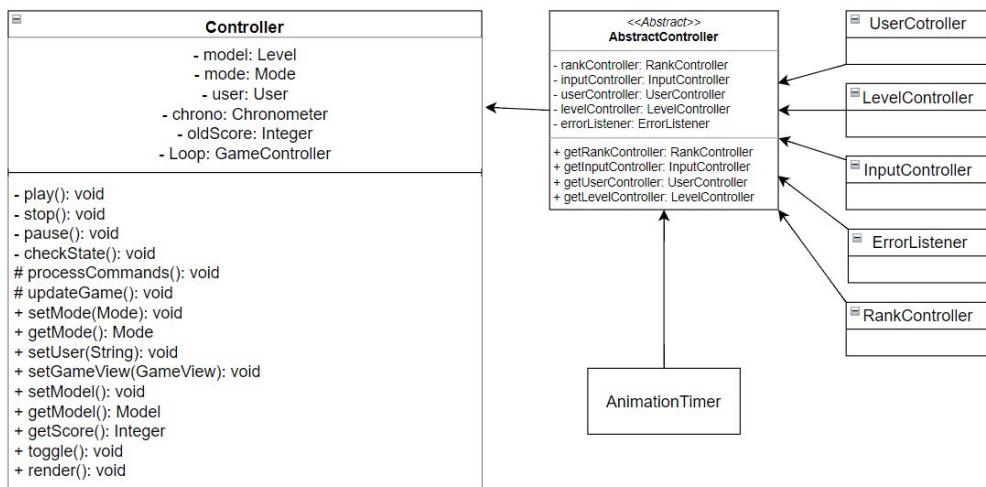


Figura 2.8: Schema UML architetturale

Game Loop

Problema: Creare un game loop non bloccante e facilmente estendibile che interagisca con il controller (vedi Figura 2.8).

Soluzione: Il problema quindi di creare un loop di gioco nel controller che potesse essere fermato e analizzato in modo continuo durante la sua esecuzione, favorendo quindi anche la messa in pausa, è stato risolto estendendo la classe astratta AbstractController, che fornisce un Template per il controller vero e proprio, con la classe JavaFX denominata AnimationTimer. Quest'ultima permette di gestire tramite l'override del metodo handle il loop di gioco. Nello specifico il metodo viene eseguito in modo continuo una volta chiamato il metodo start implementato dall'AnimationTimer e può essere fermato chiamando la stop.

Error Listener

Problema: Tener traccia degli eventuali errori durante la creazione delle istanze di gioco e l'utilizzo dei metodi del model.

Soluzione: Per quanto concerne la gestione degli errori all'interno del gioco, utili soprattutto durante la creazione del modello e al caricamento delle risorse, si è deciso di creare la classe ErrorListener che tiene traccia degli errori del codice e espone metodi per riconoscere il verificarsi di un errore. Essa rispetta quello che viene definito Observer Pattern e contiene una lista degli Errori che si possono verificare nel gioco, esponendo il metodo di notifica.

JsonUtils

Problema: Poter gestire le risorse riguardanti il salvataggio degli utenti, dei loro progressi e il caricamento delle mappe di gioco.

Soluzione: Per poter manipolare facilmente le informazioni relative agli Utenti, ai Ranks ed infine anche alle mappe di gioco si è scelto di conservare tali dati all'interno di file di tipo Json. Per quanto riguarda il caricamento iniziale dei dati, l'aggiornamento ed il loro salvataggio si è optato per creare metodi generici che prendessero come parametri il path del file e il tipo di dato da prendere in lettura (Type di java.lang.reflect). Nello specifico, dato che non è permesso lavorare in scrittura sui files contenuti all'interno di un'applicativo in jar, si è scelto di salvare tali files json all'esterno del jar e leggerne di conseguenza i dati. In caso tali files non fossero presenti, ad esempio a causa di uno spostamento dell'applicazione, essi verranno ricreati allineandosi ad uno stato iniziale mantenuto all'interno del jar.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

