

Zombie Tsunami
Relazione per il progetto
del corso di
Programmazione ad Oggetti
A.A. 2023/24

Luca Trianti
Lorenzo Tordi
Alex Frisoni,
Lukasz Wojnicz

18 febbraio 2024

Indice

1 Analisi	2
1.1 Requisiti	2
1.2 Analisi e modello del dominio	3
2 Design	5
2.1 Architettura	5
2.2 Design dettagliato	7
2.2.1 Luca Trianti	7
2.2.2 Alex Frisoni	15
2.2.3 Lorenzo Tordi	24
2.2.4 Lukasz Wojnicz	33
3 Sviluppo	39
3.1 Testing automatizzato	39
3.2 Note di sviluppo	40
3.2.1 Luca Trianti	40
3.2.2 Alex Frisoni	41
3.2.3 Lorenzo Tordi	41
3.2.4 Lukasz Wojnicz	42
4 Commenti finali	43
4.1 Autovalutazione e lavori futuri	43
4.1.1 Luca Trianti	43
4.1.2 Alex Frisoni	44
4.1.3 Lorenzo Tordi	44
4.1.4 Lukasz Wojnicz	44
4.2 Difficoltà incontrate e commenti per i docenti	45
4.2.1 Alex Frisoni	45
A Guida utente	46

Capitolo 1

Analisi

Lo scopo di questo progetto è la realizzazione di una versione semplificata in linguaggio Java del gioco per smartphone "Zombie Tsunami". La partita consiste in una corsa in cui il giocatore controlla uno zombie in un percorso della mappa affrontando ostacoli. L'obiettivo è completare il livello.

1.1 Requisiti

Requisiti funzionali:

- Il gioco prevede la scelta di un livello giocabile: all'avvio della partita, il giocatore controllerà uno e un solo zombie che, nell'attraversare la mappa di gioco, incontrerà ostacoli e civili.
- Nella partita di gioco, i civili rappresentano potenziali prede, che incrementeranno la forza dello zombie.
- Gli ostacoli che si presenteranno saranno di due tipi: bombe e oggetti breakable. Questi ultimi avranno un livello minimo di forza richiesta da parte dello zombie controllato, il quale scontrandosi, potrà romperlo per farsi facilmente strada, altrimenti se non viene saltato, porterà al game over.
- Le bombe, diversamente, se colpite arrecheranno un malus allo zombie, riducendone la forza di 1.
- Al raggiungimento della fine del livello, segnalato da una bandierina, la mappa sarà considerata conquistata.

Requisiti non funzionali:

- Il programma deve essere in grado di mettere in pausa la partita.
- Il gioco avrà una sezione nella schermata iniziale dove verranno spiegate le regole.
- Al game over oppure alla vittoria, il giocatore visualizzerà il relativo esito su schermo, dopo il quale il gioco verrà chiuso automaticamente con un'attesa di circa tre secondi.

1.2 Analisi e modello del dominio

Il modello del dominio dell'applicazione corrisponde al concetto di partita, la quale incapsula tutto ciò che concerne la logica di gioco. La parte centrale nel contesto della partita è il livello giocabile, nel quale avverranno le interazioni fra tutti gli altri elementi del dominio(zombie, persone e ostacoli). La partita (rappresentata dal `Model`) si occupa della gestione del gioco e delle collisioni delle entità.

Attraverso quest'ultima , viene gestita, grazie a `GameMap`, la mappa giocabile, il movimento, la logica della camera di gioco, e la corretta associazione tra vari `Tile` e il loro valore numerico nella mappa. La rappresentazione degli ostacoli, dei cittadini e dello zombie viene gestita da `ObstacleManager`, `PersonManager` e `Zombie`, che assegnano ai vari elementi una posizione nella mappa, mentre le loro rispettive iterazioni nella partita vengono gestite da `Collision`.

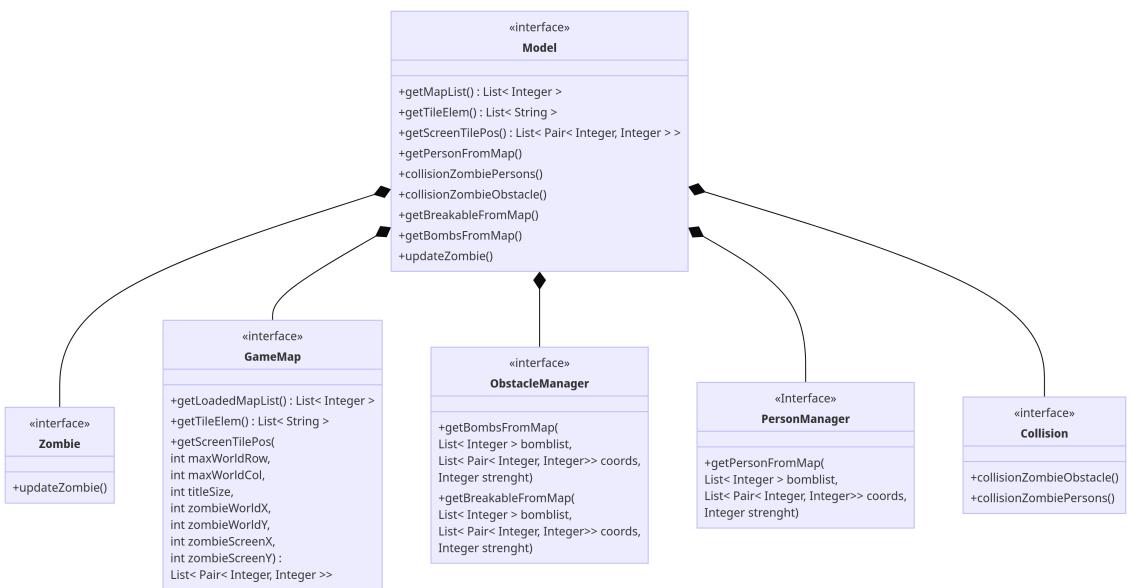


Figura 1.1: Schema UML del modello del dominio.

Capitolo 2

Design

2.1 Architettura

Il pattern architettonico che è stato usato è quello dell'MVC, il quale ha permesso di separare la parte logica del gioco da quella visualizzata dall'utente, portando così a una più facile gestione degli elementi (soprattutto in caso di modifiche) e a una maggiore pulizia e organizzazione del codice.

L'interfaccia **Model** racchiude in se tutta la logica del gioco, e quindi del dominio, separandola dal resto delle componenti del gioco (gestendo anche la possibilità di game over e di vincita).

Il **Controller** si occupa di far comunicare correttamente il Model con la View, aggiornando assiduamente le informazioni dal Model e rendendole disponibili all'utente. Un cambiamento come lo spostamento della camera di gioco, e quindi di tutti gli elementi presenti in mappa, devono essere perennemente riflessi nella View, quindi passando le informazioni in continuazione senza una richiesta precisa. Anche il controllo per l'eventualità delle varie collisioni provoca cambiamenti agli elementi presi in causa, modificando ancora una volta le loro caratteristiche nel Model. Il Controller infine ha anche il compito di comunicare correttamente all'utente se si è in una situazione di game over oppure di vincita.

La **View** è rappresentata principalmente da un suo personale controller chiamato **VController**, dalla classe **View** che gestisce l'avvio del programma, e infine dall'interfaccia **Map**, il motore grafico principale del gioco. Mentre il **VController** connette il Controller con il complesso della View, la classe **View**, grazie a **StartMenu**, offre l'interfaccia grafica iniziale, per poi avviare la scena di gioco rappresentata dalla **Map**. Quest'ultima è costantemente aggiornata per la rappresentazione progressiva dei vari elementi di gioco, dando vita alla logica del dominio e al livello giocabile principale.

In conclusione, la View si occupa di rendere visibili all’utente tutti gli elementi di gioco (zombie, ostacoli e persone) presenti sulla mappa.

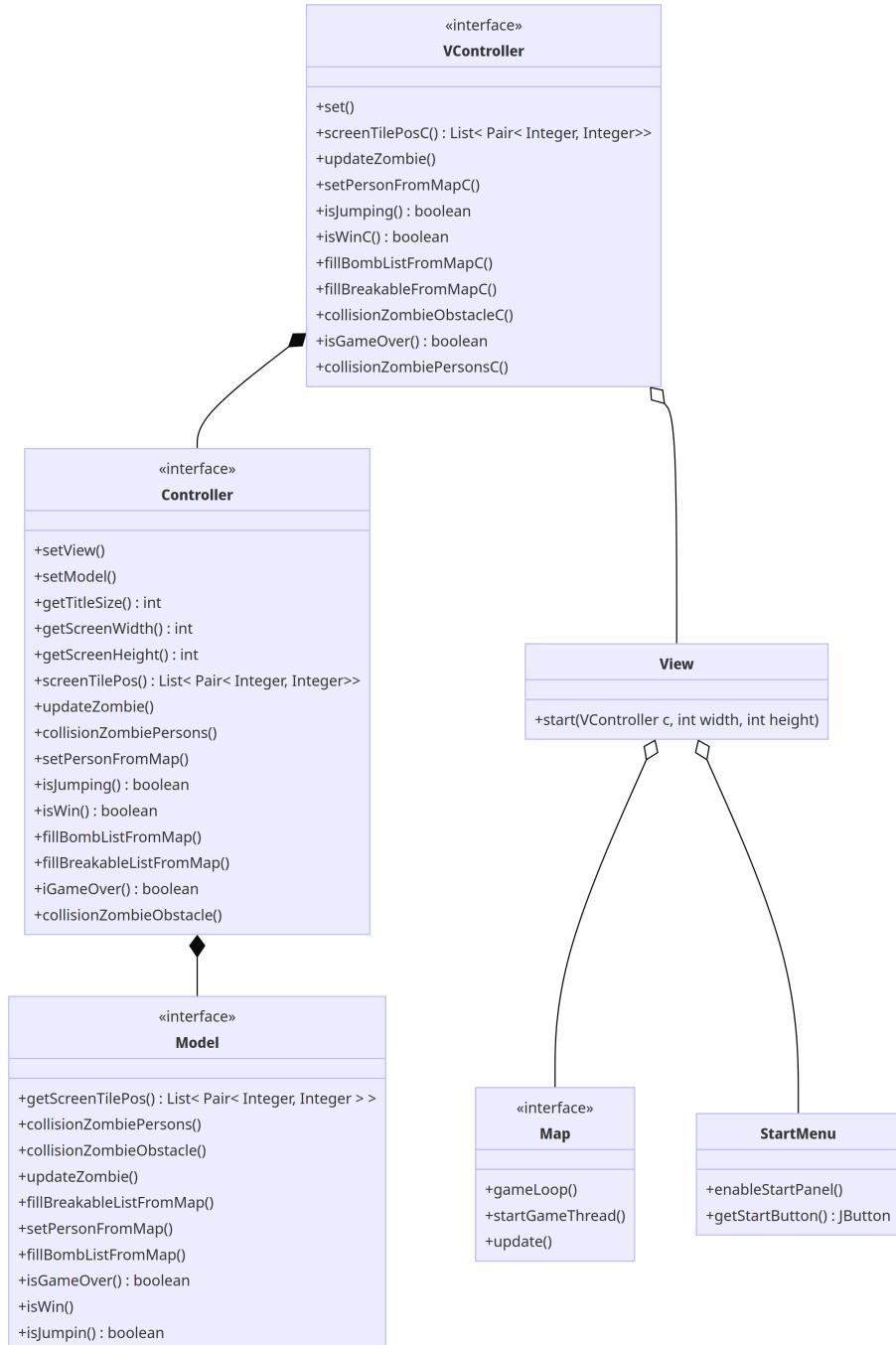


Figura 2.1: Schema UML dell’architettura dell’applicazione.

2.2 Design dettagliato

2.2.1 Luca Trianti

From

Gestione Ostacoli sulla mappa.

Per la realizzazione degli ostacoli, prima di tutto bisogna suddividerli in due categorie: *Bombe* e *Breakable* (ovvero, oggetti che si possono rompere).

- **Bombe:** Tipi di ostacoli che diminuiscono la forza dello zombie di 1 punto al contatto con esso, quindi in sostanza recano del danno. Nel codice la bomba è introdotta nella classe *Bomb*, implementata da *BombImpl*, la quale può permettere di impostare il danno della bomba con il metodo `setDamage(int damage)`, e recuperare quest'ultimo con il metodo `getDamage()`, il quale ritornerà appunto il danno della bomba sotto forma di *int*.

