Relazione per "Bombardero: the Bomberman Remake"

Federico Bagattoni, Daniele Merighi, Jacopo Turchi, Luca Venturini

11 luglio 2024

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il collettivo mira alla creazione di un videogioco ispirato alla nota saga videoludica "Bomberman". Il giocatore controlla un personaggio, in grado di muoversi liberamente nei confini della mappa e di piazzare delle bombe. L'obiettivo generale del gioco è quello di sconfiggere i nemici presenti sulla mappa piazzando strategicamente gli esplosivi, in grado di far saltare in aria eventuali ostacoli.

Requisiti funzionali

- L'applicazione dovrà fornire un ambiente di gioco virtuale rappresentato da una griglia bidimensionale. La griglia dovrà essere suddivisa in celle nelle quali i giocatori potranno muoversi solo orizzontalmente o verticalmente.

  La griglia verrà riempita da due tipologie di ostacoli:

  - Muri indistruttibili: generati in maniera fissa a scacchiera
  - Muri deboli: che verranno piazzati casualmente per riempire parzialmente i restanti spazi vuoti della griglia

- Dopo due minuti dall'inizio della partita i muri indistruttibili che formano il confine dell'arena inizieranno a collassare verso il centro, riducendo l'area calpestabile ed eliminando i giocatori che si trovano nelle loro prossimità al momento del crollo. I muri cominciano il loro crollo a partire dall'angolo in basso a sinistra, uno per volta, procedendo in senso orario.

- Si dovrà implementare un giocatore principale, controllabile da parte dell'utente mediante tastiera utilizzando i tasti W, A, S e D rispettivamente per il movimento verso l'alto, verso sinistra, verso il basso e verso destra. Verrà utilizzata la barra spaziatrice per il piazzamento delle bombe.

- Dovranno essere presenti tre nemici, controllati dal computer, il cui scopo sarà quello di sconfiggere gli altri giocatori presenti nella mappa.

- Ogni entità presente potrà piazzare delle bombe nella propria posizione corrente, che esploderanno dopo qualche secondo (permettendo ai giocatori di allontanarsi)

- L'esplosione causerà delle fiamme, che espandendosi verticalmente e orizzontalmente per un certo numero di caselle, cambieranno la conformazione dell'arena e danneggeranno le entità vicine. Inoltre, se la fiamma di una bomba ne colpisce un'altra, allora anche quest'ultima dovrà esplodere innestando un effetto a catena.

- Normalmente, una volta piazzata una bomba in una cella non sarà possibile attraversarla fino a quando essa non esplode.

- I giocatori possono attraversare celle in cui sono presenti anche altri player senza che essi presentino un ostacolo.

- Saranno presenti ostacoli nella griglia (muri deboli), che potranno essere distrutti dalle esplosioni delle bombe, permettendo al personaggio di passare attraverso lo spazio vuoto appena creato, diversamente dai muri indistruttibili che non potranno essere alterati.

- Dalle esplosioni degli ostacoli si possono generare degli oggetti speciali (powerup oppure malus), che cambieranno le abilità delle singole entità. Per ottenerli basterà raccoglierli passandoci sopra.

- Il software deve includere una breve guida che insegni all'utente come giocare e come muoversi all'interno del gioco

Requisiti non funzionali

- L'applicazione dovrà essere ottimizzata al fine di permettere un'esperienza di gioco fluida

1.2 Analisi e modello del dominio

L'arena di gioco detta `Game Map` sarà composta da `Cell`. Queste `Cell` potranno essere di diverso tipo:

- Muri indistruttibili, chiamati nel dominio `Unbreakable Wall`
- Muri distruttibili, chiamati nel dominio `Breakable Wall`
- Potenziamenti, che possono essere raccolti dai `Character`, chiamati nel dominio `PowerUp`
- Bombe, chiamate nel dominio `Bomb`
- Fiamme, generate dall'esposione delle bombe chiamate nel dominio `Flame`

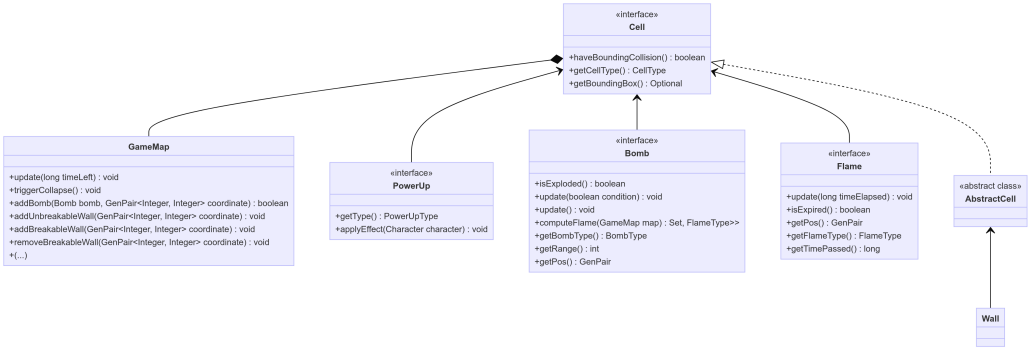

Figura 1.1: Schema UML sulle relazioni tra le varie entità di tipo Cell

Il `Character` si divide in `Player` e `Enemy`. Qualsiasi personaggio può piazzare delle bombe sulle celle libere della mappa che una volta esplose generano una serie di fiamme. Le fiamme interagiscono con i muri, con i potenziamenti e con le bombe modificando la struttura della mappa, distruggendo ciò che si trova nella stessa `Cell` ed espandendosi attraverso le celle libere. Un muro, se distrutto, può lasciare un potenziamento al suo posto.

Il tutto viene racchiuso nell'interfaccia `GameManager`, di cui si parlerà in dettaglio nella prossima sezione, nella seguente maniera:

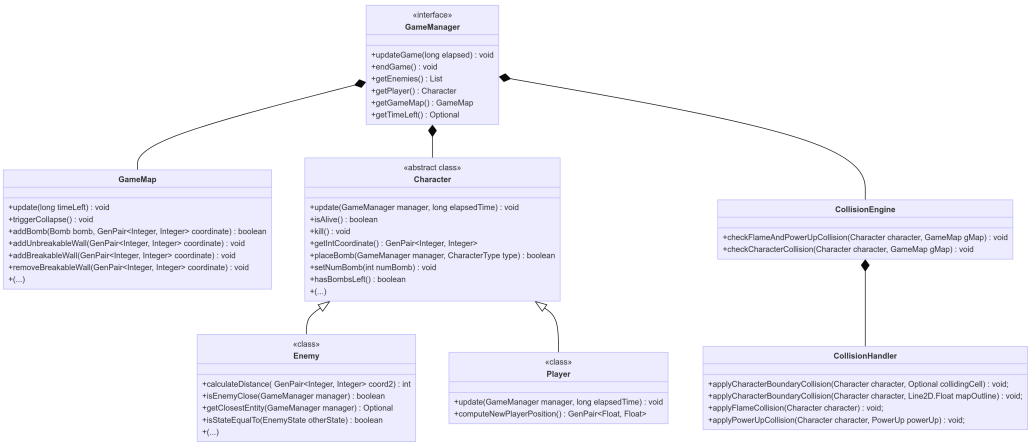

Figura 1.2: Schema UML riassuntivo sulla composizione generale del model

Capitolo 2

Design

2.1 Architettura

In questo applicativo è stato utilizzato il pattern architetturale model-view-controller. Le interfacce che compongono il pattern sono `Controller`, `GraphicsEngine` e `GameManager` mentre per quanto riguarda il game loop necessario ad aggiornare ciclicamente le componenti di gioco sia di model che di view è stata creata l'interfaccia `Engine`. Il tutto è descritto in figura 2.1

L'interfaccia `GameManager` è il punto d'ingresso al model. Si occupa dell'aggiornamento degli aspetti dinamici delle componenti del dominio e della gestione delle dinamiche di gioco.

