a) Numero di matricola:

1862806

b) Corso:

Teledidattica

c) Nome, Cognome:

Luca, Monari

d) Decisioni di progettazione:

d.1) Gestione del profilo utente, nickname, avatar, partite giocate, vinte e perse, livello

La gestione del profilo utente comprende la scelta del nickname, dell'avatar con cui giocare, tenendo traccia delle partite giocate (vinte e perse), dell'ultimo livello raggiunto e del punteggio piu' alto realizzato. Il nickname e l'avatar devono essere selezionati dal giocatore, mentre gli altri parametri sono aggiornati automaticamente del programma. Per semplificare la gestione del profilo utente, ho creato una classe dedicata che contiene i dati del player, chiamata "PlayerData". Avere una classe dedicata facilita la gestione delle informazioni dell'utente. PlayerData contiene I campi:

- nickname: identifica un utente (input dell'utente)
- avatar: indica l'avatar che e' stato selezionato tra le 4 versioni del Bomberman disponibili (input dell'utente)
- lastLevel: tiene traccia dell'ultimo livello raggiunto dal giocatore. Siccome ci sono 3 livelli giocabili, il terzo e' l'ultimo livello giocabile. Una volta vinto il terzo livello, al giocatore viene proposto di rigiocare il terzo livello con piu' nemici (aggiornato automaticamente)
- playedGames, winGames, lostGames: sono interi che tengono traccia delle partite giocate, vinte e perse (aggiornato automaticamente)
- score: equivale al punteggio piu' alto raggiunto dal giocatore (aggiornato automaticamente)

Per mantenere le informazioni utente tra diverse partite o quando l'app e' stata chiusa, la classe "PlayerData" ha dei metodi specifici per salvare e caricare le informazioni utente nel file "saveGames.txt". La classe ha un metodo savePlayerData per salvare i valori correnti nel file, mentre il metodo statico "readPlayerData" legge i valori degli utentiprecedenti e restituisce una mappa col nickname come chiave e i dati utente come valori. I dati sono salvati come stringhe, con virgole come separatori. Questo modo di salvare i dati e' stato scelto per la sua semplicita', essendo molto efficace da implementare. Il livello di sicurezza e' minimo (si possono facilmente manomettere i dati), ma i dati salvati non contengono informazioni sensibili. Questo metodo supporta molto facilmente l'espansione del codice con l'aggiunta di nuovi campi.

Per l'interfaccia grafica, l'utente puo' selezionare il nickname e l'avatar nel menu' principale. Se il nickname viene riconosciuto (utilizzando la mappa generata dal metodo statico "readPlayerData"), le altre informazioni del giocatore vengono mostrate sullo schermo (partite giocate, ultimo livello, ecc…) . La gestione della finestra e degli input dell'utente avviene nella classe MainMenu, che ha la funzione di costruire e mostrare il menu' principale. Attraverso quest'ultimo, e' possibile selezionare il numero di nemici per diminuire o abbassare la difficolta' del gioco.

I valori di PlayerData selezionati nel MainMenu sono usati per inizializzare il livello nella classe GameApp. Questa classe contiene il Game Loop che gestisce lo svolgimento della partita. I valori del profilo utente vengono aggiornati tutte le volte che il giocatore vince o perde una partita, e vengono salvati su file. Questo e' fatto per evitare di considerare partite incomplete (e serve a me per barare, perche' se sto perdendo posso chiudere l'app e la mia sconfitta non viene segnata). Un'alternativa sarebbe aggiornare le informazioni utente a inizio del gioco, ma in questo caso se si chiude il gioco durante una partita, playedGames sarebbe maggiore della somma delle partite vinte e perse (smascherando quanto sono terribile a Super Bomberman). Penso sia meglio aggiornare tutte le informazioni a fine partita, salvando i dati una sola volta in modo tale da mantenere un codice piu' semplice, anche da mantenere.

Infine, la gestione del profilo utente dell'app e' pensata per essere personalizzata per ogni utente, e' possibile infatti tenere traccia delle partite giocate salvando i risultati migliori e puo' essere facilmente ampliata dell'utente da salvare.

d.2) Gestione di una partita completa con almeno due livelli giocabili, schermata hai vinto, game over, continua, due tipi di nemici con grafica e comportamento di gioco differenti, con gestione del punteggio, delle vite, di 3 power up.