- **FactoryTest:** testa i metodi per la creazione degli oggetti di gioco.
- **BarInputTest:** testa la relazione tra l'event listener nel controller e l'effettivo cambio della posizione dell'oggetto nel model.
- **StateTest:** testa l'effettivo cambio dello stato di gioco al variare delle diverse situazioni possibili.
- **UserTest:** testa la corretta aggiunta e rimozione di uno user da file e la lettura dei suoi campi.
- **CollisionTest:** testa le collisioni tra i vari oggetti di gioco.
- **WorldTest:** testa i metodi per l'aggiunta e la rimozione di oggetti nel mondo e le relative conseguenze, come l'incremento del punteggio.
- **RankTest:** testa il caricamento, l'aggiunta e la modifica di rank.
- **LevelControllerTest:** testa il caricamento corretto delle mappe e del mondo.

3.2 Metodologia di lavoro

Come gruppo, fin dalle prime fasi di progettazione, abbiamo cercato di dividere il carico di lavoro creando un diagramma UML iniziale. Abbiamo identificato le classi necessarie, le loro relazioni e abbiamo assegnato a ciascuna classe dei compiti specifici. Questo ci ha permesso di avere una visione chiara del modello MVC.

In base a questa suddivisione dei compiti, ci siamo accordati per iniziare la creazione di una parte del modello ciascuno, per poi procedere con le altre componenti successivamente.

Classi sviluppate in gruppo

- Implementazione Controller
- Implementazione View e viewType

Bighini Luca

- Classi di utility (*brickbreaker.common*)
 - Chronometer
 - Error
 - State
 - TypeObj
 - TypePower
 - TypePowerUp
- InputController
- ErrorListener
- ApplicatorFactory
- World
- *world.gameObjects.bounding*
- *world.gameObjects.collision*
- *input*
- Brick
- PowerUp
- EndGameView

Ho contribuito a:

- GameFactory

- GameObject

- Level

Agostinelli Francesco

- Classi di utility (*brickbreaker.common*)
 - Difficulty
 - GameImages
 - GameObjectsImages
 - Vector2D
- UserController
- SetUp
- Level
- Ball
- WorldFactory
- GameFactory
- DifficultyMenuView
- GameView
- HomeView
- LevelsMenuView
- RanksView
- ViewSwitcher

Ho contribuito a:

- GameObject

- World

Tellarini Pietro

- Classi di utility (*brickbreaker.common*)
 - Chronometer
 - Mode
 - JsonUtils
 - ImageUtils
- LevelController
- RankController
- AbstractController
- GameObject
- Bar
- User
- Rank
- MapData
- SetUpView

Ho contribuito a:

- Level
- World

3.3 Note di sviluppo

Abbiamo riconosciuto che la comprensione e l'analisi più approfondita delle funzionalità richieste dal controller e dalla view hanno portato a una migliore comprensione dei requisiti e delle interazioni tra le diverse componenti del sistema. Di conseguenza, abbiamo ritenuto necessario apportare modifiche alle componenti di più alto livello del model per garantire una corretta implementazione, nonostante l'idea iniziale proposta all'interno di un UML.

Un elemento fondamentale in questo processo è stato l'utilizzo di Git. Abbiamo creato diversi branch per consentire a ciascun membro del team di lavorare in modo indipendente e testare il proprio codice. Tuttavia, abbiamo

sempre adottato un approccio di integrazione continua, riunendo le modifiche sul branch principale (main).

Questa metodologia ci ha consentito di mantenere una traccia chiara delle modifiche apportate da ciascun membro del team e di garantire che il codice fosse testato e stabile prima di essere unito al ramo principale.

Bighini Luca

- **Optional** in tutte le situazioni in cui sono presenti campi che possono non contenere un valore, il loro utilizzo mi ha permesso di non usare *null* all'interno del codice;
- **Stream** utilizzati per la lettura, estrazione e manipolazione di gruppi di dati;
- **Generici** mi hanno permesso di creare campi e interfacce comuni condivise da più classi, separate dalle classi effettive, consentendo una maggiore modularità e estensibilità del codice;
- **AnimationTimer** classe java utilizzata per la creazione del game loop;
- **Multithreading** è stato utilizzato per la creazione del cronometro in esecuzione parallelamente all'aggiornamento del mondo;
- **JavaFx** framework utilizzato per lo sviluppo della view di simulazione;
- **Google GSON** libreria Google utilizzata per la manipolazione dei file JSON;
- **Gradle** creazione del progetto utilizzando Gradle (*build.gradle.kts* fornito su Virtuale).