`Controller` è l'interfaccia rappresentante l'omonimo componente dell'architettura M.V.C. esso fa da tramite tra la view, rappresentata da `GraphicsEngine` e `GameManager`; infine vi è l'interfaccia di gestione degli input dell'utente `KeyboardInput`.

Per quanto riguarda l'indipendenza degli elementi di questa architettura, il controller mette a disposizione i metodi necessari per poter comunicare all'interfaccia grafica quali sono gli elementi da mostrare. A sua volta l'interfaccia grafica mette a disposizione diversi metodi per permettere al controller di modificare la vista ed impostare visuali diverse. La conclusione è che l'interfaccia grafica è facilmente sostituibile con una nuova interfaccia al momento del bisogno, a meno di certi aggiustamenti che potrebbero essere necessari dal punto di vista implementativo.

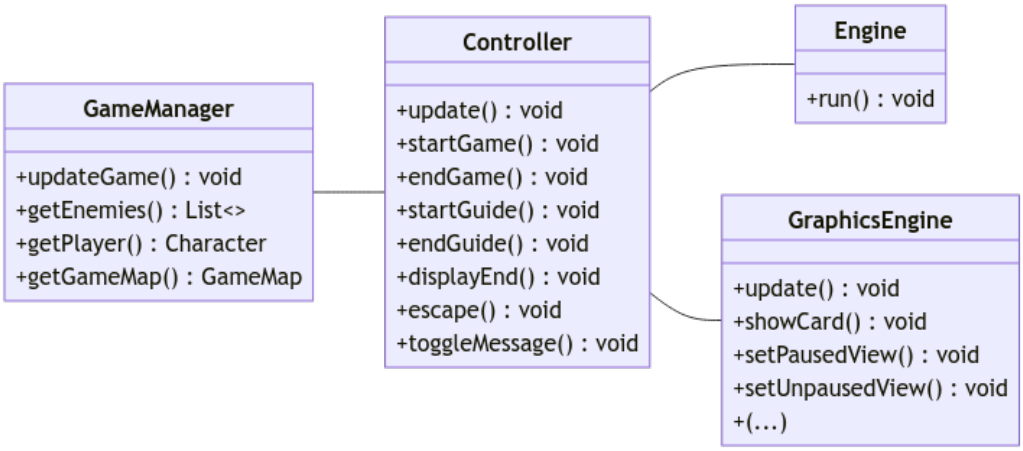

Figura 2.1: Schema UML riassuntivo dell'architettura M.V.C. dell'applicativo

2.2 Design dettagliato

2.2.1 Federico Bagattoni

Generazione della mappa di gioco

Le caratteristiche della mappa stabilite in fase di analisi impongono che essa sia formata, inizialmente, da muri distruttibili ed indistruttibili i primi disposti in ordine casuale ed i secondi disposti sempre nella stessa maniera a creare una griglia. Inoltre, durante la fase di collasso della mappa, le mura devono seguire un certo ordine nella loro caduta.
Al fine di generare i vari elementi sulla mappa è stata creata l'interfaccia `MapGenerator`, questa genera su richiesta i diversi elementi da inserire nella mappa.

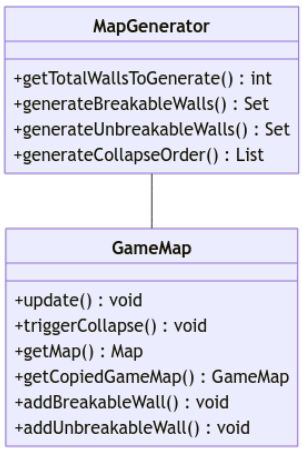

Figura 2.2: Schema UML generale riguardo la generazione della mappa

Problema: E' necessaria una maggiore flessibilità nella generazione del `collapse order` da parte di `MapGenerator`.

Soluzione: Si fa uso del pattern `Strategy` nella generazione del `collapse order` servendosi dell'interfaccia `MatrixTraversalStrategy`. Infatti il problema viene visto come un problema di travel attraverso una matrice e tramite l'interfaccia menzionata vengono forniti diversi modi di affrontare il problema.
A questo punto, la `GameMap` può scegliere facilmente quale strategia può utilizzare per il collasso della mappa. L'utilizzo del pattern consente di avere maggiore flessibilità anche nel caso si voglia espandere il gioco ed avere fantasia nella generazione di pattern di collasso.

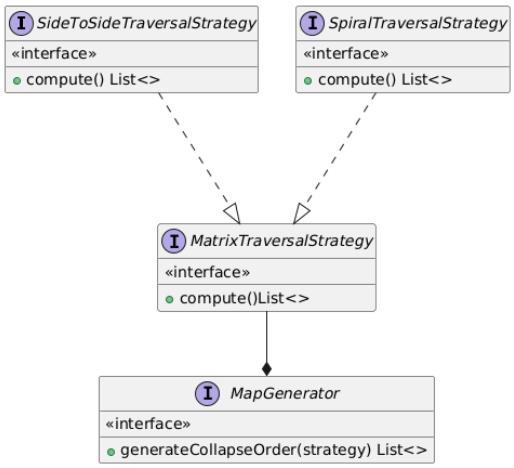

Figura 2.3: Schema UML del pattern strategy

Alternativa considerata: l'alternativa considerata era quella di implementare in maniera statica l'algoritmo, questa soluzione non favorisce il riuso, la dinamicità del codice e non è propensa a cambiamenti futuri.

Riuso del codice per la creazione della guida di gioco

Essendo l'interfaccia `GameManager` deputata alla organizzazione e sincronizzazione di tutti gli elementi di gioco viene necessario approfondire questa tematica quando si vuole creare una *guida intertattiva* che permetta ad un giocatore principiante di imparare le basi del gioco.

Problema: è necessario creare diverse versioni di `GameManager` specializzate in diverse parti del gioco.

Soluzione: creazione di una classe comune che vada a favorire il riuso specializzandola in diverse classi a seconda della modalità di gioco che si vuole controllare. Inoltre viene creata un interfaccia `GuideManager` che estende il contratto di `GameManager`.

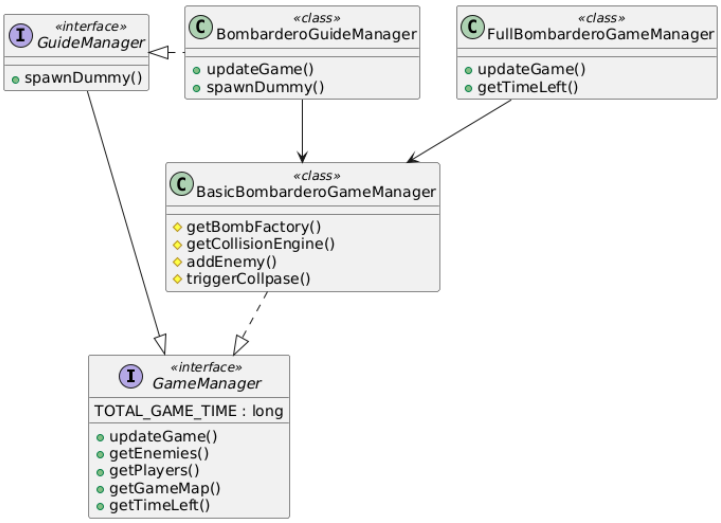

Figura 2.4: Schema UML sul riuso nel contesto di `GameManager` e `GuideManager`

Infatti la classe `BasicBombarderoGameManager` implementa un comportamento standard che *tutti* i *manager* dovrebbero avere cioè quello di aggiornare tutte le componenti, finire la partita in vittoria o sconfitta e generare nelle posizioni corrette i diversi elemenenti.