La gestione di una partita con piu' livello giocabili ha richiesto una strutturazione del codice complessa, che sfrutta design pattern, funzionalita' di JavaFX e sviluppo delle interazioni tra entita' nel gioco. Per cercare di evitare una relazione estremamente prolissa, di seguito è descritta una panoramica delle sue principali componenti e funzionalita' (ogni implementazione puo' essere dettagliata nel codice o all'orale).

Per la gestione del gioco e l'interfaccia utente, le principali classi sono:

- GameApp: Punto d'ingresso principale del gioco, avvia l'interfaccia utente, gestisce la transizione tra le varie schermate di gioco, inclusi menu, livelli e schermate di vittoria o sconfitta.
- MainMenu: Gestisce la schermata del menu principale dove i giocatori possono iniziare una nuova partita, caricare una partita salvata, modificare impostazioni o uscire dal gioco.

GameApp, in particolare, contiene la gestione del gioco e del game loop, questo significa istanziare i modelli principali, come il modello della mappa di gioco (StageModel) e il modello del Bomberman (PlayerModel); creare le classi di tutti i Controller del gioco che implementano il pattern MVC; istanziare il timer del livello; far partire il gioco. La mappa di gioco viene caricata a seconda del livello indicato nel PlayerData. La partita di un livello puo'

finire in due modi: 1) sconfitta, se il giocatore perde tutte le vite o il timer raggiunge zero; 2) vittoria, se il giocatore uccide tutti i nemici e si posiziona sul blocco per il nuovo livello. A seconda del caso, viene aperta una finestra di dialogo che chiede di nel caso 1) riprovare di nuovo, nel caso 2) continuare al livello successivo. In entrambi i casi e' possibile uscire dall'applicazione o tornare al menu' principale. Se viene raggiunto il massimo livello (livello 3), l'applicazione propone di rigiocare il livello 3 con piu' nemici per aumentare la difficolta'.

Per la gestione della mappa di gioco e delle entita' in gioco (bomberman e nemici), le principali classi sono:

- StageModel: Rappresenta la struttura di una mappa di un livello di gioco, gestendo gli elementi statici, sia permanenti (come blocchi non-distruttibili), semi-permanenti (come blocchi distruttibili), che temporanei (come bombe e power-ups).
- EntityModel e sue derivazioni (PlayerModel, EnemyModel, EnemyModel1, EnemyModel2): Definiscono le caratteristiche e i comportamenti delle entità mobili all'interno del gioco, inclusi i giocatori e i nemici.

Il campo di gioco e' organizzato come un array bidimensionale di blocchi (chiamati Tile nel programma). StageModel si occupa di gestire i blocchi (tile), inizializzando i blocchi distruttibili e non distruttibili, e controllando tutte le operazioni che coinvolgono i blocchi (come la distruzione e la sostituzione di blocchi). I blocchi possono essere blocchi distruttibili o non distruttibili, blocchi vuoti su cui passano i giocatori (EmptyTile), o blocchi speciali come i PowerUps o il blocco per accedere al nuovo livello (SpecialTile).

La classe astratta EnitityModel e' la classe di base per tutte le entita' nel gioco, sviluppando le caratteristiche e funzionalita' basilari di giocatore/nemico: muoversi nel campo di gioco, segnalare i blocchi occupati dall'entita', controllare in quale direzione ci si puo' muovere, i metodi per perdere vita e aggiornare lo stato del player a seconda del tempo trascorso nel game loop (per esempio per calcolare la velocita' di movimento).