Agostinelli Francesco

- **Optional** in tutte le situazioni in cui sono presenti campi che possono non contenere un valore, il loro utilizzo mi ha permesso di non usare *null* all'interno del codice;
- **Stream** utilizzati per la lettura, estrazione e manipolazione di gruppi di dati;
- **Strategy pattern** usato per permettere di cambiare "comportamento" a runtime per la factory dei Power up.

- **Factory pattern** usato per la creazione degli oggetti di gioco e del mondo di gioco;
- **JavaFx** framework utilizzato per lo sviluppo della view di simulazione;
- **Google GSON** libreria Google utilizzata per la manipolazione dei file JSON;

Tellarini Pietro

- **Optional** in tutte le situazioni in cui sono presenti campi che possono non contenere un valore, il loro utilizzo mi ha permesso di non usare *null* all'interno del codice;
- **Stream** utilizzati per la lettura, estrazione e manipolazione di gruppi di dati;
- **AnimationTimer** classe java utilizzata per la creazione del game loop;
- **JavaFx** framework utilizzato per lo sviluppo della view di simulazione;
- **Google GSON** libreria Google utilizzata per la manipolazione dei file JSON;
- **Gradle** creazione del progetto utilizzando Gradle (*build.gradle.kts* fornito su Virtuale).

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Bighini Luca

Sono abbastanza soddisfatto del lavoro che ho svolto nel mio primo progetto di programmazione orientata agli oggetti (OOP). Durante l'intero processo di progettazione, ho sentito un miglioramento delle mie capacità e della mia sicurezza. Questo progetto mi ha aiutato a chiarire alcuni aspetti del corso che mi erano poco chiari. In particolare, la fase iniziale è stata la più difficile per me e, guardando indietro, probabilmente avrei adottato soluzioni diverse per alcuni problemi, sapendo ora quale sarebbe stata la scelta più utile.

Valuto positivamente l'esperienza di lavorare in gruppo poiché mi ha aiutato a migliorare nella comunicazione e nel confronto. Tuttavia, a volte ho notato che non tutti i membri del gruppo contribuivano nella stessa misura e, soprattutto, quando venivano apportate modifiche al codice condiviso, ciò causava confusione. In futuro, sarebbe importante che tutti rispettassero il lavoro degli altri partecipanti e fossero consapevoli dei possibili impatti delle proprie modifiche.

Sperimentare le basi dello sviluppo di giochi è stato appassionante e divertente, ma non è un campo che mi affascina a tal punto da volerlo approfondire in futuro.

Agostinelli Francesco

Mi ritengo mediamente soddisfatto del lavoro svolto. E' stato sicuramente molto stimolante e come ogni progetto l'ho preso come una sfida contro sé stessi per migliorare le proprie competenze e conoscenze nello sviluppo di progetti importanti ed in team, per quanto, francamente, non ne sia un

amante. La tematica dello sviluppo di video games è sicuramente una tematica di un certo spessore per l'industria odierna e, nonostante non sia nella cima dei miei interessi, mi ha coinvolto parecchio, perché alla fine pur sempre di sviluppo software si tratta e certi aspetti, sono sicuro, non sarei riuscito a trovarli magari in progetti con delle tematiche più classiche.

Valuto positivamente l'esperienza di lavoro in gruppo, nonostante le difficoltà incontrate e riconosco, ora più che mai, che la progettazione e la comunicazione siano degli aspetti primi da tenere in considerazione, ma che spesso vengono, negativamente, sottovalutati.

Tellarini Pietro

In conclusione, mi ritengo pienamente soddisfatto del lavoro svolto e di come siamo riusciti a gestire e ha superare di gruppo le crescenti difficoltà che si possono verificare durante la creazione di un applicativo di moderata complessità come questo.