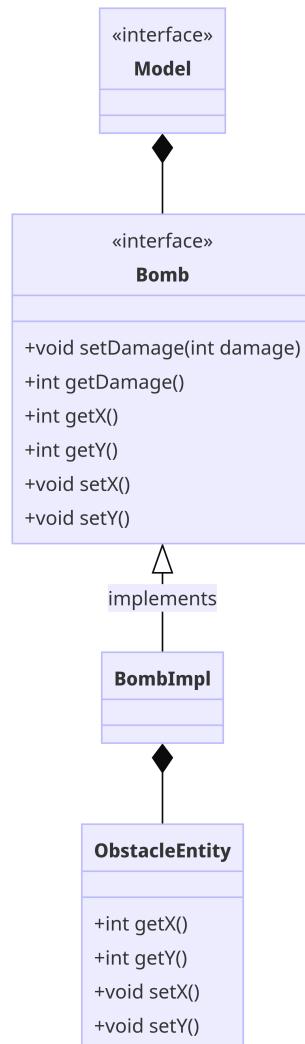


Figura 2.2: UML della logica dell’entità della bomba con i corrispettivi metodi.

- **Breakable:** Hanno una forza minima richiesta dallo zombie per essere distrutti, nel caso si abbia una forza non sufficiente, la partita verrà messa in uno stato di Game Over. Nel codice il breakable è introdotto nella classe *Breakable*, implementata da *BreakableImpl*, avente un costruttore con il parametro *minforce* al suo interno. La *minforce* si può ottenere grazie al metodo *getMinForce()* che ritorna la forza minima con la quale si può rompere il breakable, inoltre, la classe ha un metodo *canBreakObstacle(int zombieForce)* che ritorna *true* se la forza dello zombie è sufficiente per romperlo, altrimenti *false*.

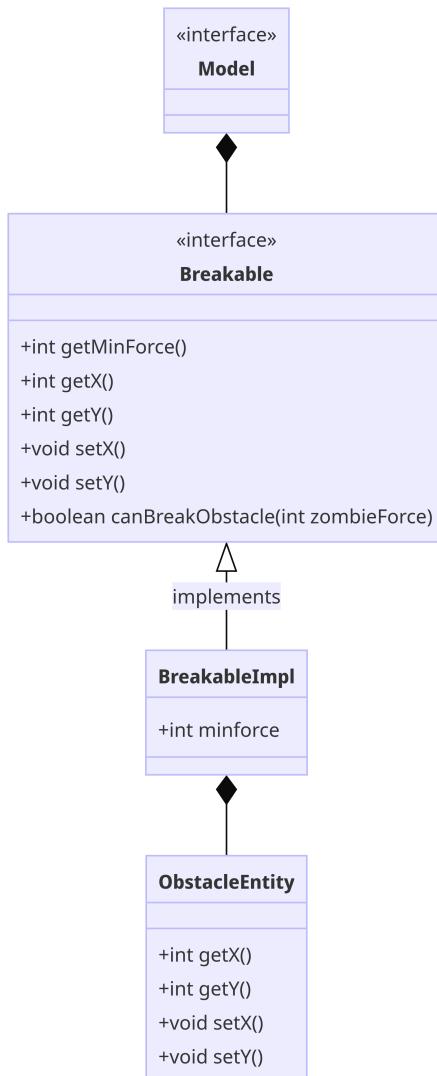


Figura 2.3: UML della logica dell’entità del breakable con i corrispettivi metodi.

Entrambi gli ostacoli, hanno un’entità al loro interno chiamata *ObstacleEntity*, la quale gestisce le coordinate *X* e *Y* di essi. Da tenere conto che, con *ObstacleEntity* vengono gestite SOLO le coordinate su schermo degli ostacoli, per cui, più lo zombie si avvicina ad essi, e più il valore delle coordinate degli ostacoli diminuiscono.

Per poter stampare a video gli ostacoli, abbiamo optato per una mappa testuale fatta di: 0, 1 e 2: 0 corrisponde ad una cella vuota, 1 ad una bomba e infine 2 ad un breakable. Nella classe *ObstacleManagerImpl*, che

implementa la corrispettiva *ObstacleManager*, abbiamo due liste: una di oggetti *Bomb* e un'altra di oggetti *Breakable* che verranno riempite grazie ai metodi: `fillBombListFromMap(...)` e `fillBreakableListFromMap(...)`, ciò consente di convertire gli 1 e 2 della mappa testuale nelle effettive bombe e breakable; questo avviene chiamando i due metodi nel metodo della view: `drawObstacleV(...)` della classe *DrawObstacleImpl* che implementa *DrawObstacle* però è qui che si verifica un problema.

Problema: Attualmente, il metodo `drawObstacleV(...)` viene invocato ad ogni frame. Questo approccio potrebbe generare problematiche nel caso in cui si aggiungano o rimuovano ostacoli dalle liste, con conseguente aggiornamento delle loro coordinate X e Y. Tale procedura può portare alla presenza di molteplici istanze degli stessi ostacoli nelle liste, ognuna con coordinate diverse.

Soluzione: Per risolvere questo problema, ho apportato delle modifiche. Ora, le liste di ostacoli vengono inizializzate ogni volta che i metodi `fillBombList FromMap(...)` e `fillBreakableListFromMap(...)` sono chiamati. In questo modo, vengono aggiunti alle liste solo gli ostacoli effettivamente presenti sullo schermo. Questa modifica elimina la possibilità di avere molteplici istanze degli stessi ostacoli nelle liste, garantendo la coerenza tra gli ostacoli visualizzati e quelli presenti nelle liste.

La classe *ObstacleManager* offre ulteriori funzionalità attraverso i metodi `getBombList()` e `getBreakableList()`, che consentono di ottenere le liste corrispondenti. Inoltre, è possibile aggiungere bombe e oggetti breakable alle liste mediante i metodi `addBomb()` e `addBreakable()`.

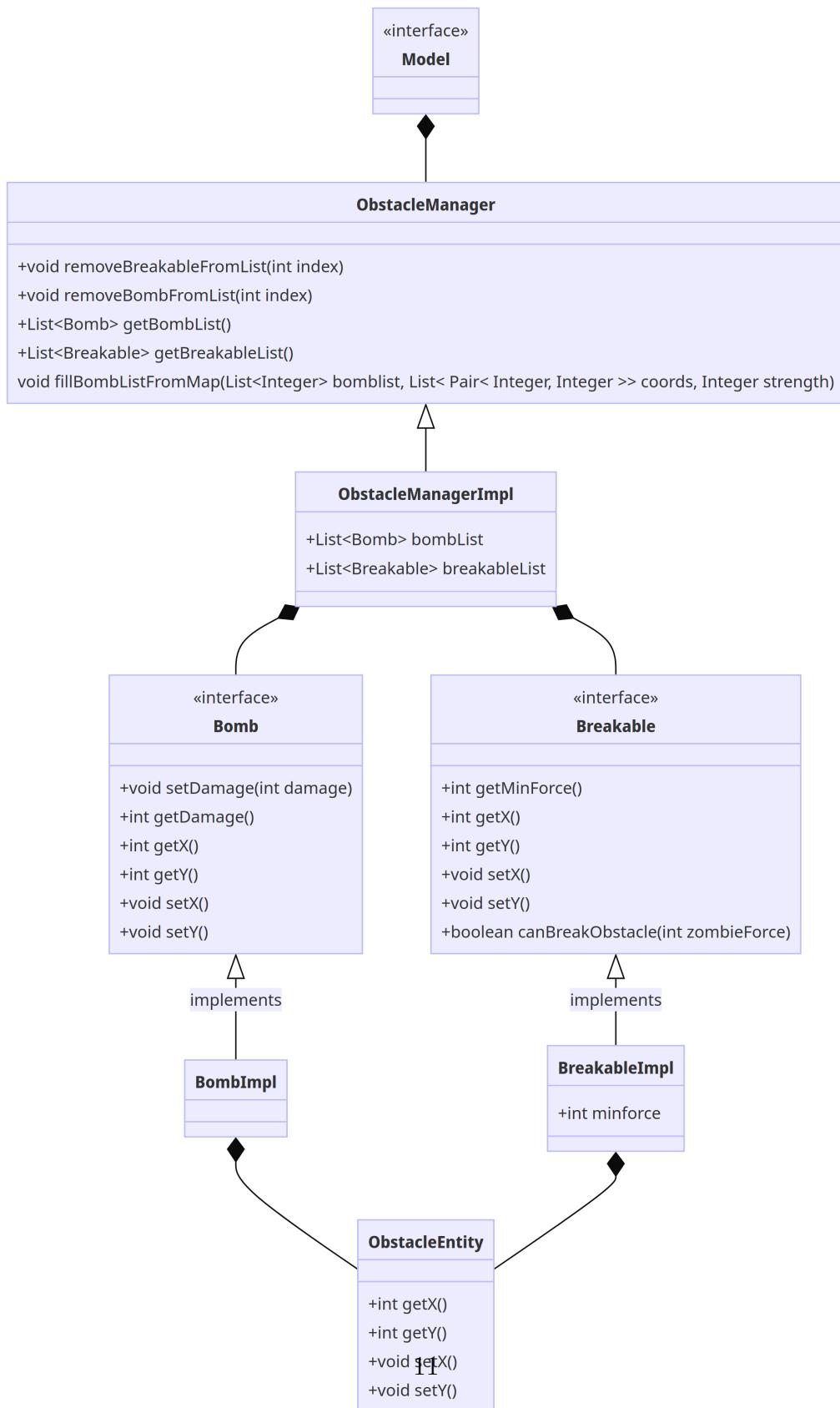


Figura 2.4: UML della logica dell'ObstacleManager

Per la rappresentazione grafica degli ostacoli sulla finestra di gioco, ho implementato la classe *DrawObstacleImpl*, che segue l’interfaccia *DrawObstacle*. Il metodo principale, `drawObstacleV(...)`, accetta una lista di interi contenenti i valori menzionati in precedenza (0, 1 e 2). Il metodo itera attraverso la lista, e se il valore è 1, invoca il metodo `fillBombListFromMap(...)` e visualizza la bomba a schermo; se il valore è 2, richiama `fillBreakableListFromMap(...)` e mostra il breakable a schermo. C’è da tenere in considerazione che questo metodo viene invocato ogni frame (quindi 60 volte al secondo), e che la posizione dell’ostacolo viene aggiornata ad ogni invocazione, sia a video e sia nel codice. Per la visualizzazione a schermo degli ostacoli, ho implementato due metodi che restituiscono un’istanza di *BufferedImage*. Questa classe rappresenta un’immagine con un buffer accessibile di dati dell’immagine. Entrambi i metodi recuperano l’immagine associata a un percorso specifico. Nel caso in cui l’operazione di lettura del file causi un’eccezione di tipo *IOException*, questa viene gestita adeguatamente. Entrambi i metodi, ovvero: `getBomb()` e `getBreakable()` sono stati implementati dentro `drawObstacleV(...)`, così facendo, la loro immagine viene aggiornata ogni frame sullo schermo.