2.2.2 Luca Venturini

1. Problema da Risolvere

Fare in modo che i personaggi all'interno dell'arena non possano oltrepassare i muri della mappa e aluni tipi di Cell utilizzando un movimento su un piano continuo.

1. Soluzione Proposta

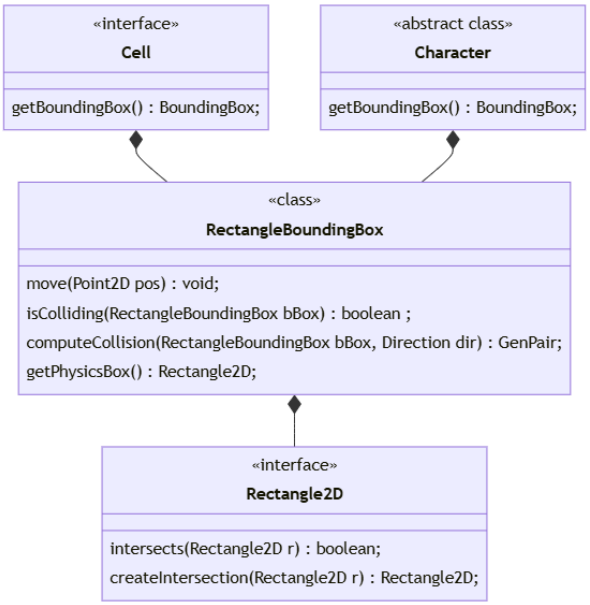

Figura 2.5: Schema UML riassuntivo della classe RectangleBoundingBox

Per risolvere questo problema è stato utilizzato il pattern `Adapter` dove la classe RectangleBoundingBox si compone di un Rectangle2D per descrivere la parte solida di un Character o di una Cell, di fatto RectangleBoundingBox utilizza i metodi di un Rectangle2D per ricavare informazioni utili, come nel caso di una collisione la distanza che il Character deve percorrere per non collidere più rispetto ad un muro. Con questa implementazione si lascia poco spazio a nuove BoundingBox magari di forme diverse però è stato valutato che in un dominio come questo fosse appropriata per la sua semplicità.

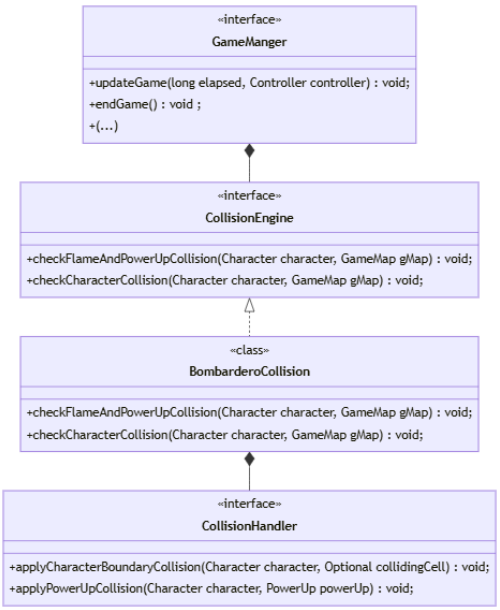

Figura 2.6: Schema UML riassuntivo della classe BombarderoCollision

Inoltre per rendere più libera la gestione delle collisioni il GameManager si affida ad una CollisionEngine, ovvero un interfaccia che si occupi di rilevare e risolvere le collisioni passatagli come `Strategia`. A sua volta l'implementazione utilizzata BombarderoCollision necessita di un CollisionHandler che sappia come risolvere le collisioni, visto che questa classe si occupa solo ti trovarle,anche questa passata come `Strategia`, così facendo si rispettano i principi DIP e SRP. Si è pensato che in questo modo nel caso di nuove modalità di gioco con interazioni diverse tra Character e Cells è facile aggiungere nuove implementazioni.

2. Problema da Risolvere

Nel gioco alcuni PowerUp danno il potere ai giocatori di utilizzare delle bombe che hanno effetti diversi come poter perforare i muri o esplodere da remoto

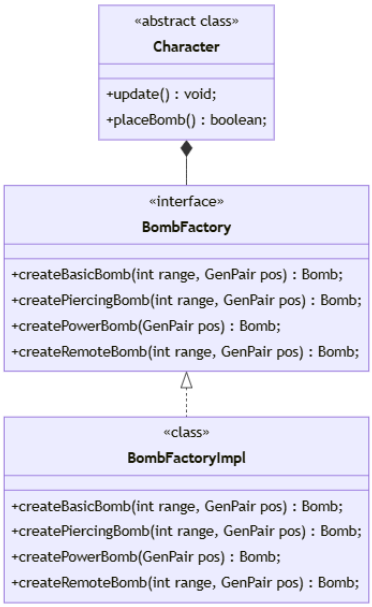

Figura 2.7: Schema UML riassuntivo della classe BombFactoryImpl

2. Soluzione Proposta

Per la creazione delle bombe viene utilizzata una BombFactory ovvero una classe che segue il pattern `Simple Factory` per cui con ogni suo metodo restituisce una Bomb con caratteristiche diverse. Tutti Character che sono in gioco quando creati richiedono una BombFactory alla quale quando, dovranno piazzare delle bombe, chiederanno la Bomb equivalente al PowerUp che hanno preso. Anche in questo caso l'utilizzo del pattern `Strategy` rende possibile una nuova implementazione di BombFactory nel caso di nuove modalita di gioco. L'implementazione delle diverse Bomb nella classe BombfactoryImpl viene fatta a partire dalla abtract class BasicBomb, sfruttando al massimo il riutilizzo del codice. Queste classi seguono il principio DIP e SRP in quanto tutte sono definite come interfacce con la loro relativa implementazione.

2. Alternativa Considerata

Nei metodi della BombFactoryImpl inizialmente era stato pensato l'utilizzo del pattern `Decorator` in alternativa all'implementazione delle Bomb a partire da una classe astratta per dare la possibilita di avere bombe con piu effetti (per esempio una bomba che perfori i muri e venga fatta esplodere da remoto), questa modalità è stata scartata quando il gruppo ha deciso di

tenere fede al gioco originale e non lasciare la possibilita di mescolare i tipi di bomba.

2.2.3 Jacopo Turchi

Creazione dei PowerUp

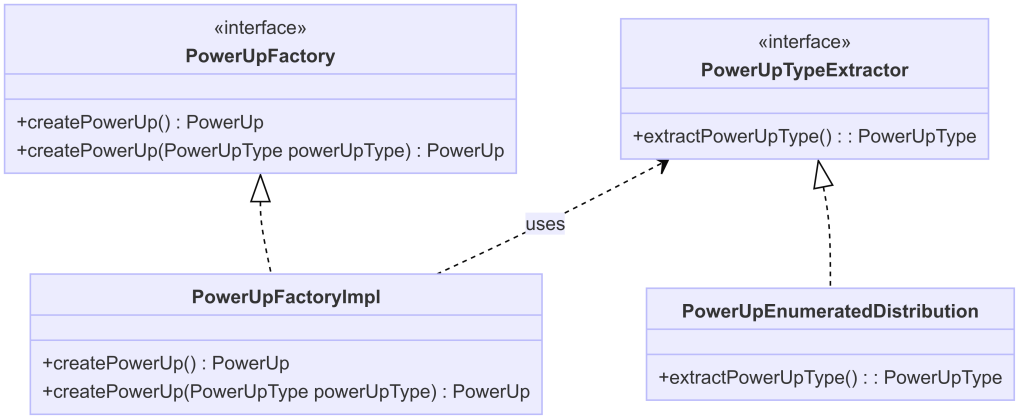

Figura 2.8: Schema UML della `PowerUpFactory` e della sua implementazione

Problema: I `PowerUp` devono essere generati a seguito della rottura di un `BreakableWall` in base alla loro rarità. Bisogna sviluppare una soluzione intelligente che permetta di creare PowerUp diversi.