La classe EnemyModel è una classe astratta a sua volta, che si assicura che i nemici implementino un movimento particolare, che viene aggiornato nello stato durante il gioco. I nemici EnemyModel1 e EnemyModel2 hanno comportamenti differenti: il primo è un nemico semplice che si muove avanti e indietro; il secondo si muove in modo casuale in una delle direzioni disponibili (che non sono occupate da blocchi) e quando viene colpito da una bomba si teletrasporta in una nuova casella libera. Dato che EnemyModel2 si muove in una direzione casuale, e' molto difficile prevedere i suoi spostamenti. Per diminuire la difficoltà, la velocità di EnemyModel2 viene diminuita. Per gestire comportamenti diversi, si potrebbe implementare lo strategy pattern. In questo caso lo strategy pattern non è stato implementato perché abbiamo solo 2 nemici da gestire. Il comportamento del nemico e' definito da diversi metodi, come movingBehaviour, loseLife, canMoveTo, checkCollision. Nel caso di EnemyModel1 e EnemyModel2, i metodi differenti sono movingBehaviour e loseLife. Implementare uno strategy pattern significherebbe creare almeno due interfacce: il comportamento di movimento (per movingBehaviour), il comportamento quando si viene colpiti da una bomba (per loseLife); con la necessità di creare altre classi con l'implementazione di questi comportamenti. Si tratterebbe di gestire almeno 6 nuove file (due interfacce, per ognuna due classi con le strategie) per solo due nemici, per cui attualmente

basta modificare 3 metodi in totale. Lo strategy pattern e' svantaggioso in questo caso, ma deve essere tenuto in considerazione se si vuole ampliare la quantità di nemici. Lo strategy pattern è usato per i Power-ups.

La classe PlayerModel estende la classe EntityModel, aggiungendo nuovi campi specifici per il player (come la potenza del raggio delle bombe, il numero di bombe che possono essere lanciate, ecc…) e aggiungendo metodi specifici per il cambiamento di questo campi (quando si passa sui powerups, ad esempio).

La grafica delle EntityModel e' gestita grazie al pattern MVC, quindi e' totalmente gestita dalla classe EntityView. Questa classe carica gli sprites che corrispondono al modello e anima i movimenti a seconda del numero di frame negli sprites. Per i due nemici differenti, vengono usati sprites diversi e un numero di frames diverso negli sprites.

Per la gestione delle bombe, vengono usate le classi:

- BombModel: Gestisce lo stato e il comportamento delle bombe piazzate dai giocatori, inclusa la loro detonazione e tiene traccia delle posizioni detonate
- BombController: Coordina le azioni delle bombe all'interno del gioco, come il piazzamento e la detonazione.

BombModel e' una classe che estende EmptyTile, quindi viene posizionata ~~in~~ nella mappa di gioco dallo StageModel. Lo StageModel si occupa anche di resettare le Tile detonate dalla bomba.

Per la gestione del punteggio e delle vite:

Tutte le EntityModel hanno una vita inziale quando vengono inizializzate. Nel caso dei nemici, la vita iniziale e' nell'ordine delle centinaia (100 per il nemico uno, 200 per il nemico due), mentre il PlayerModel e' inizializzato con solo 3 vite. Il PlayerModel ha anche un~~a~~ campo specifico per il punteggio (score). Quando lo StageModel identifica che e' stata detonata una posizione occupata da un nemico, toglie vita al nemico (di default 100 punti) e li assegna allo score del player. Se la posizione detonata e' occupata dal player, il player perde una vita. Il player puo' perdere una vita anche se cammina su un blocco occupato da un nemico. Le vite e il punteggio possono essere visualizzate nell'Head Up Display (classe HUDview).

Per la gestione dei powerups, vengono usate:

- la classe SpecialTile: identifica un blocco vuoto speciale. Questa classe deve essere definita con un tipo, che puo' essere un power up della casella per accedere al livello successivo.
- l'enum SpecialTileType: questa enumerazione definisce i tipi di SpecialTile, che possono essere un power up o il blocco per accedere al livello successivo. Contiene anche un metodo per generare un tipo casuale di power up.
- lo StageModel: quando un blocco viene distrutto, lo StageModel con probabilita' dell'80% genera un powerup (di tipo random) nella posizione distrutta.