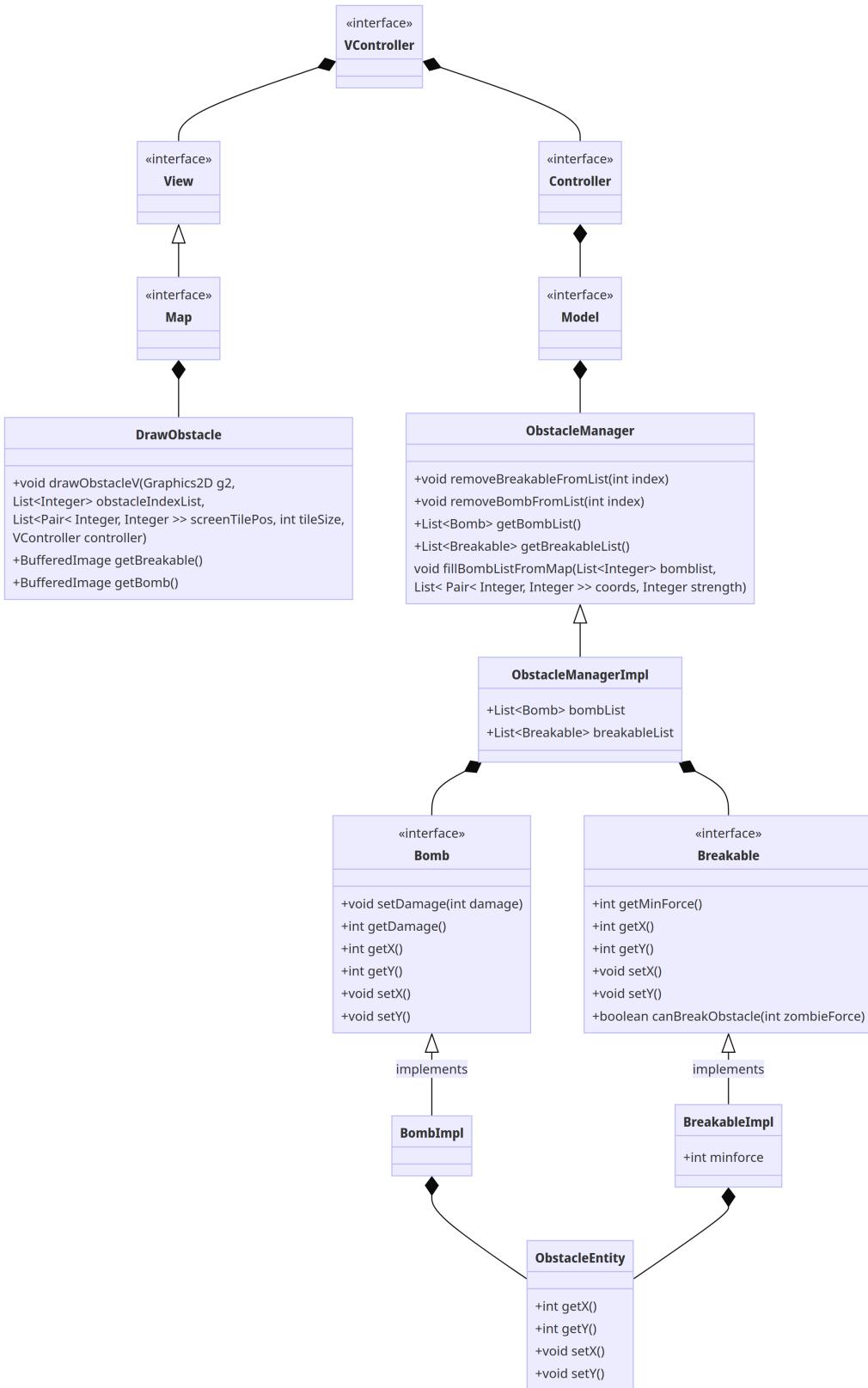


Figura 2.5: UML della logica della stampa a video degli ostacoli

Gestione collisioni tra Ostacoli e Zombie.

Per gestire le collisioni tra ostacoli e zombie, è necessario considerare il comportamento degli ostacoli sullo schermo. Mentre lo zombie attraversa la mappa, la coordinata X dell'ostacolo diminuisce progressivamente. Per affrontare questa situazione, ho implementato due threshold (soglie di coordinata X oltre le quali si verifica un'azione). Grazie a queste soglie, è possibile controllare se la coordinata X dell'ostacolo supera la soglia. In caso affermativo, si verifica anche se lo zombie si trova verticalmente all'interno dell'intervallo di altezza occupato dalla bomba nella lista.

La condizione specifica è che la coordinata Y dello zombie deve essere maggiore della posizione Y superiore della bomba ridotta di una dimensione di *tileSize* e minore della posizione Y inferiore della bomba aumentata di una dimensione di *tileSize*.

Una volta verificato ciò, se l'ostacolo è una bomba, si controlla se la forza dello zombie diminuita del danno della bomba è minore di zero. In caso affermativo, si mette il gioco in uno stato di Game Over; altrimenti, si decrementa la forza dello zombie e si rimuove la bomba sia graficamente sia logicamente dal gioco.

Nel caso in cui l'ostacolo sia un Breakable, si verifica se lo zombie abbia abbastanza forza per rompere l'ostacolo, invocando `canBreakObstacle(...)`. In caso affermativo, si rimuove il breakable sia graficamente sia logicamente dal gioco; in caso contrario, si passa allo stato di Game Over.

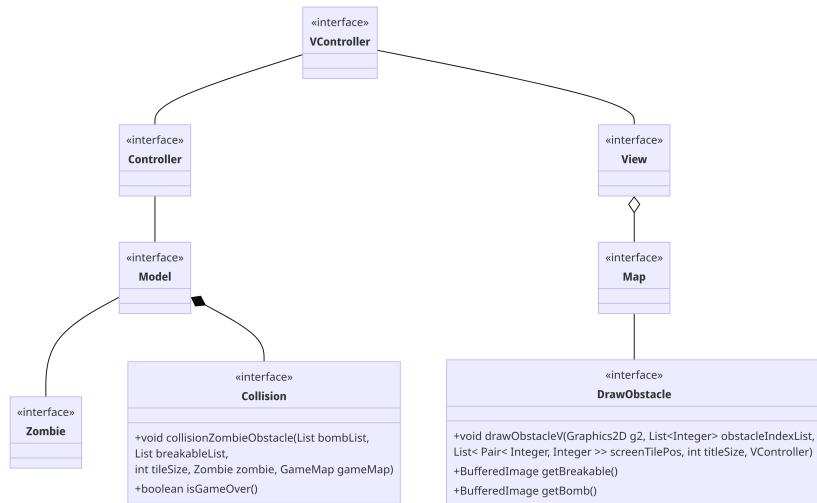


Figura 2.6: UML della logica delle collisioni

2.2.2 Alex Frisoni

Gestione delle coordinate dello zombie

Problema: Nel contesto della gestione delle coordinate dello zombie sulla mappa, il principale problema risiede nell'assicurare un comportamento corretto nell'aggiornamento delle coordinate.

Soluzione: Per garantire uno sviluppo scalabile e mantenere un codice organizzato, ho adottato l'approccio di creare separatamente una classe anzichè nella stessa dello zombie, rendendo il codice più modularizzato e facile da gestire:

- Per l'assegnazione delle coordinate e della forza dello zombie ho utilizzato la classe *Entity*, implementata da *EntityImpl*, che si occupa di gestire in modo pulito e organizzato tali attributi. Per assegnare le coordinate iniziali e la forza dello zombie utilizzo i seguenti metodi: `setX(...)`, `setY(...)` e `setSpeed(...)`, `setStrength(...)`. Successivamente per ottenere i valori correnti delle coordinate e della forza, utilizzo i metodi `getX()`, `getY()` e `getSpeed()`, `getStrength()`, `getScreenX()`, `getScreenY()` che vengono aggiornati periodicamente tramite il metodo `update()`. La Strength viene aumentata e diminuita tramite due suoi metodi appositi, ovvero, `increaseStrength()` e `decreaseStrength()`.

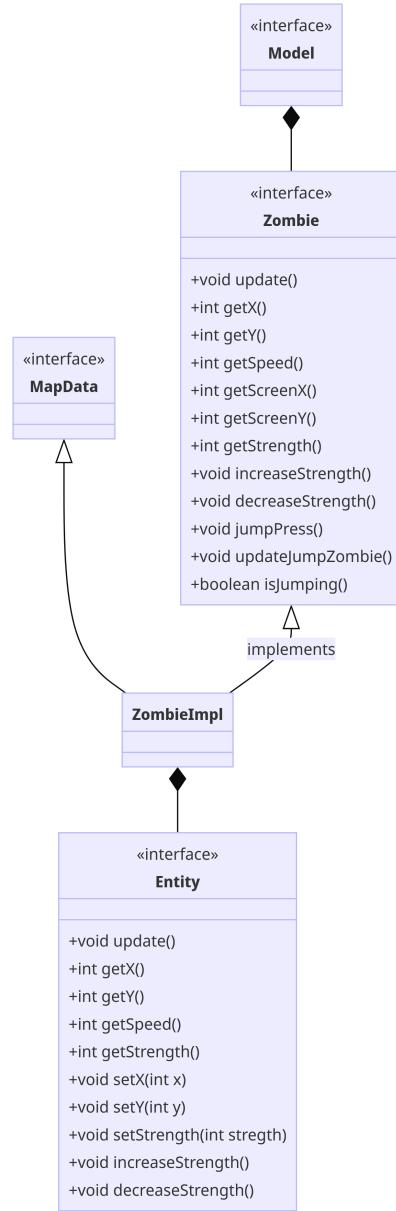


Figura 2.7: UML della logica dell'entità dello zombie con le sue coordinate

Gestione del salto dello zombie sulla mappa.

Problema: Il salto dello zombie deve essere graduale e con un comportamento fluido.

Soluzione: Ho scelto di creare a parte una classe dedicata alla logica del salto, per rendere il codice molto più semplice alla lettura e alla scalabilità.

- Per la gestione del comportamento di salto dello zombie, ho implementato la classe *JumpZombie*, implementata da *JumpZombieImpl*, dedicata esclusivamente a questa funzionalità, garantendo un’organizzazione chiara e una separazione dei compiti all’interno del codice. La gestione del salto avviene attraverso diversi metodi: `jumpPress()`, utilizzata per impostare le variabili iniziali e successivamente `updateJumpZombie()` che si occupa di far salire lo zombie fino a un’altezza preimpostata durante il salto, per poi riportarlo alla sua altezza iniziale una volta raggiunta desiderata. Inoltre, ho implementato un metodo che restituisce un valore booleano per indicare se il salto dello zombie è in corso o è già terminato.

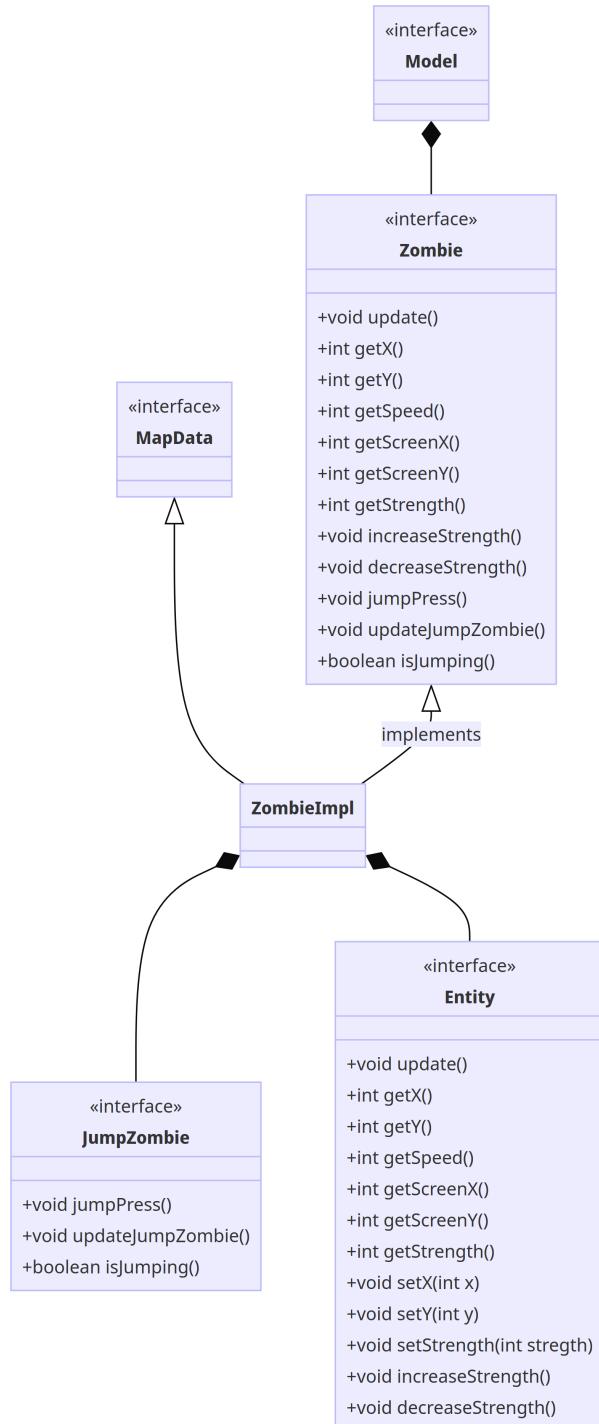


Figura 2.8: UML della logica del salto dello zombie

Gestione della classe zombie

Problema: Avendo più classi per la gestione dello zombie, ovvero, *Jump-ZombieImpl* e la *EntityImpl* può essere confusionario il richiamo dei metodi delle stesse dal model.