Soluzione: Seguendo il pattern *Method Factory*, come da Figura 2.8 rispettiamo i principi SRP, OCP e DIP, dato che potenzialmente possono essere aggiunti nuovi PowerUp senza modificare il codice esistente, ma aggiungendone solo. Inoltre la factory ha il solo compito di creare i PowerUp e non dipende da implementazioni, ma dall'interfaccia.

Gestione degli effetti dei PowerUp

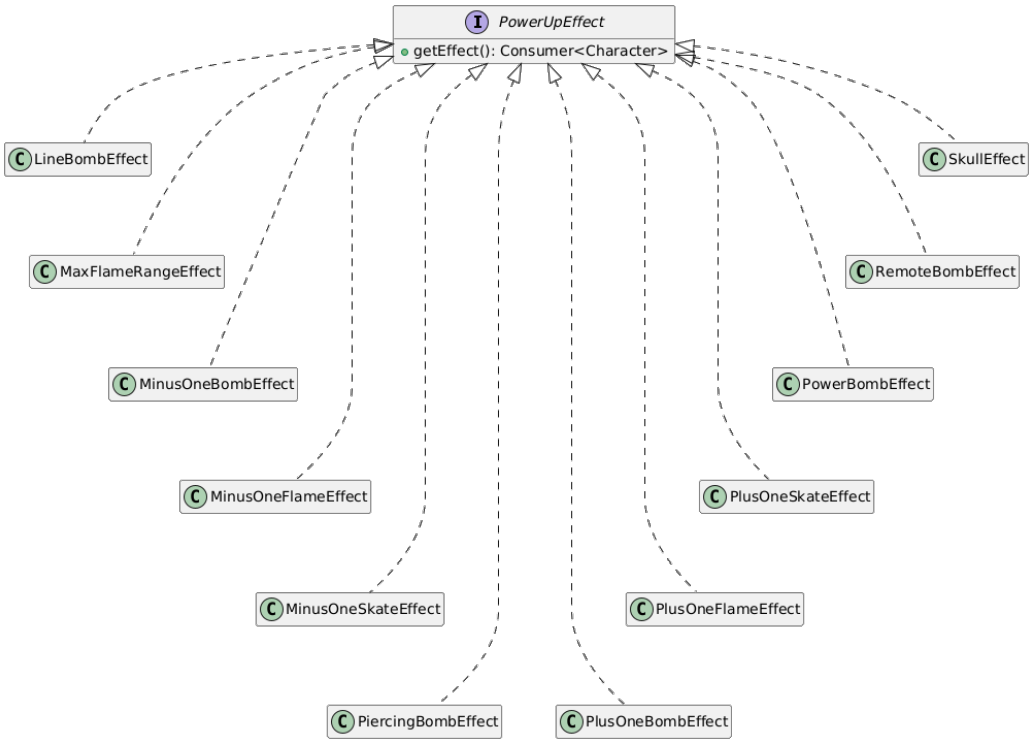

Figura 2.9: Schema UML del pattern Strategy per gli effetti dei `PowerUp`

Problema: I `PowerUp` sono caratterizzati da un `PowerUpEffect` che stabilisce l'effetto da applicare al `Character`. Questo effetto notiamo che deve essere intercambiabile ed è quindi necessario pensare a come incapsularne la logica.

Soluzione: Seguendo il pattern *Strategy*, come da Figura 2.9 abbiamo l'interfaccia `PowerUpEffect` che rappresenta la strategia da implementare con tutte le sue implementazioni. Con questa soluzione rispettiamo i principi SRP e OCP, dato che potenzialmente possono essere aggiunti nuovi effetti senza modificare il codice esistente (come l'abilità di passare attraverso i muri).

Alternativa considerata: Un'altra alternativa valutata all'inizio era quella di inserire la responsabilità della creazione degli effetti nella `PowerUpFactory`, soluzione scartata per alleggerire la Factory e rispettare il SRP.

Riuso del codice di Character

Figura 2.10: Schema UML dell'applicazione del pattern *Template Method* per il riuso del codice di `Character`

Problema: Sono presenti due personaggi nel gioco, il `Player` e gli `Enemy`. E' facile notare che sono caratterizzati da molti metodi uguali portando a duplicazione di codice.

Soluzione: Seguendo il pattern *Template Method*, come da Figura 2.10 abbiamo la classe astratta `Character` che con il suo template method *update* detta i passi da implementare nelle sue sotto-classi. In particolare esse devono implementare il metodo astratto `performCharacterActions`. Con questa soluzione sfruttiamo il pattern insieme all'uso dell'ereditarietà per diminuire la dublicazione di codice e massimizzare il riuso.

Alternativa considerata: Un'altra alternativa valutata all'inizio era quella di sfruttare la composizione per andare a definire dei manager che gestissero certe parti del `Character` come il movimento e il piazzamento di bombe (movementManager e bombManager). Tale soluzione è stata scartata in quanto si voleva evitare di complicare inutilmente la struttura e avere codice meno leggibile soprattutto in caso di chiamata da classi esterne.

2.2.4 Daniele Merighi

Bomberman, come descritto precedentemente, è un gioco a tema labirintico, dove la mappa è una griglia. E' naturale che la maggior parte dei problemi

derivino dall'esplorazione dell'arena e di conseguenza dei suoi percorsi. L'obiettivo principale del nemico, però, è rimanere l'ultimo giocatore in vita. Per arrivare a ciò, bisogna che l'agent sia in grado di muoversi per la mappa, piazzare bombe in maniera consapevole, raccogliere potenziamenti per avvantaggiarsi sugli avversari e altro ancora. Essendo il tutto molto complesso, ci si è focalizzati sul riuscire a creare un nemico che abbia funzioni di base, che siano però versatili e pronte all'espansione. Di seguito si introduce lo schema UML generico, che verrà successivamente analizzato nelle sue singole parti.