Tale elaborato mi ha permesso, oltre che di mettere in pratica ciò che ho imparato dal corso di Programmazione ad Oggetti, di toccare con mano come possa essere complicata ma al contempo fondamentale un'attenta strutturazione e analisi del progetto, utile a rispettare i tempi di creazione di un applicativo e suddivisione del codice da implementare.

Ho apprezzato molto lavorare in gruppo sebbene tale processo di creazione richieda una forte sincronia e comunicazione tra le parti: in quest'ottica si è rivelato estremamente utile un software di versioning come Git consentendoci di lavorare in modo concorrente allo stesso progetto e suddividendoci meglio i carichi di lavoro.

In conclusione ci tenevo a precisare che sebbene inizialmente abbia avuto qualche difficoltà nel cambiare paradigma di programmazione, in quanto sempre abituato per lo più a linguaggi imperativo procedurali, ho rivalutato estremamente Java, che fino a prima non avevo mai utilizzato, come linguaggio di programmazione apprezzando l'idea alla base della programmazione ad oggetti, sebbene sia convinto che alle volte un approccio che mescoli più paradigmi possa rivelarsi talvolta più adattabile alla situazione che si vuole implementare.

A livello lavorativo ritengo che il paradigma ad oggetti sarà alla base di molti percorsi che si potrebbero intraprendere e la conoscenza di un linguaggio come Java semplificherà l'apprendimento di nuovi linguaggi e di linguaggi abbastanza simili: penso a C-sharp (del quale comunque abbiamo affrontato già alcuni argomenti) se si vuole abbracciare il modo dello sviluppo di videogiochi e integrare motori grafici come Unity o eventualmente a javascript per lo sviluppo web utilizzando node.js.

In conclusione ritengo che il corso di Programmazione ad Oggetti rappresenti un corso fondante del nostro corso di laurea, e che si ponga come obiettivo quello di formare sviluppatori software di un determinato calibro che sappiano correttamente implementare un applicativo in Java, facendo uso dei Pattern che tale linguaggio mette a disposizione, in modo tale da scrivere codice che non solamente funzionante ma soprattutto di qualità.

4.2 Difficoltà incontrate e commenti per i docenti

L'inizio di un progetto senza precedenti esperienze ha presentato delle difficoltà.

Durante il percorso, si sono riscontrate alcune problematiche con il quarto componente del gruppo a causa di questioni personali che lo hanno portato ad essere assente per gran parte del tempo. Di conseguenza, non è stato possibile integrarlo nell'ultima fase del progetto.

Appendice A

Guida utente

La schermata iniziale presenta la scelta dell'utente, la sua aggiunta e la sua rimozione.

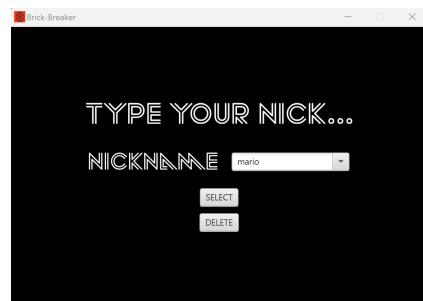


Figura A.1: Start view

Nella seconda schermata si può scegliere la modalità di gioco o di visualizzare la leaderboard.

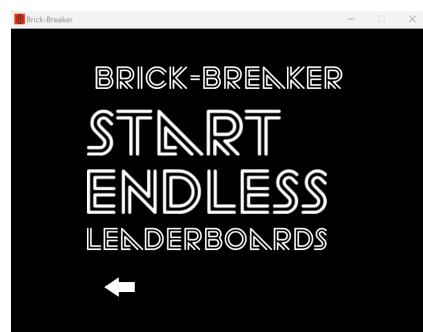


Figura A.2: Home view

La modalità **Level** presenta la scelta del livello, mentre la modalità **Endless** presenta la scelta della difficoltà.



Figura A.3: Level and Endless view

Nella schermata principale di gioco, l'avvio o la messa in pausa possono essere controllati utilizzando il carattere di "whitespace". Inoltre, la barra presente nella schermata può essere spostata utilizzando i tasti di direzione destra e sinistra.