Soluzione: Ho deciso di creare quella "Manager", in modo tale che qualsiasi classe esterna che cerca di chiamare un metodo dello zombie debba passare da quest'ultima. In questo modo ho reso non visibile all'esterno le altre classi, facilitando le modifiche future.

- Inizializzo le variabili "screenX" e "screenY" con le dimensioni dello "Screen" e del "TitleSize" fornite dalla classe *MapData* e le altre coordinate dello zombie con i metodi della classe *Entity*: `setX(...)`, `setY(...)` e `setSpeed(...)`, `setStrength(...)`. Gli altri metodi che hanno bisogno di essere chiamati all'esterno li rendo visibili per il corretto funzionamento dello zombie.

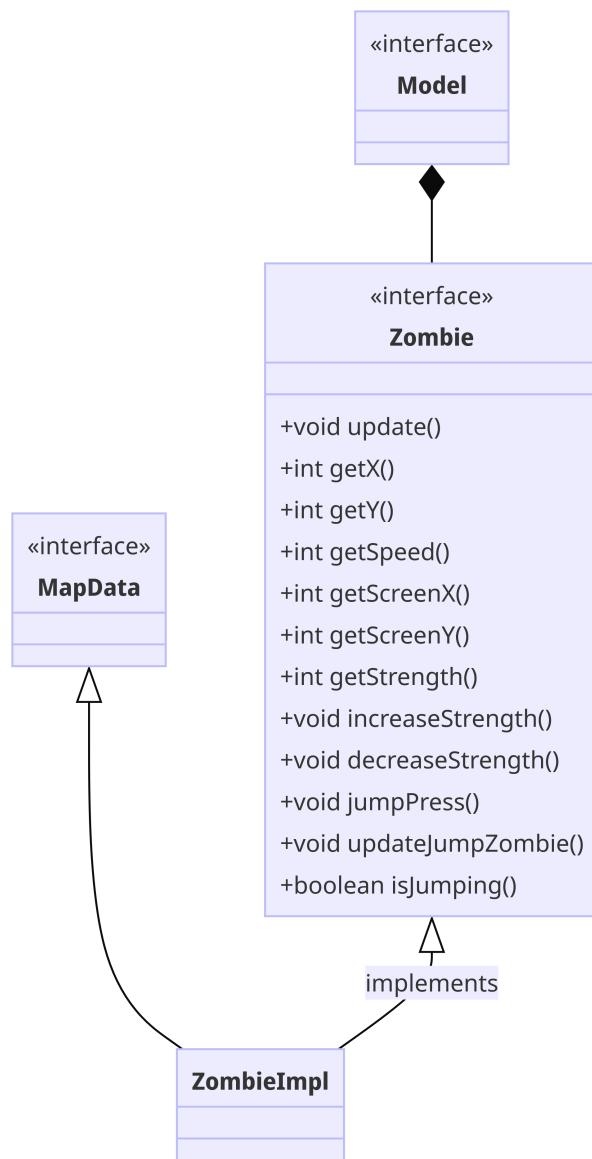


Figura 2.9: UML della classe dello zombie con i metodi che rende visibili

Gestione dell'input da tastiera

Problema: Prendere in input i tasti corretti e trovare la soluzione nel caso in cui ci sia una ripetuta pressione del tasto.

Soluzione: Ho utilizzato la classe KeyHandlerImpl con la quale ho implementato i metodi per quando viene premuto e rilasciato il tasto e con l'aggiunta di un metodo per capire se il tasto è premuto.

- Per gestire l'input da tastiera ho implementato la classe *KeyHandler* implementata da *KeyHandlerImpl* che si occupa di gestire gli eventi di input della tastiera all'interno del gioco. `keyPressed(KeyEvent e)` e `keyReleased(KeyEvent e)`: Questi metodi vengono chiamati quando un tasto viene premuto o rilasciato il tasto "SPACE". `keyTyped(KeyEvent e)`: Questo metodo non viene utilizzato nella nostra implementazione e non esegue alcuna azione. `isPressed()`: Questo metodo restituisce un valore booleano che indica se il tasto per il salto dello zombie è attualmente premuto sulla tastiera. `isOnPause()`: Questo metodo è stato implementato per mettere il gioco in pausa o meno tramite il tasto "ESC" della tastiera.

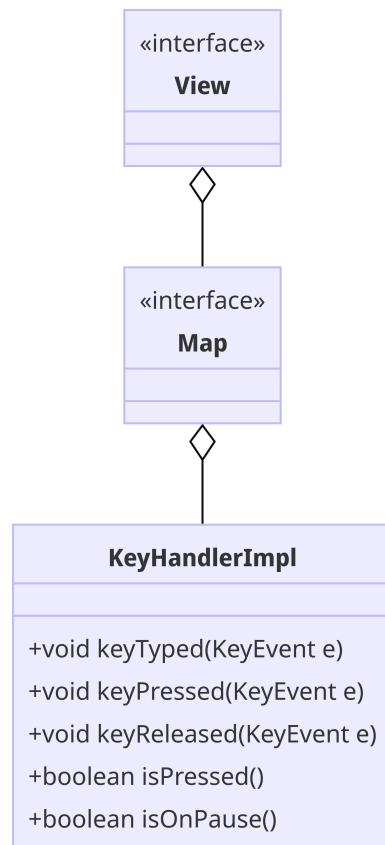


Figura 2.10: UML della classe che si occupa dell'input a tastiera

Gestione dello zombie sulla mappa.

Problema: Disegnare lo zombie sulla mappa alternando le immagini dello zombie per rendere il suo movimento fluido.

Soluzione: Per alternare in modo corretto lo zombie utilizzo un contatore che alterna l'immagine, rendendo il movimento scorrevole.

- Per disegnare lo zombie sulla mappa utilizzo la classe *DrawZombie* implementata da *DrawZombieImpl* col metodo `drawZombieV(...)`. Questo metodo prende in input un oggetto `Graphics2D` per il contesto grafico e un oggetto `VController` che fornisce informazioni relative al gioco. Utilizzando queste informazioni, il metodo disegna l'immagine dello zombie sulla mappa. La posizione dello zombie è determinata dalle coordinate fornite dal controller, e le dimensioni dell'immagine dello zombie sono adattate in base alla dimensione dei titoli nella mappa. L'immagine viene alternata tramite un contatore, se esso è maggiore di una costante(`FRAMESCHANGE`), allora viene azzerato e il successivo `update()` cambierà immagine. L'ultimo metodo della classe è `handleKeyPress(...)` che controlla tramite il `KeyHandler` passato come parametro se il tasto è premuto e se la variabile booleana è stata impostata a true, in quel caso inizializza le variabili nella classe *JumpZombie* e avvia il ciclo di salto.

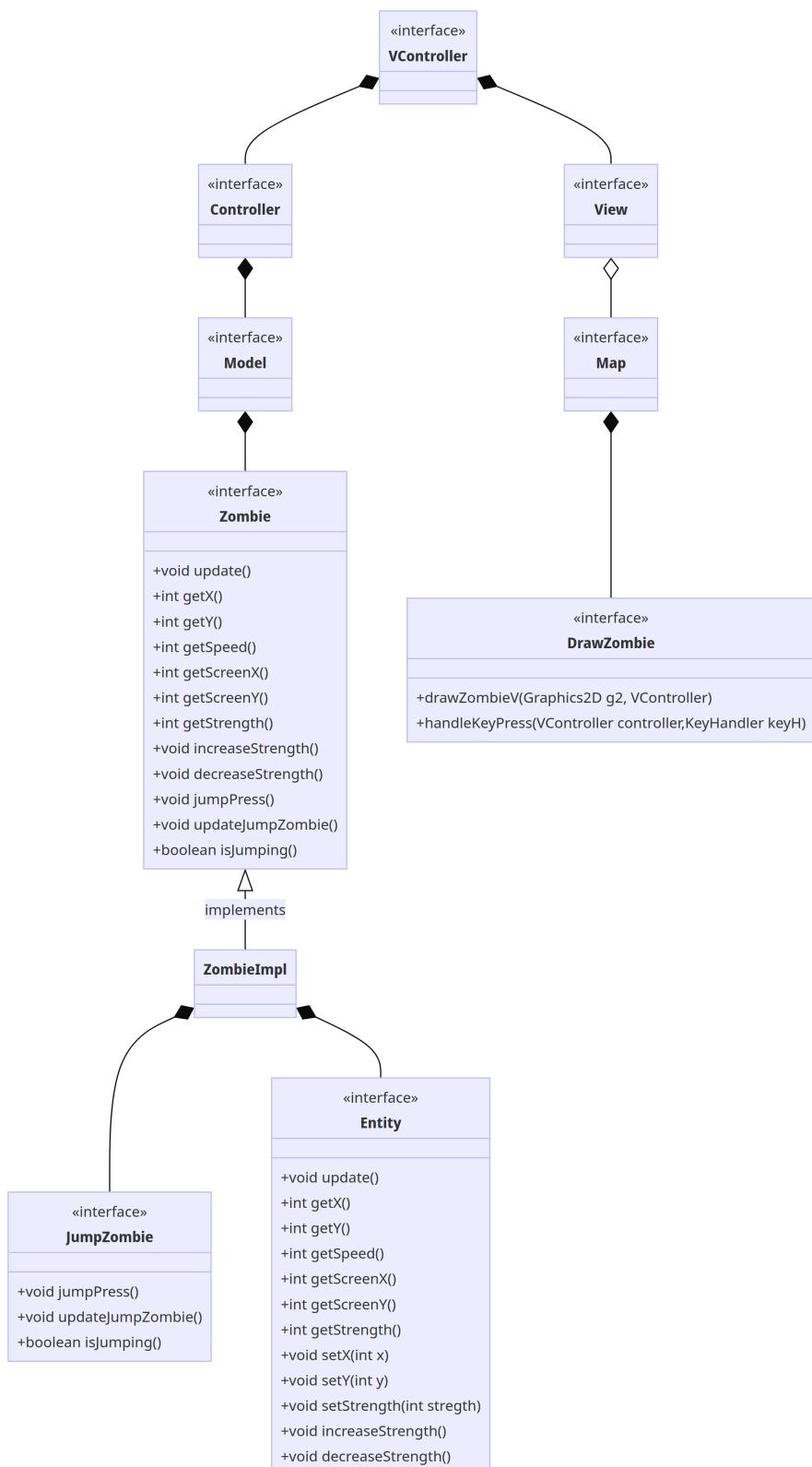


Figura 2.11: UML della classe per disegnare lo zombie sulla mappa

Gestione delle collisioni dello zombie sulla mappa

Problema: Confrontare le coordinate dello zombie con oggetti della mappa.

Soluzione: Utilizziamo un range di coordinate per confrontare se avviene la collisione o no.

- Per controllare se lo zombie collide con un oggetto sulla mappa vengono effettuati due controlli con i seguenti metodi:`collisionZombieObstacle(...)`, `collisionZombiePerson(...)`, presenti nella classe. I metodi controllano che la coordinata Y dello zombie fornite da `getScreenY()` sia nel range della bomba o della persona, nel caso affermativo, diminuisce il valore della sua forza o lo aumenta.