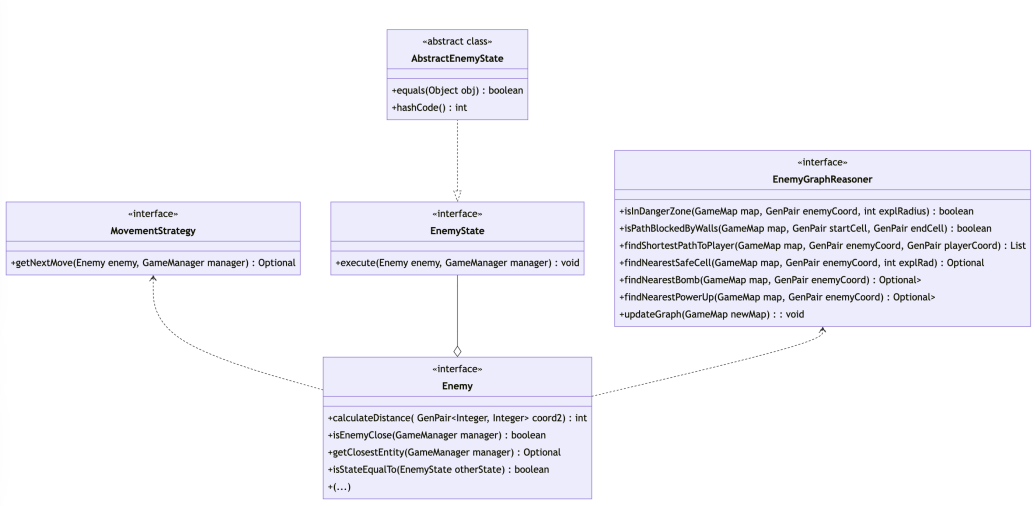

Figura 2.11: Schema UML riassuntivo della classe Enemy

1. Problema da Risolvere

Uno dei problemi principali quando si gestisce il comportamento di un nemico (classe Enemy) in un videogioco è il modo in cui prende decisioni in tempo reale in base all'ambiente circostante. L'avversario deve tenere conto di numerosi ostacoli e pericoli per potersi muovere in sicurezza per la mappa di gioco. Il suo comportamento deve adattarsi continuamente alla situazione che gli si propone, deve inoltre essere flessibile e facilmente espandibile.

1. Soluzione Proposta

Per risolvere il problema, sono stati utilizzati diversi pattern di programmazione noti per strutturare il comportamento dell'Enemy. I principali utilizzati sono i seguenti:

- **State Pattern:** utilizzato per rappresentare i vari stati del nemico (WAITING, ESCAPE, EXPLORING, PATROL, CHASE). Ogni stato decide cosa è meglio per l'entità in quel preciso momento e può transitare ad altri stati in base a determinate condizioni.
- **Strategy Pattern:** utilizzato per incapsulare diverse strategie di movimento del nemico. Questo permette di cambiare la logica senza modificare il codice stesso dell'entità.

1. Alternativa Considerata

Un'alternativa presa in considerazione era l'uso di una macchina a stati finiti (FSM) al posto dello State Pattern. Tuttavia, questa soluzione avrebbe reso il codice meno modulare e più difficile da estendere.

Il nemico utilizza il pattern State, come illustrato in figura, per gestire i diversi comportamenti. I vari stati possono essere modificati e aggiunti e questo va ad influire sul comportamento del nemico. Durante lo sviluppo sono state definite cinque differenti "personalità" :

- **Chase:** il nemico dà priorità ad inseguire l'entità a lui più vicina entro un dato raggio.
- **Waiting:** il nemico aspetta l'esplosione di una bomba, posizionandosi in una cella sicura.
- **Escape:** il nemico cerca di raggiungere una cella sicura.
- **Exploring:** il nemico raccoglie potenziamenti nelle vicinanze.
- **Patrol:** se il nemico non ha particolari obiettivi si muove in modo casuale.

Ogni stato estende la classe AbstractEnemyState, progettata per ridurre la ridondanza di codice, la quale implementa l'interfaccia EnemyState, che rappresenta lo stato generico.

Il nemico utilizza anche il pattern Strategy, come illustrato in figura, per determinare i movimenti all'interno della mappa di gioco. Questo design permette di aggiungere in futuro strategie di movimento più sofisticate, come l'attacco e la difesa, permettendo così di rendere il comportamento del nemico più dinamico e imprevedibile. Durante la fase di sviluppo sono state realizzate diverse strategie di movimento, ognuna con un suo comportamento specifico:

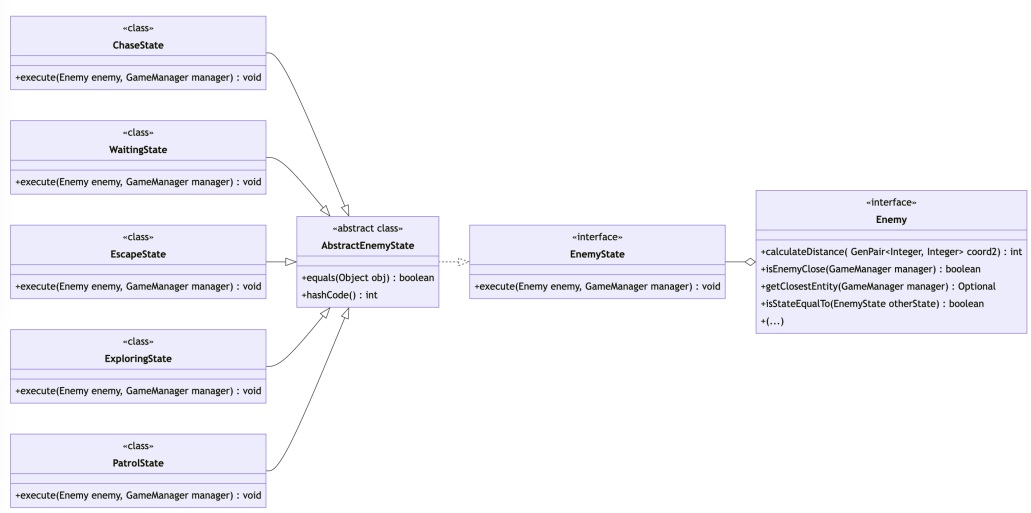

Figura 2.12: Schema UML che rappresenta i vari State dell'Enemy

- **ChaseMovementStrategy:** viene restituita la casella più vicina per raggiungere l'entità desiderata.
- **RandomMovementStrategy:** viene restituita una cella casuale adiacente.
- **EscapeMovementStrategy:** viene restituita, se esiste, la cella sicura più vicina a lui.
- **ExploreMovementStrategy:** viene restituito la posizione di un potenziamento a lui vicino.
- **ShortestMovementStrategy:** data una cella target, ti restituisce le celle che compongono il percorso minimo.

L'interfaccia MovementStrategy definisce il contratto per calcolare il prossimo movimento di un nemico. Ogni classe che implementa questa interfaccia fornisce una diversa logica di movimento.

2. Problema da Risolvere

Un aspetto fondamentale dei giochi "labirintici" come Bomberman è trovare il percorso migliore da un punto A ad un punto B. Quindi, dato un grafo G=(V,E) e un vertice sorgente s si ha bisogno di una strategia per esplorare sistematicamente gli archi di G per trovare il percorso raggiungibile da s che meglio si adatta alla nostra situazione.

