Relazione
“SMOL”

Ettore Farinelli
Marco Galeri
Giovanni Paradisi
Mounir Samite

8 aprile 2023

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il gruppo si pone come obiettivo quello di realizzare un videogioco 2D arcade con visuale dall'alto di nome "SMOL". Per arcade si intende una struttura di gioco ripetitiva in cui l'obbiettivo è accumulare più punti possibile.

Requisiti funzionali

- Il software dovrà essere in grado di gestire l'ambiente di gioco durante l'intera durata della partita elaborando le interazioni tra le diverse entità del gioco.
- L'obbiettivo del gioco è difendere i campi di ortaggi dalle talpe che invadono la mappa, schiacciandole con un martello. I campi subiranno danni anche in caso il giocatore li attraversi.
- Con il progredire dello score la difficoltà di gioco aumenterà, diminuendo il tempo tra la creazione di talpe e modificando la probabilità con cui si generano diversi tipi di talpa.
- Durante la partita sarà possibile effetuare diverse operazioni:
  - Il giocatore potrà muovere il personaggio, attraverso i tasti direzionali, all'interno della mappa.
  - Usando il mouse invece, controllerà la direzione del martello che attraverso una pressione prolungata potrà aumentare il raggio di azione.
- Il gioco termina quando tutti gli ortaggi risultano distrutti.

Requisiti non funzionali

- Si avrà la possibilità di cambiare l'aspetto grafico scegliendo tra quelli proposti nell'interfaccia del menù.
- Il programma sarà in grado di salvare il miglior punteggio in locale.
- Il gioco avrà una sezione in cui verrano visualizzate le istruzioni di gioco.
- Implementazione di una pagina di game over che permette di tornare al menù senza chiudere l'applicazione e mantenendo lo stesso pacchetto grafico precedentemente utilizzato.

1.2 Analisi e modello del dominio

In SMOL il sofware dovrà essere in grado di controllare le interazioni tra diverse tipologie di entità. Tra le entità saranno presenti:

- Ortaggi: queste entità fungeranno come vita del giocatore ed a contatto con altre entità perderanno vita
- Giocatore: questa entità sarà controllata attraverso la tastiera e permetterà di spostarsi all'interno della mappa.
- Martello: questa entità sarà controllata attraverso il mouse che, con una pressione prolungata permetterà di aumentare il proprio raggio di azione, mentre al rilascio, se si troverà a contatto con le talpe, gli toglierà vita.
- Talpe: questa entità fungerà da nemico principale del giocatore, spostandosi attraverso la mappa cercherà di mangiare gli ortaggi presenti ed in caso entri in contatto col giocatore gli bloccherà temporaneamente i movimenti.
- Muri: impediranno al giocatore di uscire dai confini della mappa di gioco.

Il sofware sarà responsabile anche della creazione iniziale degli ortaggi e la generazione delle talpe durante il resto della partita attraverso un sistema che incrementerà in base al punteggio corrente del giocatore il numero di talpe generate nel tempo, insieme alla probabilità di generare talpe più complicate da abbattere. Gli elementi costitutivi del dominio sono sintetizzati in Per rientrare nel monte ore richiesto non è stato possibile implementare

una modalità di multiplayer locale, che sarà possibilmente implementata in futuro.