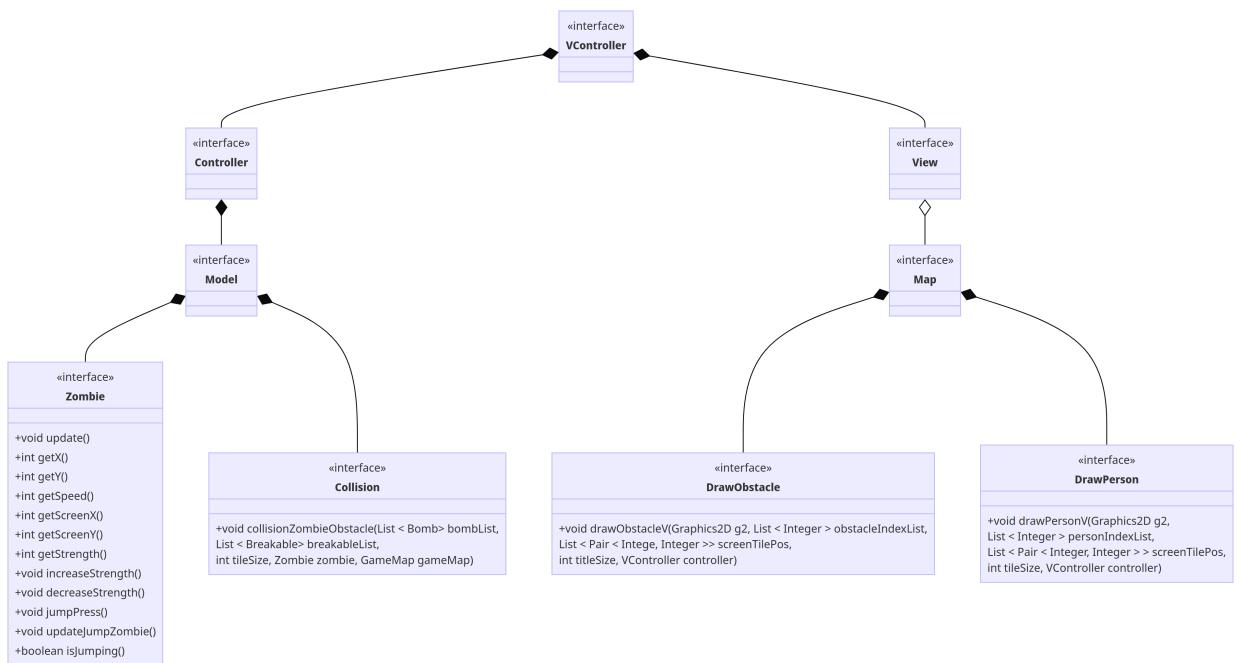


Figura 2.12: UML della classe che controlla le collisioni

2.2.3 Lorenzo Tordi

Gestione degli elementi della mappa.

Per la realizzazione della mappa di gioco ho suddiviso la logica della sua rappresentazione dalla sua effettiva grafica, suddividendo la sua parte di Model e la sua parte di View.

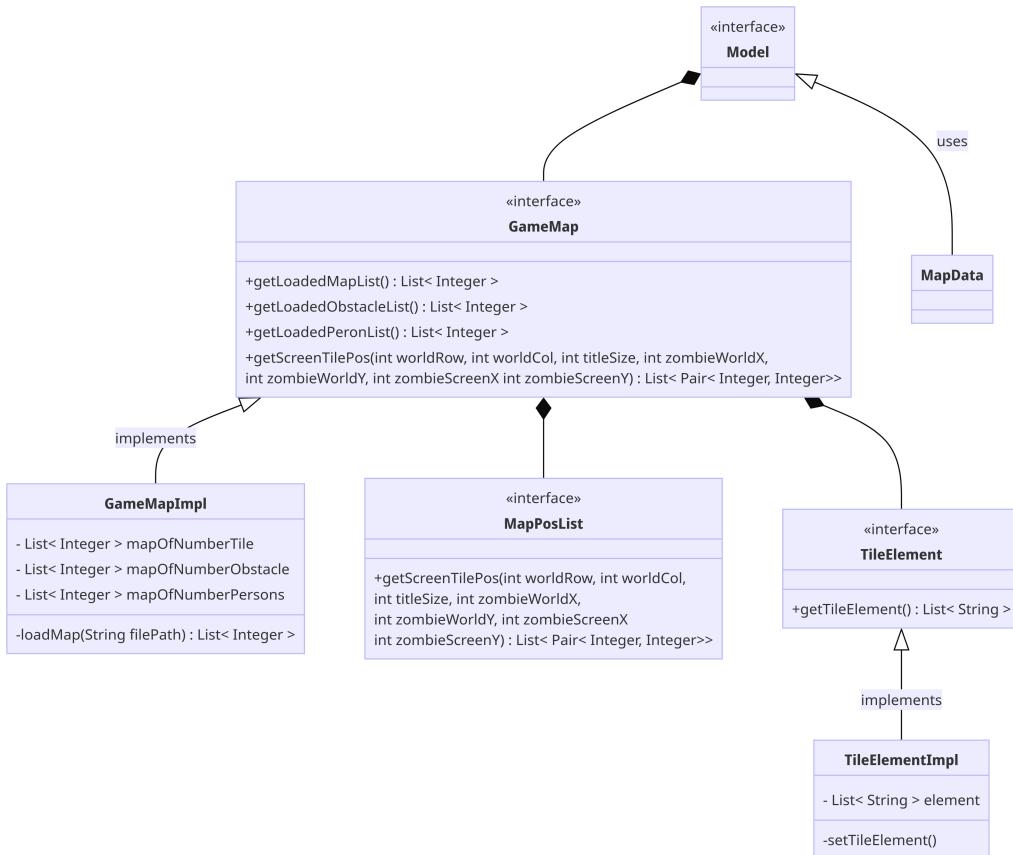


Figura 2.13: UML della logica della mappa di gioco e delle sue parti rappresentative.

Problema: la mappa è composta da diversi tasselli, ognuno con associato il proprio valore numerico (0 per un tassello, 3 per un altro e così via...), e la propria posizione nella finestra di gioco. Inoltre la mappa è inizialmente un file di testo con all'interno solo i valori numerici dei vari tasselli.

Soluzione: ho scelto di assegnare a ogni caratteristica del problema un suo oggetto personale per una maggiore pulizia, organizzazione e riusabilità del codice:

- Per l'assegnazione di ogni tassello al proprio valore numerico che lo rappresenta, abbiamo l'interfaccia **TileElement**, implementata da **TileElementImpl**, che si occupa di registrare il nome di ogni tipo di tassello presente nella mappa, associandolo al proprio valore numerico in base alla loro posizione (per esempio, in posizione 0 avremo il tassello che rappresenta la terra, che è a sua volta rappresentato dal valore 0 in mappa) attrav-

verso il metodo `setTileElement()`, per poi restituirlo con il metodo `getTileElement()`.

- Per l'assegnazione a ogni tassello della sua personale posizione nella schermata di gioco, abbiamo l'interfaccia `MapPosList` e il suo metodo `getScreenTilePos(...)`, che sfrutta le dimensioni generali della mappa di gioco e le coordinate dello zombie per assegnare progressivamente le posizioni attraverso dei `Pair` in base al movimento dello zombie.
- Una interfaccia `GameMap` implementata dalla classe `GameMapImpl` con il metodo `loadMap(String filePath)` che permette la riusabilità e la lettura di vari file di testo delle mappe presenti nel progetto, registrando i valori letti, per poi usarli associati ai loro rispettivi tasselli.

Per tenere traccia delle varie informazioni riguardo la mappa di gioco, le sue dimensioni (numero di colonne e righe) e quelle della schermata principale, ho ideato una utility class `MapData`, usata dal Model per attingere ai parametri richiesti poi dai vari metodi.

Meccanica di Vittoria.

Per gestire la possibilità di vittoria del giocatore, ho impostato alla fine della mappa di gioco un tassello con valore numerico rappresentativo 10 raffigurante una bandiera, che se raggiunta porta appunto alla vittoria e alla conquista del livello.

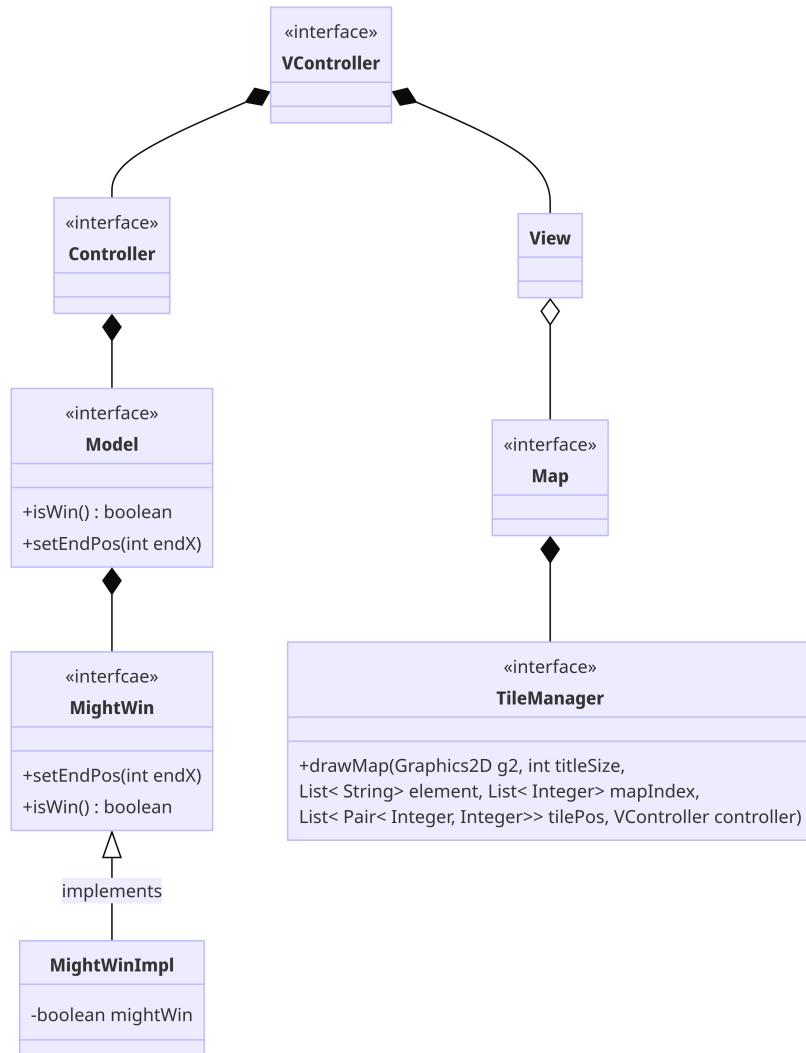


Figura 2.14: UML che rappresenta la logica di vittoria e la sua comunicazione con gli elementi disegnati in mappa.

Problema: la mappa del livello viene, graficamente parlando, caricata progressivamente all'avanzare dello zombie, non permettendo quindi di considerare oggetti oltre la visuale della camera di gioco, e quindi l'immediata considerazione del tassello che indica la fine del livello.

Soluzione: continuando a tenere separata la logica della vittoria dall'esperienza visiva dell'utente, ho scelto di creare l'interfaccia `MightWin`, implementata dalla classe `MightWinImpl`, e i suoi due metodi:

- `setEndPos(int endX)` che, in comunicazione con `Model`, `Controller` e `VController`, viene utilizzata direttamente all'interno del metodo `drawMap(...)`

dell’interfaccia `TileManager`, così che al presentarsi nella grafica della mappa il tassello rappresentante la bandiera della vittoria, prenda la sua coordinata X, la registri e la continui ad aggiornare fino a quando è presente.

- `isWin()` ci dice se la bandiera di fine livello ha raggiunto la posizione necessaria alla vittoria nella schermata di gioco.

In questo modo la possibilità di vittoria si manifesta solo nel momento in cui viene rappresentata nella camera di gioco la bandiera della fine del livello.

Pannello principale del livello giocabile.

Per gestire l’intero livello giocabile, e rappresentare graficamente tutti gli elementi del gioco (zombie, ostacoli, persone e mappa) ho optato per la realizzazione della classe `MapImpl`, che implementa l’interfaccia `Map`, estende `JPanel` di java swing, e implementa `Runnable` per la gestione del thread principale del gioco.

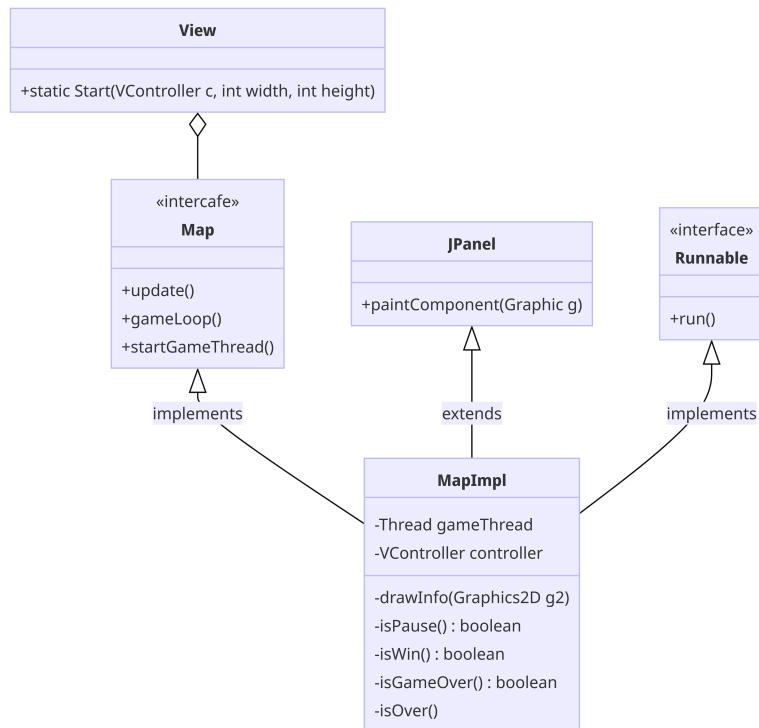


Figura 2.15: UML del pannello principale del gioco, rappresentato dall’interfaccia `Map` e la classe `MapImpl`.