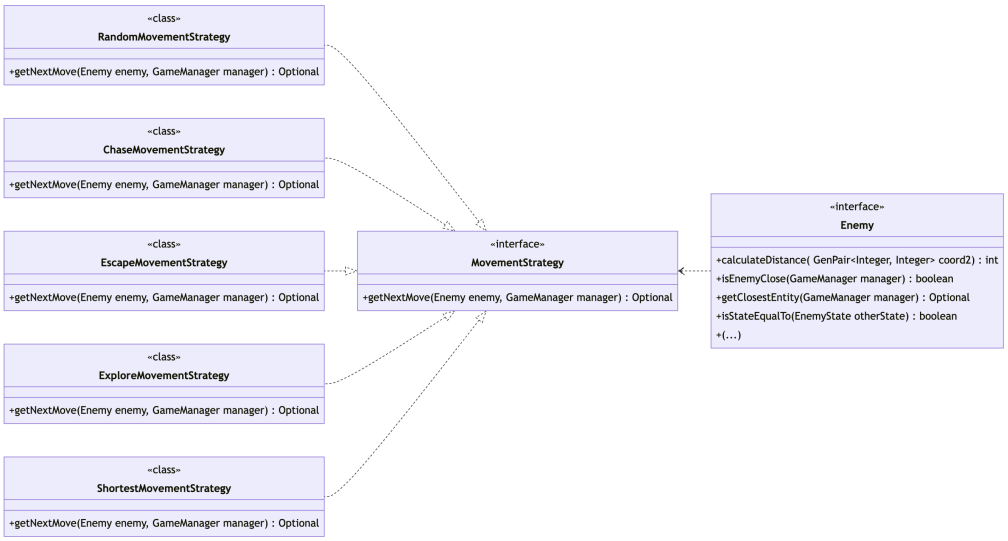

Figura 2.13: Schema UML che rappresenta i vari Movement dell'Enemy

2. Soluzione Proposta

Per risolvere il problema si è deciso di trasformare la mappa di gioco da un dizionario (che associa ad ogni coordinata (x,y) una cella) ad un grafo. La classe che gestisce ciò è GraphBuilderImpl, definita statica, dove ogni cella della mappa (tranne i muri indistruttibili) viene vista come un nodo. Questi sono collegati tra loro da archi pesati, che rappresentano un cammino che il nemico può compiere, da il nodo di origine a quello di destinazione. Solo le celle tra loro adiacenti vengono quindi collegate da un arco. Il suo costo può essere di due tipi:

- da muro distruttibile a ogni altro nodo: costo 2.5
- ogni altro collegamento: costo 1

Questa decisione è stata presa per dare maggiore importanza ai percorsi già disponibili nella mappa, invece che crearne di nuovi. Questo favorisce anche il percorso di inseguimento da parte del nemico quando vede un'altra entità.

Una volta definito il grafo, sono state scelte due tipologie di algoritmi per la sua esplorazione:

- **Breadth-First Search (BFS):** questa produce un albero con nodo radice s che contiene tutti i vertici raggiungibili. Per ogni vertice v raggiungibile da s, il cammino semplice nell'albero prodotto dalla BFS da s a v, corrisponde ad un cammino minimo da s a v in G.

- **Dijkstra:** questo trova il percorso minimo da un nodo sorgente a tutti gli altri nodi in un grafo aciclico e con pesi non negativi.

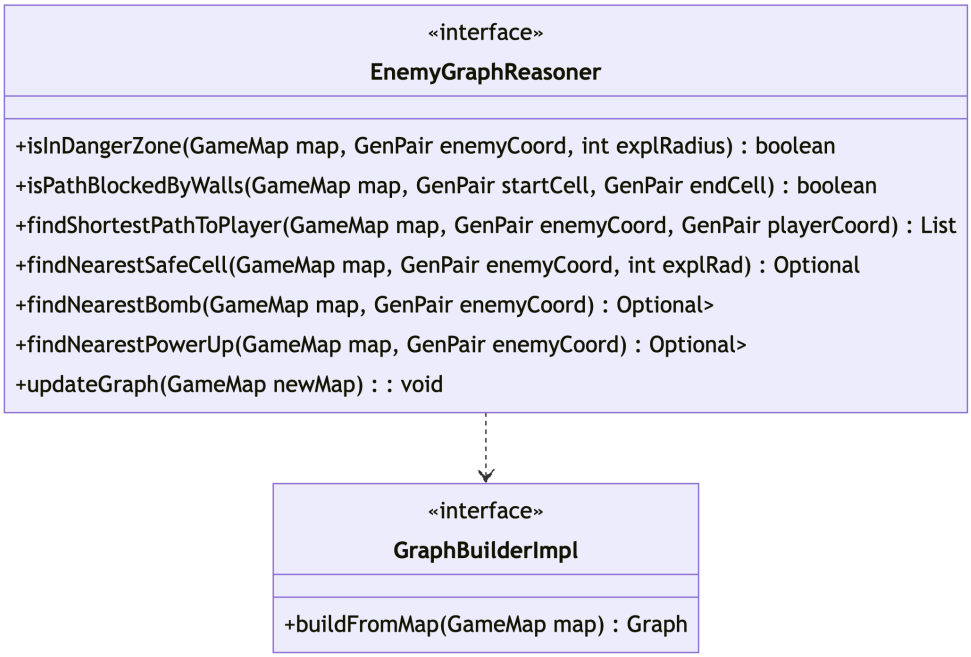

Figura 2.14: Schema UML che rappresenta la classe del Grafo

Capitolo 3

Sviluppo

3.1 Testing automatizzato

La classe `TestGraphBuilder` verifica la corretta creazione del grafo a partire dalla mappa di gioco. Di seguito sono descritti i principali casi di test:

- verifica delle dimensioni della mappa vuota (con solo muri indistruttibili)
- verifica delle dimensioni della mappa con ostacoli
- verifica dei pesi degli archi nel grafo

La classe `TestReasoner` testa le interrogazioni che vengono fatte al grafo. Di seguito sono descritti i principali casi di test:

- verifica se una cella è in una zona di pericolo (dove sta per esplodere una bomba)
- verifica se tra due celle è presente un muro
- verifica del percorso minimo tra un nemico ed il player
- verifica della cella sicura più vicina al nemico dopo l'esplosione di una bomba

La classe `TestEnemy` verifica il giusto comportamento del nemico al verificarsi di determinate situazioni. Di seguito sono descritti i principali casi di test:

- verifica se lo stato del nemico è Patrol se il player non è in un dato raggio e di conseguenza se si muove in maniera casuale

- verifica se lo stato del nemico cambia a Chase quando il player è nel raggio visivo
- verifica se il nemico "insegue" il player quando è in Chase
- verifica se il nemico in presenza di una bomba cambia il suo stato a Escape, posizionandosi poi in una zona sicura
- verifica se il nemico posiziona una bomba se tra lui e il player è presente un muro distruttibile

La classe `TestInput` verifica che la classe KeyboardInput gestisca correttamente i vari input che riceve. Di seguito sono descritti i principali casi di test:

- verifica la corretta funzione dei vari tasti (ESC, SPACE, L, P, W, A, S, D)
- verifica il corretto funzionamento di movimenti complessi
- verifica la possibilità di piazzare bombe mentre il personaggio si muove

La classe `TestPowerUp` verifica che la classe PowerUp e la sua corrispettiva factory funzionino. In particolare:

- verifica la corretta creazione dei PowerUp
- verifica che lo SkullEffect funzioni correttamente

La classe `TestPlayer` verifica che la classe Player e la sua classe astratta Character da cui eredita i metodi funzionino. In particolare:

- verifica la capacità del player di girarsi nelle quattro direzioni possibili
- verifica che il movimento del player venga eseguito correttamente
- verifica che il player sia in grado di piazzare le bombe

La classe `TestMap` verifica che l'apparato descritto nella sezione 2.2.1:

- generi un certo numero di muri indistruttibili in determinate posizioni (devono essere di una quantità fissa) ed il loro posizionamento nella mappa
- generi un certo numero di muri distruttibili (in funzione della percentuale sul terreno libero che viene impostata dal gioco)

- non generi muri distruttibili in corrispondenza delle tre celle ad ogni angolo della mappa (un blocco in questi punti non darebbe possibilità di piazzare bombe in sicurezza al giocatore)

La classe `TestBomb` : verifica che la BombFactoryImpl e le Bomb funzionino nel modo corretto

- verifica l'interazione dell'esplosione con vari tipi di Cell
- verifica il corretto piazzamento dei diversi tipi di Bomb
- verifica gli effetti di tutti i tipi di Bomb