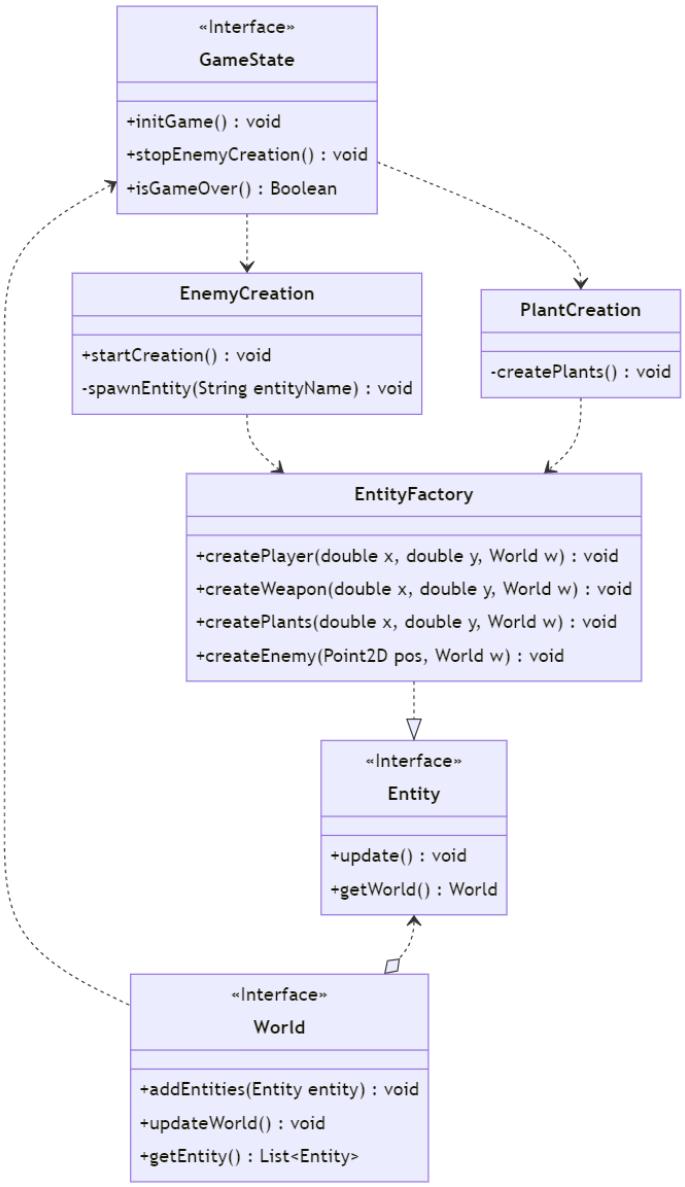

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

SMOL è stato progettato seguendo lo schema architetturale del MVC applicando però la struttura dell'EC (*Entity Component*) per l'organizzazione dell'entità.
L'utilizzo di component associati alle entità, consente di semplificare e suddividere quest'ultime permettendo di aggevolarne la gestione utilizzando un solo metodo di *update* per aggiornare tutti i component semplificandone la gestione. Applicando l'EC al *Model* dell'applicazione, si avrà che ogni elemento risulterà essere un entità che fungerà da "metodo di comunicazione" per i propri component permettendo ad essi di comunicare tra loro indirettamente, evitando così ripetizione di codice e permettendo di aggiungere facilmente qualsiasi tipo di oggetto al gioco facendolo passare come entità e ponendo opzionali i vari component.
Per quanto riguarda il patter architetturale *Model-View-Controller*, le varie parti sono già visivamente suddivise in diverse cartelle a livello di codice, ma vengono principalmente rappresentate da:

- **Model:** L'interfaccia *World* assume il principale ruolo di model all'interno del programma contenendo la maggior parte degli oggetti generali dell'applicazione.

- **View:** Questo ruolo è principalmente occupato dall'interfaccia WindowState che viene implementata da tutte le diverse scene del gioco, inoltre risulta avere un ruolo molto importante anche *GraphicsDraw* e *LoadImgs* che hanno il ruolo di contenere tutte le immagini e successivamente disegnarle.

- **Controller:** A ricoprire questo ruolo troviamo l'interfaccia *GameState* che attraverso i suoi metodi notifica i cambiamenti che avvengono nella *View* al *Model*.

Le diverse modalità che permettono di comunicare alle varie parti che caratterizzano il modello MVC sono rappresentati qui sotto2.1.