Problema: il gioco deve poter rappresentare e aggiornare assiduamente le informazioni sui suoi componenti, svolgendo controlli sulle interazioni tra i vari oggetti (come per le collisioni) e sull'avanzare di posizione dello zombie. Inoltre, essendo estensione di JPanel e considerandosi quindi un pannello, ha bisogno di un frame principale.

Soluzione: per la realizzazione di un sistema adeguato al problema ho optato per la realizzazione della classe `View` e dell'interfaccia `Map` implementata da `MapImpl`.

- La classe `View` contiene il metodo `start(VController c, int width, int height)` per avviare il JFrame principale del gioco, che ospiterà tutte le varie scene, avvalendosi del principale componente `VController` e delle misure per la grandezza del frame. Possiamo considerare questa classe come l'involucro del gioco.
- L'interfaccia `Map`, assieme alla sua classe che la implementa, può essere considerata il motore principale del gioco. Presentandosi come un `JPanel`, estendendo quest'ultimo, e implementando `Runnable`, la classe `MapImpl` offre il thread principale, supportato dai metodi `update()`, `gameLoop()` e `startGameThread()`. Se nel metodo `update()` vengono effettuati i controlli come le collisioni e il movimento dello zombie, in `gameLoop()` abbiamo il cuore del motore di gioco, che decide il numero di FPS e svolge l'`update()` e il `repaint()`, personalizzato grazie all'override del metodo `paintComponent(Graphic g)` del `JPanel`. La classe gestisce anche le situazioni di pausa, vittoria, game over e shutdown del gioco grazie ai metodi `isPause()`, `isWin()`, `isGameOver()` e `isOver()`.

Sia la classe `View` che `MapImpl` hanno lo stesso `VController` in comune, in quanto il modello del software prevede la creazione di un solo `VController` poi comune per tutti. Inoltre per quanto riguarda il cuore del game engine, ovvero il metodo `gameLoop()`, ho deciso di implementarlo direttamente all'interno della classe principale del gioco, e non di separarlo da essa, in quanto dovendo svolgere il compito di ridisegno del pannello, avevo bisogno del diretto accesso al metodo `repaint()` del `JPanel`.

Caricamento delle immagini della mappa di gioco.

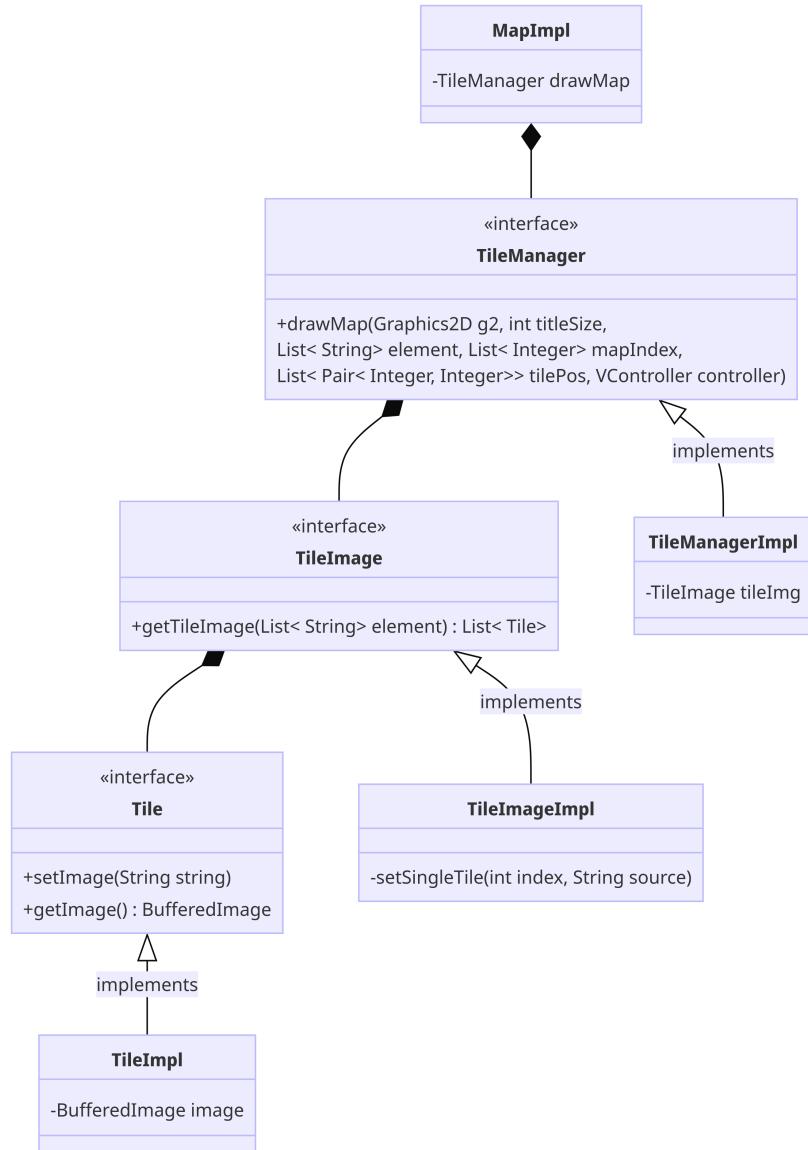


Figura 2.16: UML della logica di caricamento dei tasselli della mappa di gioco.

Problema: la mappa di gioco è composta da diversi tasselli che rappresentano parti diverse di quest'ultima, dovendo quindi prima attingere alle risorse del progetto per la grafica, e poi assegnarle correttamente alle loro posizioni e ai valori numerici.

Soluzione: per il caricamento dei tasselli della mappa ho deciso di suddividere il meccanismo in due principali interfacce, `TileImage` e `TileManager`, e creare un’interfaccia per rappresentare i singoli tasselli chiamata `Tile`.

- L’interfaccia `Tile`, implementata dalla classe `TileImpl`, permette di impostare una immagine attraverso il metodo `setImage(String string)`, e restituirla con `getImage()`. Così ogni tassello ha una sua immagine correlata, e in generale, il codice risulta riusabile e ordinato.
- L’interfaccia `TileImage`, implementata dalla classe `TileImageImpl`, ha il ruolo di prendere i `Tile` e assegnarli ognuno al proprio valore numerico rappresentativo in base al loro elemento, avvalendosi del metodo privato `setSingleTile(int index, String source)` (l’`index` rappresenta il valore numerico e `source` il percorso da seguire per arrivare alla risorsa dell’immagine del tassello), che a sua volta richiama i metodi dell’interfaccia `Tile`. Così lavorando si ottiene una lista di `Tile` dal metodo `getTileImage(...)` già assegnati alle loro caratteristiche (valore numerico e immagine).
- Infine l’interfaccia `TileManager`, implementata anche essa da una classe, `TileManagerImpl`, ha il ruolo di disegnare con il metodo `drawMap(...)`, avvalendosi delle precedenti interfacce (e dell’oggetto `Graphics2D`), l’intera mappa di gioco, assegnando le giuste posizioni a ogni tassello. E’ in questo metodo che si attinge alla logica per l’organizzazione della mappa assegnata al Model. Il metodo `drawMap` rimane in questo modo separato da tutto il resto, non rendendolo dipendente dal complesso, ma anzi, in situazioni di modifica della mappa, rimane efficace e accessibile.

Utilizzando questo tipo di design, rimane comunque perfettamente separata la logica dalla rappresentazione per l’utente, lasciando il tutto riusabile e ordinato.

Gestione e rappresentazione delle varie scene di gioco.

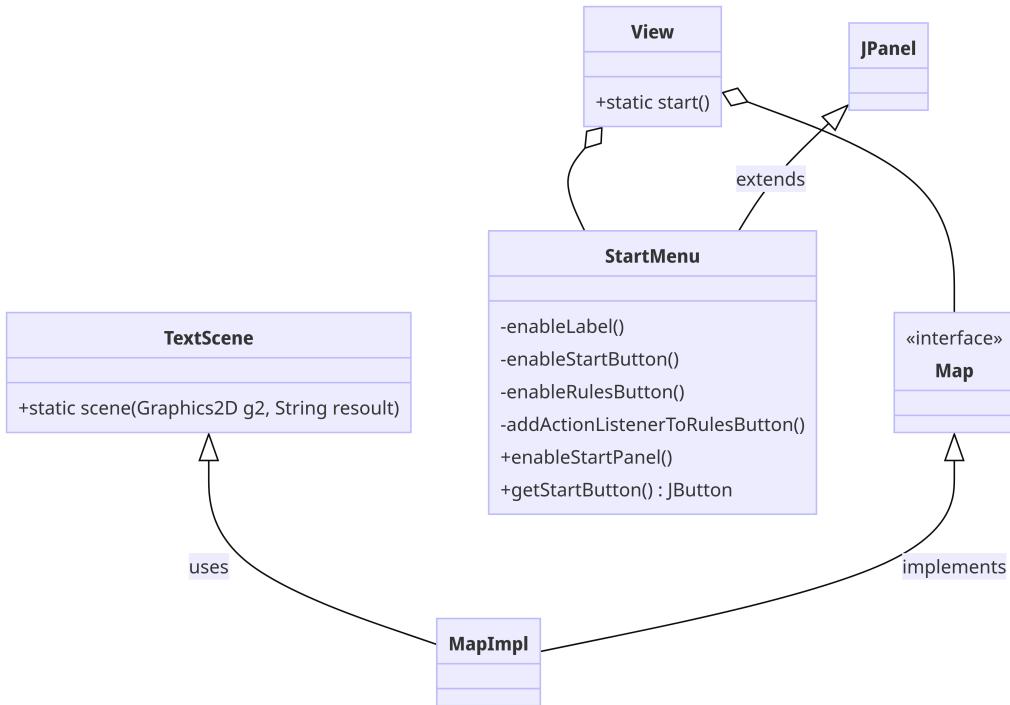


Figura 2.17: UML delle meccanica per le varie scene di menu iniziale, pausa, vittoria e game over del gioco.

Problema: il software deve essere in grado di gestire una schermata di menù iniziale ma anche le varie scene di pausa, vittoria e game over, senza interrompere il thread, ma solo fermando il ciclo di gioco principale.

Soluzione: ho deciso di implementare una seconda classe **StartMenu** che estende **JPanel**, così da poter gestire il menu e il livello giocabile entrambi come pannelli del frame principale. Il pannello in questione, con due buttoni, permette così di visionare una sezione dedicata alle regole del gioco, e un'azione per iniziare a giocare, abilitando il tutto con il metodo `enableStartPanel()`. Riguardo il bottone per l'avvio del livello giocabile, essendo la classe **View** a gestire i vari pannelli nel frame, e dovendo sostituire i pannelli all'occorrenza, l'oggetto in questione viene restituito direttamente alla classe **View** con il metodo `getStartButton()`, potendo così configurare correttamente la rimozione del menù iniziale e l'aggiunta di quello del livello. Per la gestione invece delle scene di pausa, vittoria e game over, ho deciso di optare per una classe più riusabile per le varie occasioni, chiamata **TextScene**: questa utility class permette di disegnare sul display grazie al suo metodo

`scene(Graphic2D g2, String result)` un rettangolo in sovrapposizione con al suo interno scritto il risultato voluto tra pausa, vittoria e game over. Il tutto viene poi gestito dal game loop principale, fermandosi nell'eventualità si abbia una delle possibili condizioni prima citate, ridisegnando il pannello e rendendo visibili le modifiche.

2.2.4 Lukasz Wojnicz

Gestione Persone sulla mappa.

Per la realizzazione dei civili ho creato una classe *Person* implementata in *PersonImpl*, la quale imposta le coordinate del civile.