La classe `TestCollision` : verifica le classi RectangleBoundingBox, BombarderoCollision e CollisionHandler

- verifica che le collisioni con un ostacolo siamo riscontrate in tutte e 4 le direzioni e successivamente risolte
- verifica quando un personaggio si trova sopra ad una Cell di tipo Powerup o Flame

per il testing specifico della risoluzione delle collisioni si è preferito fare test di persona così da poter decidere la grandezza delle BoundingBox

3.2 Note di sviluppo

3.2.1 Federico Bagattoni

Utilizzo di `Lambda Expression`

Utilizzo di `Lambda Expression` in diversi punti, ad esempio per la creazione delle procedure nella guida interattiva. Si veda: `https://github.com/DanieleMerighi/OOP23-bombardero/blob/b621c490e0f6aebeb2d828b960f0263de4801cb0/src/main/java/it/unibo/bombardero/core/impl/BombarderoGuideManager.java#L60-L80`

Utilizzo di Java `Stream`

Utilizzo di Java `Stream`, per esempio per la generazione delle posizioni dei muri. Si veda: `https://github.com/DanieleMerighi/OOP23-bombardero/blob/b621c490e0f6aebeb2d828b960f0263de4801cb0/src/main/java/it/unibo/bombardero/map/impl/MapGeneratorImpl.java#L49-L58`

Utilizzo di `functional interfaces`

Utilizzo di functional interfaces come `BiPredicate` e `BiConsumer`, per esempio nella creazione delle procedura della guida interattiva. Si veda: `https://github.com/DanieleMerighi/OOP23-bombardero/blob/b621c490e0f6aebeb2d828b960f0263d`
`src/main/java/it/unibo/bombardero/core/api/GuideStep.java#L15`

Algoritmo di traversal in spirale di una matrice

Trovo moralmente giusto menzionare, in quanto non di mia creazione, la fonte dell'algoritmo di traversal di una matrice in forma spirale che viene usato nella generazione del `collapse order`, cito la fonte che ho usato anche se le prime ricerche danno tutte lo stesso risultato:
Permalink allo snippet nel progetto: `https://github.com/DanieleMerighi/OOP23-bombardero/blob/3cd34d82edae666c7312e2901513517cadcdd165/src/main/java/it/unibo/bombardero/map/impl/SpiralTraversalStrategy.java#L15-L44`
Link al sito web da cui si è preso il codice: `https://takeuforward.org/data-structure/spiral-traversal-of-matrix/`

Riadattamento della classe `BombarderoEngine` a partire da una già esistente

Per lo sviluppo della classe `BombarderoEngine` è stata riadattata la classe `GameEngine` sviluppata dal Prof. Alessandro Ricci durante il seminario *Game Programming Patterns in Java* nell'a.a. 2022-2023.
Permalink al seminario: `https://github.com/pslab-unibo/oop-game-prog-patterns-2022/blob/master/step-01-game-loop/src/rollball/core/GameEngine.java#L1-L71`
Permalink a `BombarderoEngine`: `https://github.com/DanieleMerighi/OOP23-bombardero/blob/b621c490e0f6aebeb2d828b960f0263de4801cb0/src/main/java/it/unibo/bombardero/core/BombarderoEngine.java#L37-L65`

3.2.2 Luca Venturini

Utilizzo di Stream Java

Permalink: `https://github.com/DanieleMerighi/OOP23-bombardero/blob/43c0418f98602e303e34baf20629ccd1dd0a7c1b/src/main/java/it/unibo/bombardero/bomb/impl/BasicBomb.java#L115-L140`

Utilizzo di generici e Functional Interface

Permalink: https://github.com/DanieleMerighi/OOP23-bombardero/blob/a94a829f9745cfda90a3125da40c183ad68c571e/src/main/java/it/unibo/bombardero/map/api/GenPair.java#L46-L57

Functional Interface e Lambda expression

Permalink: https://github.com/DanieleMerighi/OOP23-bombardero/blob/a94a829f9745cfda90a3125da40c183ad68c571e/src/main/java/it/unibo/bombardero/map/api/Functions.java#L22-L85

3.2.3 Jacopo Turchi

Uso di lambda expression in vari punti, in particolare per la Strategy, combinato all'uso di Runnable, Method Reference e Optional

Permalink: https://github.com/DanieleMerighi/OOP23-bombardero/blob/43c0418f98602e303e34baf20629ccd1dd0a7c1b/src/main/java/it/unibo/bombardero/cell/powerUp/impl/SkullEffect.java#L54C9-L72C11

Uso di stream in vari punti, per esempio:

Permalink: https://github.com/DanieleMerighi/OOP23-bombardero/blob/43c0418f98602e303e34baf20629ccd1dd0a7c1b/src/main/java/it/unibo/bombardero/cell/powerUp/impl/PowerUpImpl.java#L41-L62

Uso di Supplier e Consumer:

Permalink: https://github.com/DanieleMerighi/OOP23-bombardero/blob/43c0418f98602e303e34baf20629ccd1dd0a7c1b/src/main/java/it/unibo/bombardero/cell/powerUp/impl/PowerUpFactoryImpl.java#L64C9-L68C67
Permalink: https://github.com/DanieleMerighi/OOP23-bombardero/blob/43c0418f98602e303e34baf20629ccd1dd0a7c1b/src/main/java/it/unibo/bombardero/cell/powerUp/impl/PowerBombEffect.java#L27C5-L29C6

Uso della libreria Apache Commons

Permalink: https://github.com/DanieleMerighi/OOP23-bombardero/blob/43c0418f98602e303e34baf20629ccd1dd0a7c1b/src/main/java/it/unibo/bombardero/cell/powerUp/impl/PowerUpEnumeratedDistribution.java#L25-L40

3.2.4 Daniele Merighi

Utilizzo della libreria jgrapht.Graph

Utilizzata in vari punti, un esempio è: https://github.com/DanieleMerighi/OOP23-bombardero/blob/a94a829f9745cfda90a3125da40c183ad68c571e/src/main/java/it/unibo/bombardero/character/AI/impl/GraphBuilderImpl.java#L45-L57

Utilizzo della libreria jgrapht.traverse

Permalink: https://github.com/DanieleMerighi/OOP23-bombardero/blob/a94a829f9745cfda90a3125da40c183ad68c571e/src/main/java/it/unibo/bombardero/character/AI/impl/EnemyGraphReasonerImpl.java#L229-L241

Utilizzo delle `Stream` Java

Utilizzando dove è stato possibile, il seguente è un singolo esempio: https://github.com/DanieleMerighi/OOP23-bombardero/blob/a94a829f9745cfda90a3125da40c183a
src/main/java/it/unibo/bombardero/character/AI/impl/EnemyGraphReasonerImpl.java#L75C5-L85C6

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Federico Bagattoni

Sono particolarmente contento della collaborazione e della riuscita del progetto. Essendo una persona competitiva, ma per le cose inutili, tenevo particolarmente alla riuscita della parte grafica e della user experience che in certi punti mi ha lasciato amareggiato a causa dei ritardi nella settimana antecedente la consegna. Per quanto riguarda la parte di modello avrei sicuramente svolto una notevole fase di progettazione prima di iniziare a scrivere codice, quando in realtà questa strada non è stata seguita. Complessivamente, essendo al primo progetto di queste dimensioni, mi ritengo soddisfatto unicamente per averlo portato a termine con un risultato che mi aggrada, considerando anche altre attività al di fuori dell'università che dovevo continuare a perseguire.
Di questo progetto mi rimane la voglia ad avere più pazienza ed essere più teorico nel mesterie dell'ingegnere