I powerup possono essere distrutti dalle bombe, e rimossi quando un nemico ci passa sopra. Se il player passa sopra un powerup, il powerup viene attivato e il player aumenta di un caratteristica (a seconda del powerup). I powerup implementati sono:

- PowerUpBomb: aumenta il numero di bombe che il player puo' piazzare
- PowerUpBlast: aumenta la potenza del raggio delle bombe
- PowerUpSpeed: aumenta la velocita' di movimento del player

Una volta che il powerup viene usato, si comporta come una EmptyTile (a meno che non venga colpito da una bomba, in quel caso viene rimosso).

e-d.3) Utilizzo di design pattern

Model-View-Controller (MVC)

Il pattern MVC è ampiamente utilizzato per separare la logica del gioco (le classi Model) dalla logica per il display del gioco (le classi View), con un controller che funge da intermediario tra i due e da intermediario con gli input da tastiera che provengono dal giocatore. Nel progetto, il pattern MVC e' implementato usando le classi:

- Model: Le classi come PlayerModel, EnemyModel, BombModel e StageModel contengono la logica di gioco, gli stati e le regole senza preoccuparsi della presentazione o dell'interazione con l'utente.
- View: Le classi come EntityView, BombView e StageView sono responsabili della rappresentazione visiva del gioco, gestendo la renderizzazione e le animazioni.
- Controller: Classi come BombController, PlayerController, GameApp gestiscono l'input dell'utente e aggiornano lo stato dei Model. La classe EnemiesController svolge l'azione di controller e gestendo la creazione e l'aggiornamento degli EnemyModel, oltre a creare Views per ogni EnemyModel, ma senza gestire input dall'utente.

Questa separazione consente di modificare e estendere la logica di gioco o la presentazione senza influire sull'altra parte, facilitando anche il testing delle componenti in isolamento.

Tutte le classi Model sono Observable e le classi View sono Observer, in modo che le view possano essere aggiornate quando i modelli cambiano di stato.

Una classe particolare e' il caso del Player, che ha a disposizione una classe Model (Observable), Controller (gestisce gli input), View (Observer del Model) and Sound (Observer del Model), estendendo il pattern MVC anche al livello audio.

Observer/Observable Pattern

Il pattern Observer è utilizzato per creare un sistema di notifica dove gli oggetti si registrano per ricevere aggiornamenti sul cambiamento dello stato di un oggetto. Il design pattern di solito e' implementato tramite interfacce che definiscono i metodi per registrare, rimuovere e notificare gli osservatori. Le classiche interfacce java.util.Observer e java.util.Observable sono state deprecate, quindi ho implementato un sistema di notifica personalizzato basato sulle classi di JavaFX. JavaFX fornisce un'interfaccia per gli Observable (java.fx.beans.Observable) e un'interfaccia per gli Observer (java.fx.beans.InvalidationListener) che possono essere estese per creare un sistema di notifica personalizzato.[ba4] L'interfaccia Observable richiede l'implementazione dei metodi addListener e removeListener per aggiungere oggetti alla lista degli Observer e rimuoverli. A questi metodi ho aggiunto il metodo notifyListeners per notificare tutti gli osservatori registrati

quando si verificano cambiamenti di stato. L'interfaccia InvalidationListener richiede l'implementazione del metodo invalidated, che viene chiamato quando l'oggetto osservato e' invalido.

Oltre alle interfacce di JavaFX, ho implementato un sistema di notifica personalizzato per le classi Model e View. In particolare, ho creato un'interfaccia Observer che estende InvalidationListener e richiede l'implementazione di un metodo update() per aggiornare lo stato dell'Observer quando gli Observable notificano un cambiamento di stato. Questo sistema di notifica personalizzato e' stato utilizzato per gestire le interazioni tra i modelli e le viste:

- Observable: BombModel e EntityModel (in particolare PlayerModel) agiscono come soggetti osservabili, notificando gli osservatori registrati quando si verificano cambiamenti di stato.
- Observer: Interfacce e classi che implementano BombObserver (implementata da BombView), EntityStateObserver (implementata da EntityView), e StageObserver (implementata da StageView) ricevono aggiornamenti dagli osservabili a cui sono registrati, permettendo di aggiornare la vista o modificare altri modelli in risposta agli eventi.