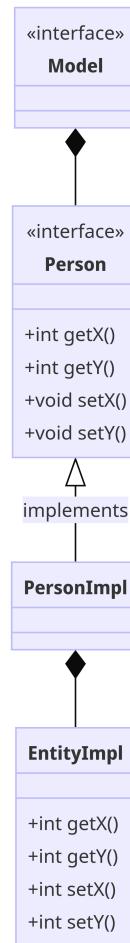


Figura 2.18: UML della logica di Person con i corrispettivi metodi

Person al suo interno ha una chiamata a *Entity*, la quale gestisce le coordinate *X* e *Y*.

Per poter stampare i civili, ho optato per una mappa testuale fatta di 0 e 1: lo 0 corrisponde a celle vuote mentre gli 1 corrispondono ai civili. Nella classe *PersonsManagerImpl*, che implementa *PersonsManager*, ho una lista di civili che viene riempita dal metodo `setPersonFromMap(...)`, questo mi permette di stampare i civili sfruttando il metodo `drawPersonV(...)` nella view, ed è all'interno dell'interfaccia *DrawPerson* che viene implementata da *DrawPersonImpl*. Oltre a ciò la classe *PersonsManager* ha altri metodi:

- `getPersonList()` che consente di ottenere la lista dei civili.
- `addPerson()` che permette di aggiungere civili alla lista.

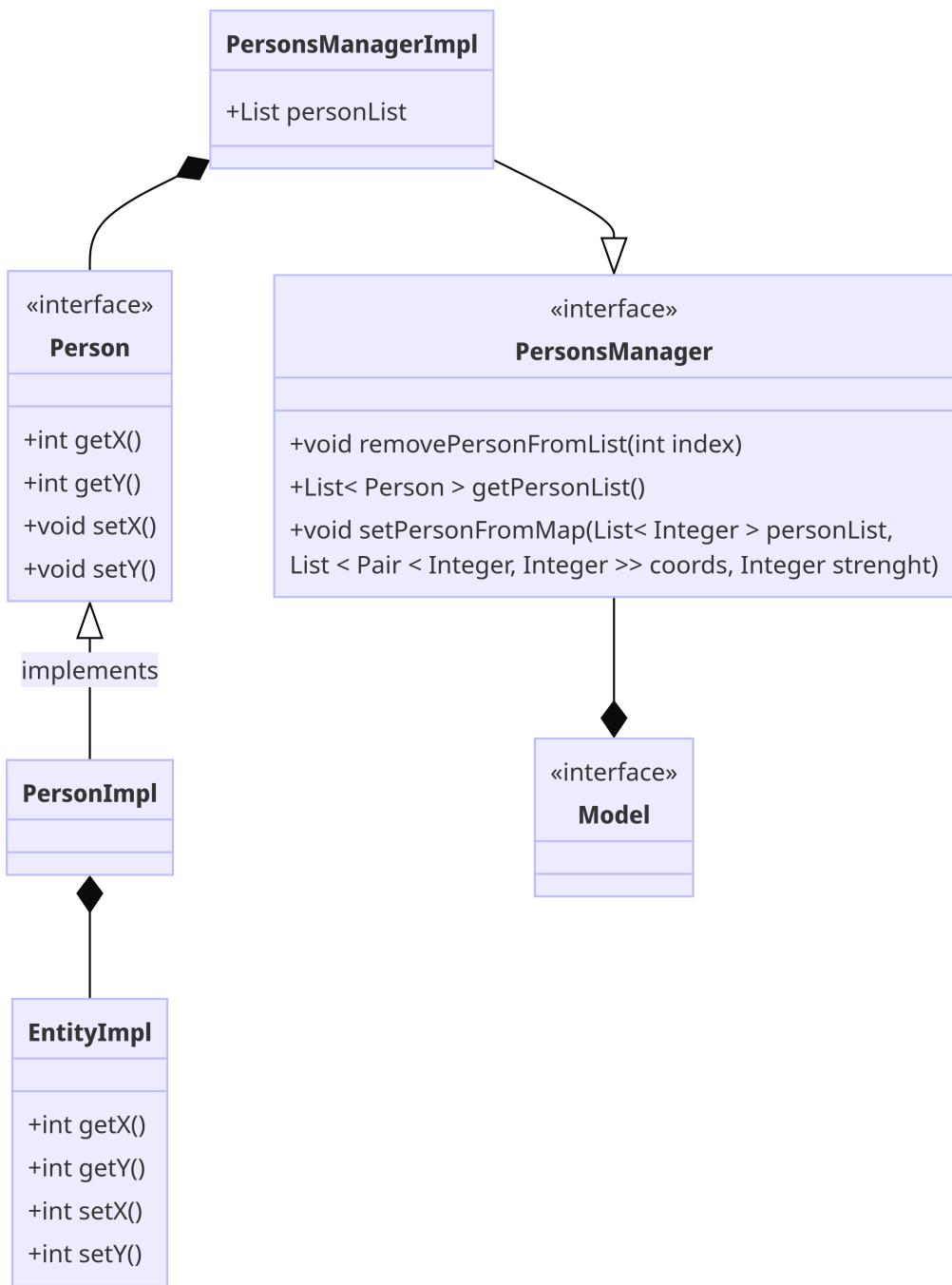


Figura 2.19: UML della logica di PersonManager

Per rappresentare graficamente i civili ho creato *DrawPerson* che è implementata in *DrawPersonImpl*, al suo interno troviamo il metodo citato

precedentemente. Per la visualizzazione su schermo dei civili ho implementato un’istanza *BufferedImage*, che attraverso un percorso specifico recuperano l’immagine e utilizzando due metodi privati: *getChange* e *increaseChange* permettono l’alternanza di due immagini; creando l’illusione che il civile si muova. Inoltre all’interno di *drawPersonV* si trova *getPerson()* che permette l’aggiornamento di ogni frame sullo schermo.

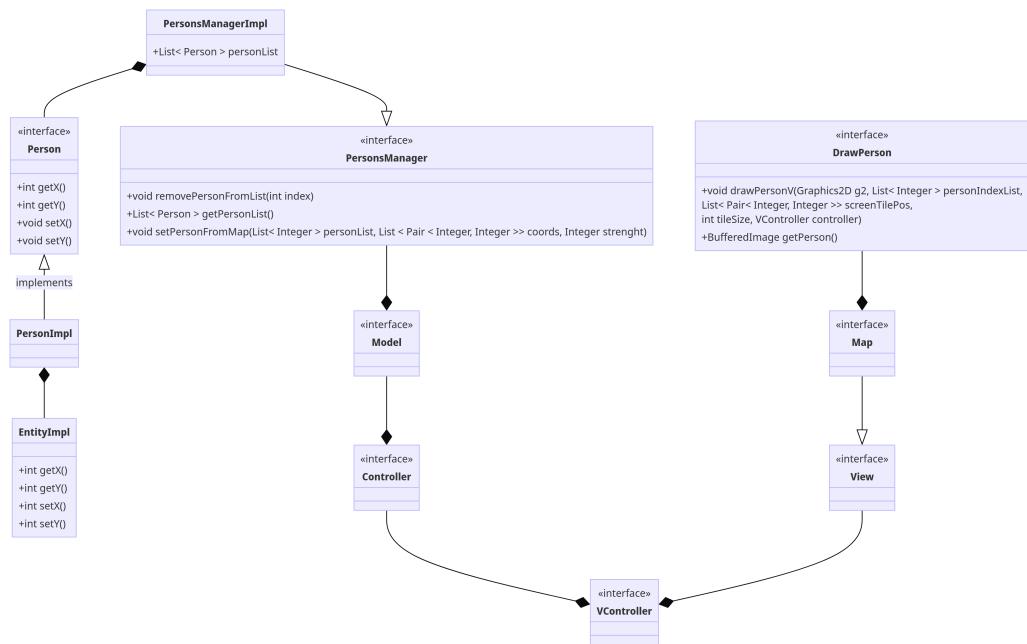


Figura 2.20: UML della logica di *drawPerson*

Gestione collisioni tra Persone e Zombie.

Per gestire la collisione tra civile e zombie ho creato un range in cui si trova un civile, se lo zombi entra in questo range, la forza dello zombie verrà aumentata di 1 e il civile sparirà.

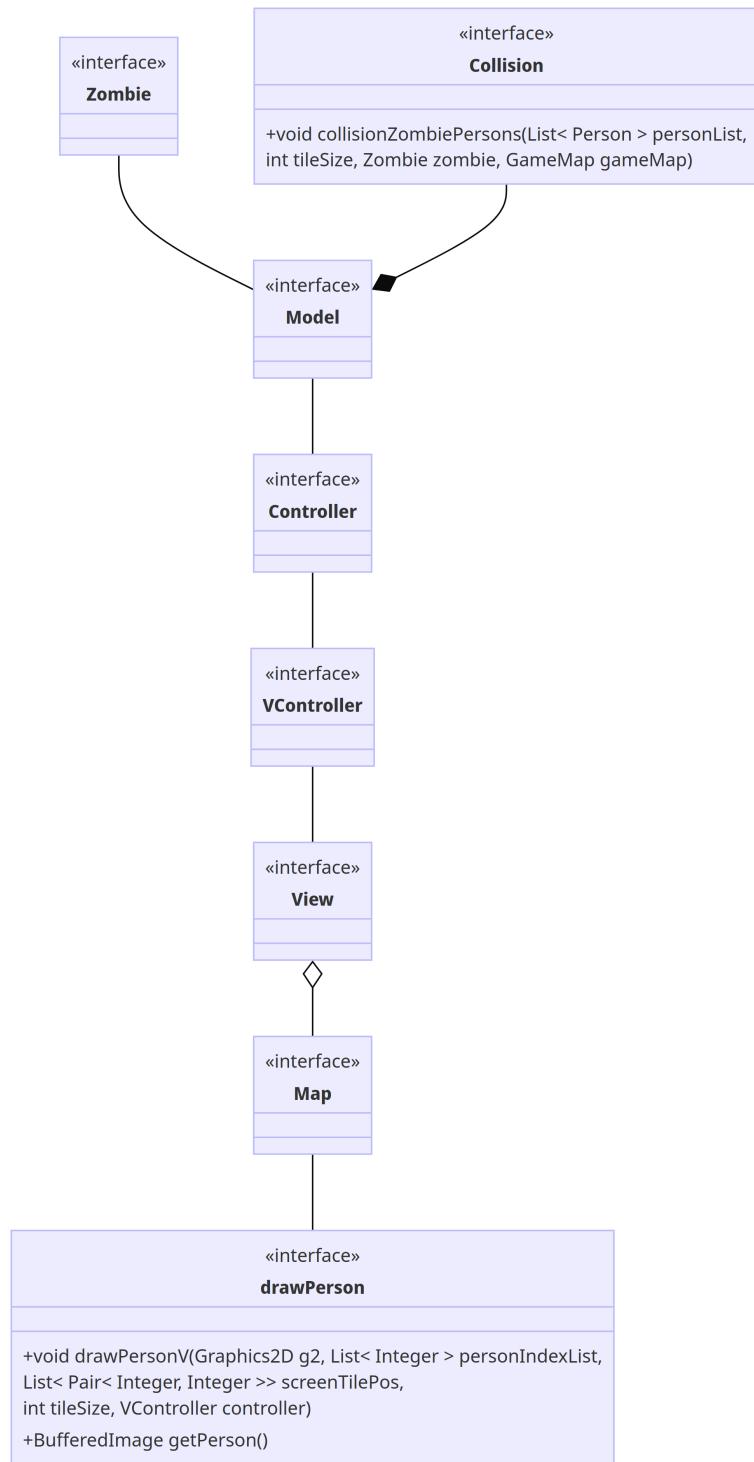


Figura 2.21: UML della logica sulle collisioni di Person

Meccanica di Sconfitta

Ci sono due modalità di sconfitta:

- *isNotEnough()* che accade nel caso lo zombie vada a scontrarsi con un Breakable, ma non ha forza sufficiente per romperlo.
- *isStrengthZero()* che accade quando la forza dello zombie raggiunge zero.