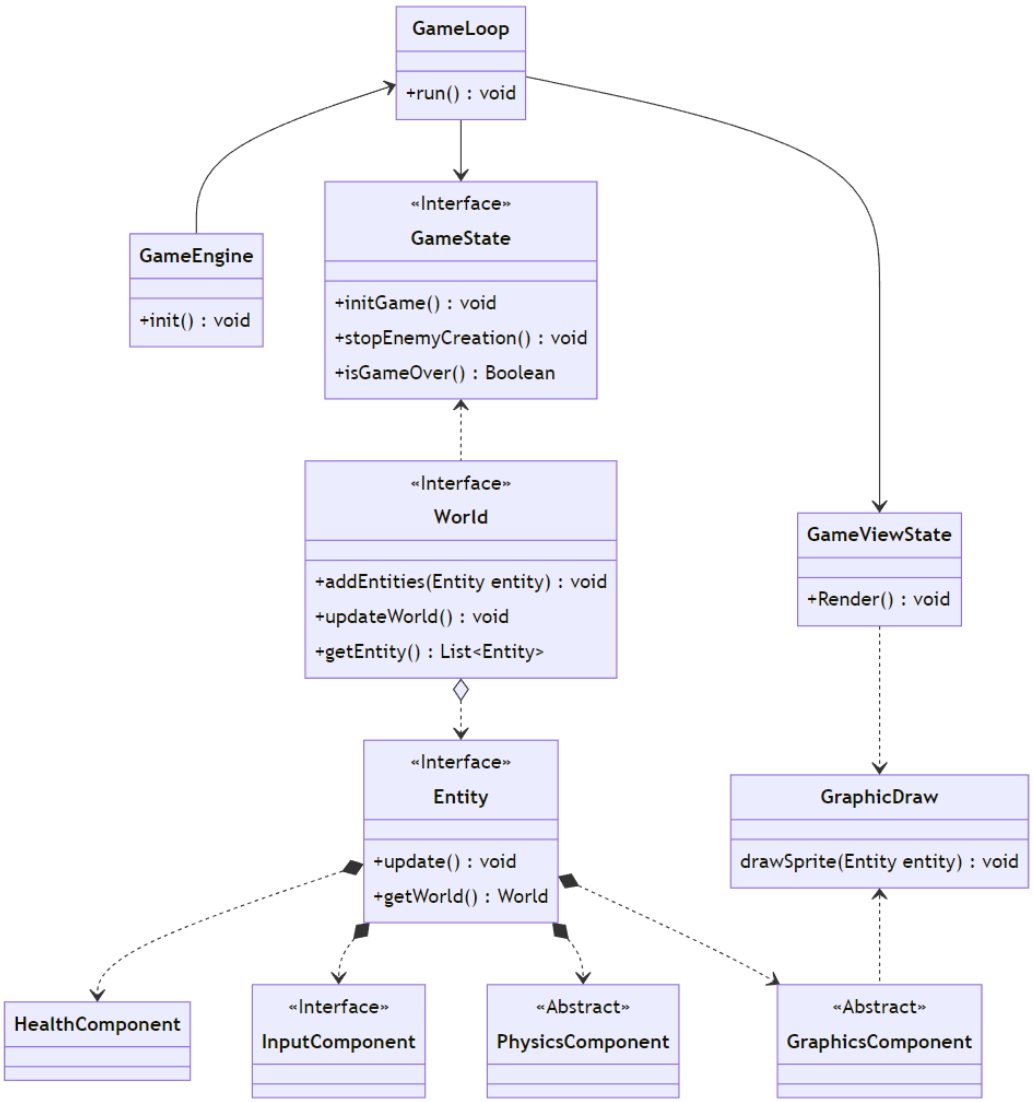

Figura 2.1: Schema UML raffigurante l'architettura di SMOL.

2.2 Design dettagliato

Ettore Farinelli

Implementazione degli input del giocatore

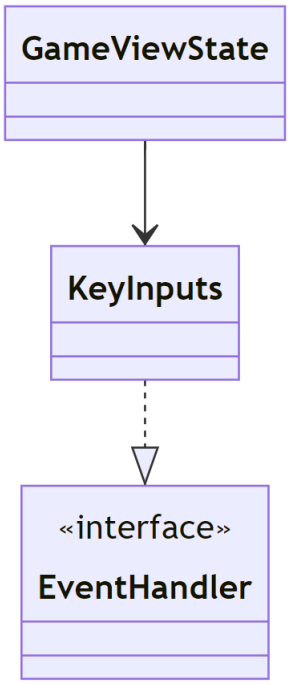

Figura 2.2: Schema UML rappresentante il ricevimento degli input della classe *KeyInputs*.