Tutte le classi Model estendono Observable e le classi View implementano Observer, in modo che le viste possono essere aggiornate quando i modelli cambiano di stato.

In particolare, la classe PlayerModel fa largo uso del design pattern Observer Observable aggiornando anche la classe HUDView e la classe PlayerSound quando cambia il suo stato. La classe HUDView aggiorna le label per mostrare le vite e il punteggio del giocatore, mentre la classe PlayerSound riproduce i suoni quando il giocatore perde una vita o raggiunge un powerup (vedi punto d.6).[ba5]

E' importante notare che javafx integra gia' un sistema di notifica basato su Observable e Observer tramite classi native JavaFX come javafx.beans.property.SimpleIntegerProperty e javafx.beans.property.SimpleObjectProperty. Queste classi possono essere utilizzate per creare un sistema di notifica personalizzato, ma ho preferito implementare il classico Design Pattern per avere un maggiore controllo e flessibilità sulle notifiche e gli aggiornamenti. L'unico caso in cui ho usato le classi native di JavaFX e' stato per la gestione del tempo del livello, che e' un IntegerProperty aggiornato nella classe GameApp e a cui e' connesso un listener per aggiornare la label nella classe HUDView.

Strategy Pattern

Il programma utilizza uno Strategy Pattern per gestire i PowerUps. Questo pattern permette di definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. In questo caso, i PowerUps sono implementati come classi separate che estendono un'interfaccia comune PowerUpBehaviour. Questa interfaccia definisce un metodo apply che modifica le caratteristiche del PlayerModel quando il PowerUp viene raccolto. Le classi PowerUpBomb, PowerUpBlast e PowerUpSpeed implementano l'interfaccia PowerUpBehaviour e forniscono l'implementazione specifica per ciascun PowerUp. Quando il PlayerModel occupa un blocco di tipo PowerUp, viene applicato il metodo applyPowerUp corrispondente per modificare le caratteristiche del PlayerModel. Questo approccio consente di aggiungere facilmente nuovi PowerUps estendendo l'interfaccia PowerUpBehaviour e creando una nuova classe

PowerUp che implementa l'interfaccia. Inoltre, il pattern Strategy consente di separare l'implementazione dei PowerUps dalla logica del gioco, facilitando la manutenzione e l'estensione del codice.

Una menzione dovrebbe essere fatta per la classe EnemyModel. Come discusso nel punto 2, i nemici sono implementati con comportamenti diversi (EnemyModel1 e EnemyModel2) senza l'utilizzo dello Strategy Pattern. Questa scelta è stata fatta per mantenere il codice più semplice e leggibile, poiché i comportamenti dei nemici sono abbastanza semplici e richiedono solo poche modifiche nei metodi di movimento e aggiornamento. Tuttavia, se si prevede di aggiungere ulteriori tipi di nemici con comportamenti più complessi, sarebbe consigliabile implementare uno Strategy Pattern per gestire i diversi comportamenti dei nemici in modo più flessibile e modulare.

d.4) Adozione di Java Swing [2] o JavaFX [3] per la GUI

Java Swing e JavaFx sono due framework per la creazione di interfacce grafiche in Java. Java Swing e' piu' vecchio e piu' consolidato, mentre JavaFX e' piu' moderno e offre funzionalita' avanzate per la creazione di interfacce utente. Entrambi i framework sono supportati da Oracle e possono essere utilizzati per creare applicazioni desktop con GUI in Java. Per un progetto come JBomberMan, entrambe le librerie sono ottime candidate, per cui la scelta della libreria nel mio caso e' stata fatta in modo personale. Le considerazioni che mi hanno portato a scegliere JavaFX rispetto aJavaSwing sono:

- JavaFX offre funzionalità avanzate per la creazione di interfacce utente, come il supporto per animazioni, effetti grafici, e rendering hardware accelerato. Queste funzionalità sono utili per creare un'interfaccia utente interattiva e coinvolgente per un gioco come JBomberMan e semplifica la creazione di elementi grafici complessi.
- JavaFX è più moderno e supporta lo sviluppo di applicazioni desktop con un aspetto e un comportamento più simile alle applicazioni moderne (almeno questo è quello che ho letto cercando informazioni su internet). Molte volte javafx viene descritto come il successore di Swing, dato che e' piu' moderno e offre funzionalita' piu' avanzate per la creazione di interfacce utente.
- JavaFX integra in modo nativo il supporto per CSS. Questo e' particolarmente utile per la creazione di interfacce utente personalizzate in modo semplice e abbastanza immediato (CSS è usato per personalizzare i colori e i font della GUI).
- JavaFX integra nativamente design pattern come MVC e Observer/Observable, che semplificano la creazione di applicazioni. In questo progetto ho preferito implementare il classico design pattern Observer/Observable, ma JavaFX offre un supporto nativo per questo pattern tramite classi come javafx.beans.property.SimpleIntegerProperty che supportano facilmente la notifica e il binding tra proprieta' diverse. In una prima versione del codice ho utilizzato queste classi, ma ho preferito implementare il classico design pattern per avere un maggiore controllo e flessibilità sulle notifiche e gli aggiornamenti. Nella classe HUD View c'è del codice commentato che mostra come sarebbe stato possibile utilizzare le classi native di JavaFX per settare le label tramite binding con le proprietà del PlayerModel.

- JavaFx offre funzionalità comprensive per lo sviluppo di un'app, come la produzione di suoni, animazioni e musica di sottofondo (vedi punto 6).

Queste considerazioni mi hanno portato a scegliere di usare JavaFx per la creazione dell'interfaccia grafica del gioco JBomberMan (e devo dire che non mi sono pentito).

f-d.5) Utilizzo appropriato di stream

Gli stream sono una funzionalità introdotta in Java 8 per semplificare il lavoro con le collezioni e i dati. Gli stream permettono di eseguire operazioni di trasformazione, filtraggio e aggregazione sui dati in modo dichiarativo e funzionale. Il modo dichiarativo e' particolarmente utile perché rende il codice molto leggibile e conciso. Nel progetto, ho usato gli stream in diversi punti per eseguire operazioni su collezioni di dati. Alcuni esempi di utilizzo degli stream nel progetto sono:

- Nella classe Player Data, gli stream vengono utilizzati per leggere e scrivere i dati degli utenti da e su file. Questo e' particolarmente utile per leggere e scrivere i dati in modo semplice ed efficiente
- Nella classe EnemiesController, gli stream vengono utilizzati per filtrare i nemici vivi e morti e per eseguire operazioni su di essi. Ad esempio, gli stream vengono utilizzati per filtrare i nemici vivi, mappare i nemici in oggetti EnemyModel e creare una lista di nemici vivi per il controller.
- Nella classe EntityView, gli stream vengono utilizzati per caricare gli sprite delle entita' da file e per animare i movimenti delle entita'.

d.6) riproduzione di audio

La riproduzione audio è una funzionalità importante per creare un'esperienza di gioco coinvolgente. Nel progetto, ho utilizzato la classe AudioClip di JavaFX per riprodurre effetti sonori e musica di sottofondo. La classe AudioClip fornisce un'interfaccia semplice per caricare e riprodurre file audio corti. In particolare, ho utilizzato la classe AudioClip per riprodurre i seguenti suoni: piazzamento di una bomba; esplosione di una bomba; perdita di una vita; vittoria di una partita; sconfitta di una partita; raccolta di un powerup; camminata del bomberman. Tutti i suoni vengono prodotti in modo statico della classe AudioUtils. In particolare, ho implementato la classe PlayerSound come Observer della classe PlayerModel che si occupa di produrre tutte le clip relative al player.

Oltre alla riproduzione di audio corti per effetti sonori, ho utilizzato la classe MediaPlayer di JavaFX per riprodurre musica di sottofondo. La classe MediaPlayer fornisce funzionalita' avanzate per la riproduzione di file audio piu' lunghi, inclusa la possibilita' di controllare la riproduzione, la pausa e il volume. La classe GameApp si occupa di inizializzare e riprodurre la musica di sottofondo quando la partita viene inizializzata.

d.7) animazioni ed effetti speciali

Nel progetto, ho utilizzato JavaFX per creare animazioni e effetti speciali per le entita' del gioco. In particolare, ho utilizzato le classi Timeline, AnimationTimer, e KeyFrame e utilizzando gli sprites delle entita' a diversi frame.