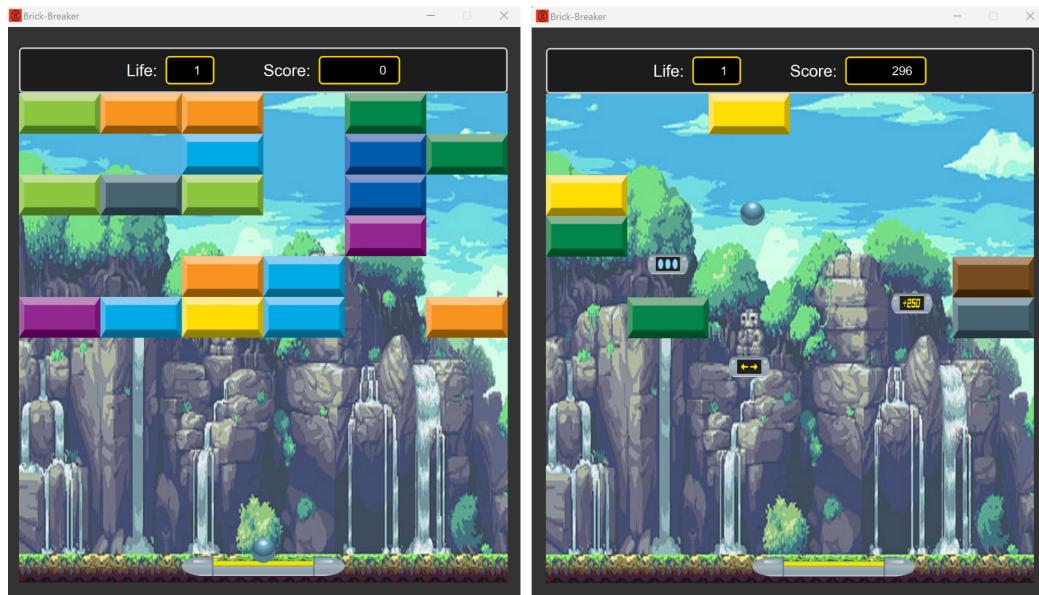


Figura A.4: Game view

Nella schermata dei rank è possibile visualizzare i rank delle diverse modalità di gioco cliccando sull'etichetta "endless" o "level".

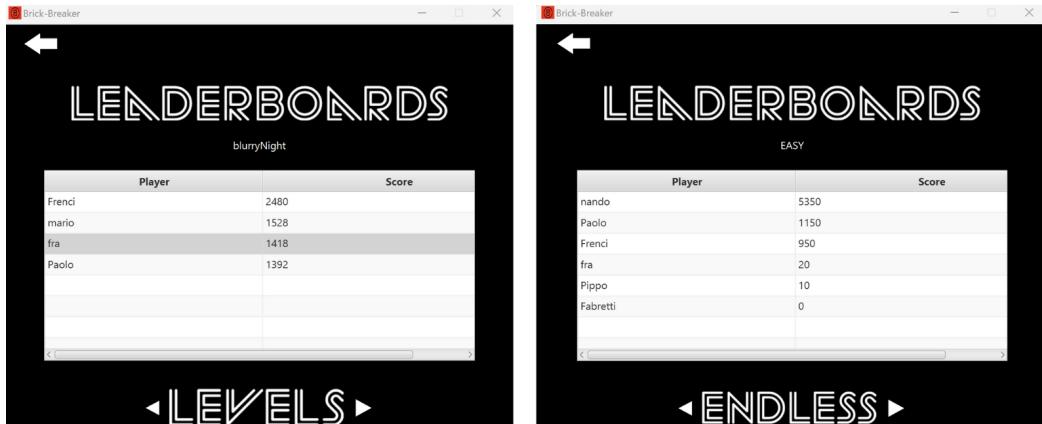


Figura A.5: Rank view

Il gioco termina con le schermate di vittoria o sconfitta, con la possibilità di tornare alla home.



Figura A.6: End view