4.1.2 Daniele Merighi

In qualità di membro del gruppo responsabile dello sviluppo del nemico, ritengo di aver raggiunto i requisiti inizialmente prefissati in fase di progettazione. Tuttavia essendo la prima volta che realizzo un progetto basato su un videogioco ho dovuto affrontare problematiche concettualmente nuove. Sicuramente il codice attualmente presente è una buona base su cui partire per lo sviluppo di un'entità più complessa, ma se dovessi ripensare alla classe Enemy la strutturerei diversamente. Il nemico è progettato per essere espandibile e riusabile ma adotterei tecniche più evolute per la sua parte

comportamentale, adoperando ad esempio un Decision Tree. Anche il grafo lo penserei in maniera differente, creando dei "sotto grafi" specifici per ogni azione, ad esempio uno con solo il suo campo visivo, uno con solo le zone di pericolo ecc. Sarebbe opportuno che l'agent avesse sempre un suo obiettivo andando a ridurre al minimo, se non zero, la parte di movimento casuale. Dopo aver affrontato il corso di metodi numerici si intende proseguire il progetto andando ad implementare anche una parte di Deep Learning per rendere il nemico una vera e propria AI.

4.1.3 Jacopo Turchi

Inizio con lo scrivere che questo progetto è stato formativo in quanto non avevo mai progettato in gruppo e soprattutto non avevo mai programmato ad oggetti e infatti mi è risultato difficile partire da zero. Però col tempo ho preso il mio ritmo e sono riuscito a fare la mia parte di questo gioco che ha segnato la mia infanzia. Sono molto contento del prodotto ottenuto e mi ritengo soddisfatto soprattuto di come si sia evoluto il mio codice col procedere del progetto. Partendo da una banale estensione della classe Character, fino ad arrivare ad utilizzare anche diversi pattern di programmazione per i PowerUp, che ritengo essere il mio punto forte. Un mio punto debole è stato partire un po' in ritardo e non contribuire molto alla struttura iniziale del progetto, ma sono riuscito a portare una boccata d'aria nello sviluppo quando, a stato avanzato, l'umore generale non era alle stelle. Potessi tornare inidetro probabilmente non rifarei alcune delle prime scelte di programmazione, ma tutto sommato mi ritengo soddisfatto del mio lavoro

4.1.4 Luca Venturini

Personalmente sono soddisfatto del mio lavoro svolto all'interno di questo gruppo sia a livello di codice sia a livello di collaborazione con gli altri membri del gruppo, soprattutto nelle ultime settimane dove il lavoro si è intensificato. Sono consapevole della mia inesperienza nel programmare videogiochi, proprio per questo sono convinto che alcune cose potrebbero esser state fatte in maniera migliore per esempio l'analisi iniziale che a parer mio non e stata poi ottimale per lo sviluppo , d'altra parte sono contento di come ho sviluppato alcune feature come le collisioni. Lavorare in gruppo mi ha sicuramente spinto a lavorare più seriamente al progetto visto che non sei responsabile solo del tuo risultato ma di anche quello degli altri, proprio per questo ho cercato di essere il più disponibile possibile.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Federico Bagattoni

Le principali difficoltà che io sento di aver incontrato durante il progetto sono:

- la fretta nell'iniziare la scrittura del codice (e soprattutto la **voglia**)
- la **poca voglia** (al contrario del punto precedente) nel voler fare una adeguata analisi e progettazione
- senso di impotenza scaturito dall'imperante regolamento e formalismi che talvolta possono intimorire lo studente nella fase iniziale

Interessante notare invece che col procedere del progetto il focus cambia rispetto alle tre condizioni sopracitate. Infatti la fretta di scrivere codice si placa e la necessità di una maggiore progettazione affiora, notando l'accumularsi di difetti di progettazione che si devono aggirare tramite numerosi refactor. Mentre il senso di impotenza (almeno per me) si è trasformato, una volta capito quale era l'ambiente, in una grande volontà di voler essere preciso, formale e teorico.

Tirando le somme percepisco che durante il corso si è affermata nella mia mente (e sono sicuro anche in quella di molti altri studenti) non la figura del "Corso di *Programmazione (**progettazione e design**) ad oggetti*" quanto più il "Corso di *Java*" ed è per questo che valuto (e rimpiango), solo alla fine, gli aspetti progettuali di questo corso che il corpo docente insegna con una notevole passione.

Appendice A

Guida utente

Il menu di gioco mostra all'utente due possibilità: iniziare una partita o intraprendere la guida.
Incoraggiamo fortemente un principiante a seguire il secondo punto al fine di comprendere in prima persona il funzionamento del gioco.
Tuttavia è opportuno leggere ugualmente l'intera appendice in quanto la guida utente non contiene la spiegazione di meccaniche di gioco avanzate.

A.1 Match

All'inizio di un match si è generati nell'angolo in alto a sinistra di una mappa quadrata. L'obbiettivo del gioco è rimanere l'ultimo giocatore in vita eliminando gli altri giocatori attraverso l'esplosione di una bomba.
Premendo W, A, S, D il giocatore può muoversi. Tramite la barra spaziatrice space è possibile piazzare una bomba. Ogni giocatore, all'inizio della partita, può piazzare una sola bomba mentre il giocatore può allargare questo spazio con l'uso di potenziamenti.
Detonare le bombe vicino alle casse permetterà di distruggerle fornendo spazio di movimento al giocatore e da la possibilità alla apparizione di un potenziamento (powerup).
I powerup si presentano come dei cubi verdi con un immagine centrata al loro interno. Permettono di potenziare o depotenziare le statistiche del giocatore oppure forniscono una *abilità speciale* come per esempio la *line bomb* oppure la *remote bomb*, spiegate nella sezione A.1.1. I powerup danno il loro effetto sul giocatore nel momento in cui sono raccolti. Per i powerup che forniscono una *abilità speciale* se ne viene raccolto un altro della stessa categoria, il precedente viene sovrascritto.
Un powerup da cui si deve stare in guardia è lo `Skull` (v. figura A.1) che

dona uno svantaggio casuale al player per un periodo limitato di tempo, tra cui: velocità estremamente alta o bassa, piazzamento di bombe involontario e costante mentre il player cammina o l'impossibilità di piazzarle a piacimento.

Figura A.1: Powerup Skull nel gioco

Dopo 2 minuti la mappa inizierà a **collassare** partendo dall'angolo in alto a sinistra e seguendo una forma a spirale, se si è colpiti da un muro si viene eliminati.

Figura A.2: Collasso dell'arena a seguito dello scadere del tempo, il personaggio in basso a sinistra è stato appena colpito da un muro

A.1.1 Powerup con comportamento particolare

E' opportuno segnalare all'utente che i seguenti powerup hanno un comportamento particolare che potrebbe non essere compreso immediatamente:

- Remote bomb: le bombe piazzate sono **telecomandate** e vengono detonate esclusivamente premendo il tasto P

Figura A.3: Remote bomb powerup

- Line bomb: premendo il tasto L è possibile piazzare tutte le bombe che si ha nell'inventario (a meno di incontrare un ostacolo) nella direzione in cui si sta guardando. E' comunque possibile piazzare una bomba normale premendo Space

Figura A.4: Line bomb powerup

- Piercing bomb: la fiamma generata dalla bomba piazzata distruggerà **tutti** i muri che incontra fino a raggiungere il suo range massimo, distruggendo eventualmente tutti i powerup generati da questi muri

Figura A.5: Piercing bomb

Appendice B

Esercitazioni di laboratorio

Nessuno studente ha consegnato i laboratori di programmazione ad oggetti.