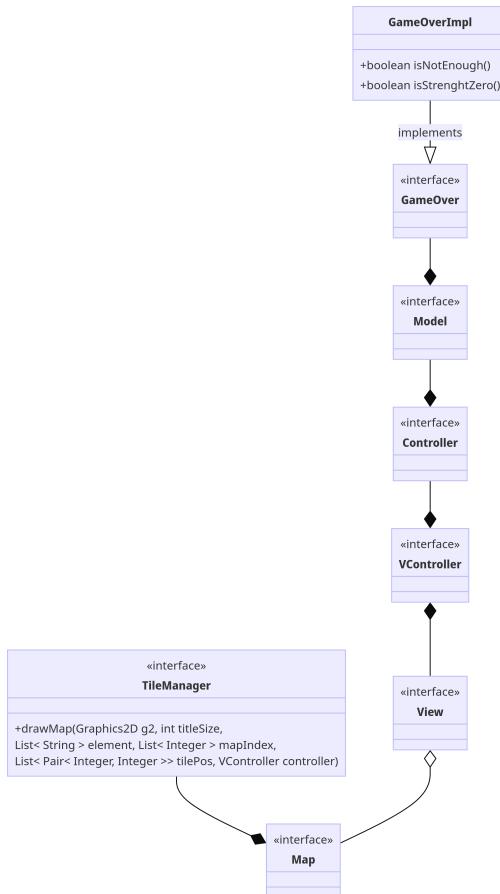


Figura 2.22: UML della logica sul Game Over

In caso di Game Over, a seguito di una delle due condizioni sopra elencate, apparirà la scritta "LOSE" e dopo di che si chiuderà in automatico la schermata di gioco.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il testing automatizzato dell'applicazione è stato svolto su elementi logici di quest'ultima. Sono stati infatti testati generalmente componenti singole del dominio:

- per la mappa di gioco, si è testato il corretto assegnamento degli elementi dei tasselli e la corretta lettura e assegnazione dei valori della mappa (letti dal file .txt). Successivamente si è provveduto a testare la corretta rappresentazione del tutto direttamente sul display di gioco, controllando il fluido movimento della mappa.
- per la gestione dello zombie, sono stati valutati i valori iniziali se sono correttamente inizializzati. Successivamente si controlla che venga aggiornata correttamente la coordinata X, mantenendo la Y costante. Per la verifica del funzionamento del salto viene controllato che lo zombie entri nello stato apposito, raggiunga l'altezza massima e ritorni alla sua posizione originale. Inoltre viene verificato che lo zombie non possa iniziare un nuovo salto nel mentre ne sta eseguendo uno. Per il test della forza vengono provati i metodi che incrementano e diminuiscono quest'ultima, poi confrontati con i risultati teorici. Infine viene controllato se tutti i metodi che sono forniti al model, restituiscono gli stessi risultati di quelli dello zombie.
- Nel contesto della gestione di bombe e breakable, l'approccio è stato uniforme per entrambi gli ostacoli, assicurandosi che le coordinate venissero impostate correttamente. Si sono verificate l'aggiunta e la rimozione di ostacoli, garantendo che fossero *null* dopo la rimozione, e si è

controllato il corretto riempimento delle liste con `fillBombListFromMap(...)` e `fillBreakableListFromMap(...)`. Per la classe `CollisionImpl`, i test sono stati effettuati con bombe e breakable sovrapposti a uno zombie. La funzione `collisionZombieObstacle(...)` è stata testata con successo, rimuovendo oggetti e assicurandosi che le liste fossero vuote. Ulteriormente, è stato verificato il corretto funzionamento di `isGameOver()`, confrontandone il comportamento con il Model.

- Per la gestione dei civili, si è assicurati che le coordinate venissero impostate correttamente. Inoltre si è verificata l'aggiunta e la rimozione dei civili dalle liste, quindi anche il corretto funzionamento di `setPersonFromMap`, controllando il corretto riempimento. Poi si sono effettuati dei test per la collisione dei civili, controllando `collisionZombiePersons` testando la rimozione dei civili e se le liste fossero vuote.

Per la scrittura e lo svolgimento dei test, ‘è stato utilizzato il framework JUnit 5.

3.2 Note di sviluppo

3.2.1 Luca Trianti

Uso della libreria javax.imageio.ImageIO per caricare le immagini

Permalink: <https://github.com/Lukottino/OOP23-zombietsunami/blob/main/src/main/java/zombietsunami/view/obstacleview/impl/DrawObstacleImpl.java>

Uso della libreria java.awt.Graphics2D

Permalink: <https://github.com/Lukottino/OOP23-zombietsunami/blob/main/src/main/java/zombietsunami/view/obstacleview/impl/DrawObstacleImpl.java>

Uso della libreria java.util.logging.Logger per gestire l’errore nel caso in cui non si riesca a recuperare l’immagine

Permalink: <https://github.com/Lukottino/OOP23-zombietsunami/blob/main/src/main/java/zombietsunami/view/obstacleview/impl/DrawObstacleImpl.java>

3.2.2 Alex Frisoni

Uso della libreria javax.imageio.ImageIO per caricare le immagini

Permalink: <https://github.com/Lukottino/OOP23-zombietsunami/blob/main/src/main/java/zombietsunami/view/zombieview/impl/DrawZombieImpl.java>

Uso della libreria java.awt.Graphics2D

Permalinks: <https://github.com/Lukottino/OOP23-zombietsunami/blob/main/src/main/java/zombietsunami/view/zombieview/impl/DrawZombieImpl.java>

Uso della libreria java.util.logging.Logger per gestire l'errore nel caso in cui non si riesca a recuperare l'immagine

Permalink: <https://github.com/Lukottino/OOP23-zombietsunami/blob/main/src/main/java/zombietsunami/view/zombieview/impl/DrawZombieImpl.java>

Uso della libreria java.awt.event.KeyListener per l'input da tastiera

Permalink: <https://github.com/Lukottino/OOP23-zombietsunami/blob/main/src/main/java/zombietsunami/view/KeyHandlerImpl.java>

Uso della libreria java.awt.event.KeyEvent per la gestione degli eventi

Permalink: <https://github.com/Lukottino/OOP23-zombietsunami/blob/main/src/main/java/zombietsunami/view/KeyHandlerImpl.java>

3.2.3 Lorenzo Tordi

Uso di Stream

Permalink: <https://github.com/Lukottino/OOP23-zombietsunami/blob/bdc7aceb4a63c79a7ef0b2b0248be1638aef28a1/src/main/java/zombietsunami/model/mapmodel/impl/GameMapImpl.java#L84>

Uso della libreria javax.imageio.ImageIO per caricare immagini

Permalink: <https://github.com/Lukottino/OOP23-zombietsunami/blob/bdc7aceb4a63c79a7ef0b2b0248be1638aef28a1/src/main/java/zombietsunami/view/mapview/impl/TileImpl.java#L24>

Uso della libreria java.awt.Graphics2D

Permalinks:

<https://github.com/Lukottino/OOP23-zombietsunami/blob/bdc7aceb4a63c79a7ef0b2b0248be1638aef28a1/src/main/java/zombietsunami/view/mapview/impl/MapImpl.java#L114>

<https://github.com/Lukottino/OOP23-zombietsunami/blob/bdc7aceb4a63c79a7ef0b2b0248be1638aef28a1/src/main/java/zombietsunami/view/TextScene.java#L29>

Link alle risorse utilizzate:

- Codice della classe Pair presa dal docente: <https://bitbucket.org/mviroli/oop2022-esami/src/master/a01a/e1/Pair.java>
- Ispirazione per la meccanica del metodo gameLoop() e movimento della camera di gioco: https://www.youtube.com/watch?v=VpH33Uw-_0E
https://www.youtube.com/watch?v=Ny_YHoTYcxo

3.2.4 Lukasz Wojnicz

Uso della libreria javax.imageio.ImageIO per caricare le immagini

Permalink: <https://github.com/Lukottino/OOP23-zombietsunami/blob/main/src/main/java/zombietsunami/view/personview/impl/DrawPersonImpl.java>

Uso della libreria java.awt.Graphics2D

Permalink: <https://github.com/Lukottino/OOP23-zombietsunami/blob/main/src/main/java/zombietsunami/view/personview/impl/DrawPersonImpl.java>

Uso della libreria java.util.logging.Logger per gestire l'errore nel caso in cui non si riesca a recuperare l'immagine

Permalink: <https://github.com/Lukottino/OOP23-zombietsunami/blob/main/src/main/java/zombietsunami/view/personview/impl/DrawPersonImpl.java>

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Luca Trianti

Nel corso di questo progetto, ho assunto un ruolo significativo contribuendo in diversi aspetti chiave. In particolare, ho creato le classi Bomb e Breakable, gestendo l'implementazione dell'ObstacleManager. La mia attività è stata fondamentale per la definizione di ostacoli nel gioco, garantendo una loro corretta rappresentazione sulla mappa di gioco. Inoltre, ho collaborato con il resto del team per gestire le collisioni tra gli ostacoli e lo zombie, concentrando il mio impegno sulla classe Collision (e CollisionImpl). Questa componente si è rivelata essenziale per il corretto funzionamento del gioco, e sono soddisfatto del contributo che ho potuto offrire a questo aspetto cruciale del progetto. La comunicazione all'interno del team è stata pressocchè positiva. Ho avuto il piacere di lavorare in un ambiente in cui c'era un sostegno reciproco, e ho trovato molto utile il supporto offerto dagli altri membri del gruppo quando necessario. La mia esperienza nel risolvere problemi e bug è stata arricchente, e sono stato in grado di offrire la mia competenza quando c'era bisogno di risolvere eventuali difficoltà. Complessivamente, ritengo che la mia abilità di collaborare in un team sia notevolmente migliorata durante questo progetto. Ho imparato a comunicare in modo efficace con i membri del gruppo, a condividere idee e a risolvere con successo problemi insieme. Sono grato per l'opportunità di contribuire a questo progetto e ritengo che la mia partecipazione abbia apportato un valore significativo al team e al risultato finale del lavoro svolto.

4.1.2 Alex Frisoni

Come membro del team di sviluppo, ho partecipato attivamente alla realizzazione del progetto dello zombie, contribuendo in modo significativo alla sua implementazione e al raggiungimento degli obiettivi prefissati. Mi sono impegnato con costanza e dedizione nel portare avanti il progetto, grazie al quale ho acquisito molte conoscenze in più. Ho lavorato collaborando talvolta con gli altri membri del team, partecipando attivamente alle discussioni e offrendo il mio contributo per risolvere problemi. Riconosco che ci sono stati momenti in cui la mia comunicazione con il team avrebbe potuto essere più chiara ed efficace. Cercherò di migliorare la mia capacità di comunicazione. In conclusione, ritengo che il mio contributo al progetto dello zombie sia stato significativo, ma che ci siano margini di miglioramento su cui lavorare. Sono determinato a sviluppare le mie capacità, contribuendo al successo dei futuri lavori in team.

4.1.3 Lorenzo Tordi

Nonostante l'impegno messo per la realizzazione al meglio della mia parte personale, c'è sicuramente un margine di miglioramento per una programmazione più avanzata e tecnica. Detto ciò, durante tutta la realizzazione del software potrei affermare che la mia figura all'interno del gruppo è stata in ogni caso decisamente di spicco e aiuto per tutti i membri partecipanti, sia per quanto riguarda problemi logici, sia per problemi organizzativi, dalla generale architettura applicata (MVC in questo caso), alla gestione di meccaniche di collisione, lettura di mappe per oggetti e civili, e game over. Generalmente quindi, considerato l'impegno e il tempo dedicato, reputo il mio operato decisivo per il gruppo.

4.1.4 Lukasz Wojnicz

Come membro del team ho cercato di partecipare attivamente agli incontri virtuali, cercando di contribuire con il mio aiuto nei momenti di bisogno e chiedendo una mano nei momenti di difficoltà; gli altri membri del team sono sempre stati molto disponibili a rispondere ad eventuali domande su dubbi. Inoltre credo che il mio contributo al progetto sia stato sufficiente, ma ci sono molti margini di miglioramento. In conclusione questa esperienza mi ha aiutato a migliorare le mie capacità nel lavorare con altre persone, aiutandomi per lavori futuri.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Alex Frisoni

Il laboratorio non ha aiutato particolarmente alla mia formazione, dato che non l'ho trovato molto chiaro e soprattutto non ho trovato molto invogliante il metodo di incoraggiamento allo studio da parte degli insegnanti. Una questione, secondo me, molto rilevante è la tempistica della spiegazione per iniziare il progetto , che se fosse stata spiegata con più anticipo, ci si potrebbe approcciare ad esso in modo lento ma costante durante le lezioni, in modo tale che si abbia più tempo e che non si sovrappongano troppa mole di lavoro alla fine.

Appendice A

Guida utente

Si fa presente che per mettere in pausa e riprendere il gioco è necessario premere il tasto 'Esc' della tastiera, mentre per far saltare lo zombie si prema la barra spaziatrice.