In particolare:

- Per le diverse entita' in gioco, ho organizzato gli sprites in un'immagine unica e ho utilizzato le classi ImageView e SpriteAnimation per animare i movimenti delle entita'. La classe SpriteAnimation si occupa di caricare gli sprites da un'immagine e di animare i movimenti delle entita' a diversi frame. Questo approccio consente di creare animazioni fluide e realistiche per le entita' del gioco. Ogni entita' ha un'animazione specifica con uno sprite sheet dedicato e un numero di frame dedicato.
- Per l'animazione delle bombe, per simulare la "pulsazione", ho utilizzato una Timeline che cambia lo sprite [ba7] della bomba in modo ciclico (ho tre frame nello sprite sheet). Questo crea un effetto visivo che simula l'espansione e la contrazione della bomba prima dell'esplosione.
- Per l'esplosione delle bombe, costruisco un'animazione che si espande in tutte le direzioni fino a raggiungere il raggio massimo. Questo e' fatto con una Timeline che aggiunge ImageView per ogni blocco dell'esplosione e le fa sparire dopo un certo tempo. L'animazione dell'esplosione richiede quindi una composizione di immagini che cambia nel tempo (la fiammata aumenta la larghezza). Per capire il raggio massimo in ogni direzione controllo le posizioni detonate dal BombModel.
- La creazione dell'immagine della mappa viene composta da un'immagine di sfondo e da ImageView per ogni blocco della mappa. Cio' e' stato fatto per permettere di cambiare facilmente la mappa di gioco e per permettere di cambiare i blocchi distruttibili, non distruttibili, powerups, ecc... senza dover cambiare il codice. Questo e' particolarmente utile per creare nuovi livelli di gioco in modo semplice e veloce. Per comporre l'immagine ho usato la classe javafx.scene.image.PixelReader e javafx.scene.image.PixelWriter di JavaFX.

g) altre note progettuali e di sviluppo

Sviluppo del Progetto

Ho sviluppato il progetto usando Maven per la gestione delle dipendenze e la compilazione del progetto, usando Visual Studio Code come IDE. Maven semplifica la gestione delle dipendenze e la compilazione del codice. In particolare, ho utilizzato Maven per gestire le dipendenze di JavaFX, visto che JavaFX non e' piu' incluso nel JDK a partire dalla versione 11. Maven mi ha permesso di usare plugins per generare un file Jar, la documentazione con Javadoc e abbozzare la struttura del progetto con PlantUML. PlantUML mi ha aiutato a creare diagrammi UML in modo dichiarativo, il che e' molto utile visto che il progetto e' abbastanza complesso per disegnare le classi UML a mano.

Controllo delle collisioni

Per il controllo delle collisioni del gioco, uso un approccio basato sulla presenza delle entita' in un blocco (tile). Ogni entita' ha una posizione (x, y), una bounding box (x, y) e una bounding offset (x, y). La bounding box rappresenta l'area occupata dall'entita' e viene posizionata in un blocco nella mappa di gioco. Per gestire le collisioni con i blocchi, controllo che la bounding box dell'entita' non intersechi un blocco della mappa. Un'entita' puo' muoversi solo su EmptyTile o sottoclassi. La bounding offset puo' correggere la bounding

box se il modello del giocatore non e' centrato nei blocchi della mappa. Questo e' particolarmente utile per evitare collisioni quando il giocatore si muove tra i blocchi.

Quando un'entita' si muove, occupa tutte le caselle libere raggiunte dalla bounding box. Per questo, per controllare la collisione tra due entita', posso controllare semplicemente se l'entita' sta cercando di muoversi in un blocco occupato da un'altra entita'. Questo approccio semplifica il controllo delle collisioni, ma per entita' piu' complesse potrebbe essere necessario un approccio piu' sofisticato.

Istruzioni per giocare

Usare le frecce per muoversi e la barra spaziatrice per piazzare bombe.