Problema Un input inserito da tastiera deve essere elaborato e immagazzinato cosi da permettere al player di muoversi.

Soluzione Ogni input inserito da tastiera viene ricevuto e passato alla classe *KeyInputs* che lo filtra e in base al tipo inserisce una *Direction* all'interno di una coda che verrà successivamente svuotata un elemento alla volta dall'*InputComponent* (in questo caso dal *PlayerInputComponent*) ricevendo così le direzioni che verranno applicate all'entità stessa permettendole di muoversi nella mappa di gioco, Come da figura2.3.

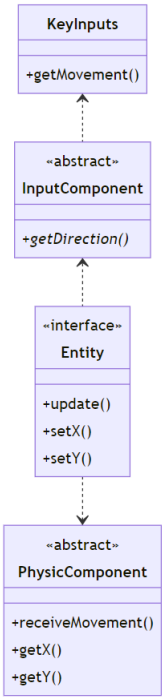

Figura 2.3: Schema UML che rappresenta in che modo un input venga elaborato e ricevuto da un entità.

Gestione del martello

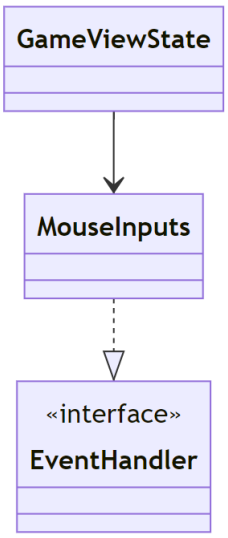

Figura 2.4: Schema UML rappresentante il ricevimento degli input della classe *MouseInputs*.

Problema Il martello dovrà avere una posizione risultante distante dal giocatore di una misura uguale al raggio della circonferenza di diamentro WEAPONRANGE e in una posizione presente nella retta tracciata dal centro del giocatore al cursore del mouse. Inoltre quando il martello viene rilasciato dovrà bloccare gli input per permettere al player (virtualmente) di colpire con il martello la talpa su cui il bersaglio si trova.

Soluzione Come per il problema degli input da tastiera, quelli da mouse vengono ricevuti e elaborati in maniera simile con una differenza sostanziale. Infatti, *MouseInputs* salva direttamente la posizione del cursore quando viene mosso o rilasciato, che viene quindi ricevuta dall'*InputComponent* e poi elaborata nel *PhysicComponent* dove vengono quindi fatti tutti i calcoli [1] per trovare la effettiva posizione in cui il martello colpirà al rilascio2.5. Sempre al rilascio del tasto sinistro del mouse, per bloccare gli input da tastiera e da mouse inizialmente ho pensato di utilizzare dei metodi, e quindi campi, statici accessibili sia da *MouseInputs* che da *KeyInputs*, infine poi, a seguito

[1] `https://github.com/TheDarkRuler/OOP22-SMOL/blob/757228125cb2531fccca35b35f7645ceba9d5ae9/src/main/java/it/unibo/smol/model/impl/physicscomponent/WeaponPhysicsComponent.java#L46-L51`

di diverse migliorie ho optato per includere un campo di tipo *KeyInputs* all'interno di *MouseInputs* così da poter notificare il blocco degli input da una classe all'altra.

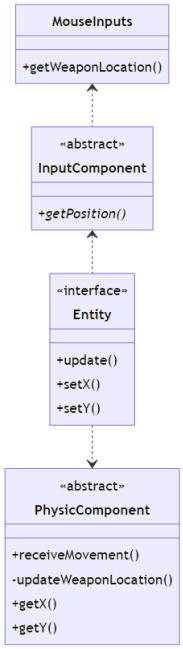

Figura 2.5: Schema UML che rappresenta in che modo un input venga elaborato e ricevuto da un entità.

Creazione AI talpe

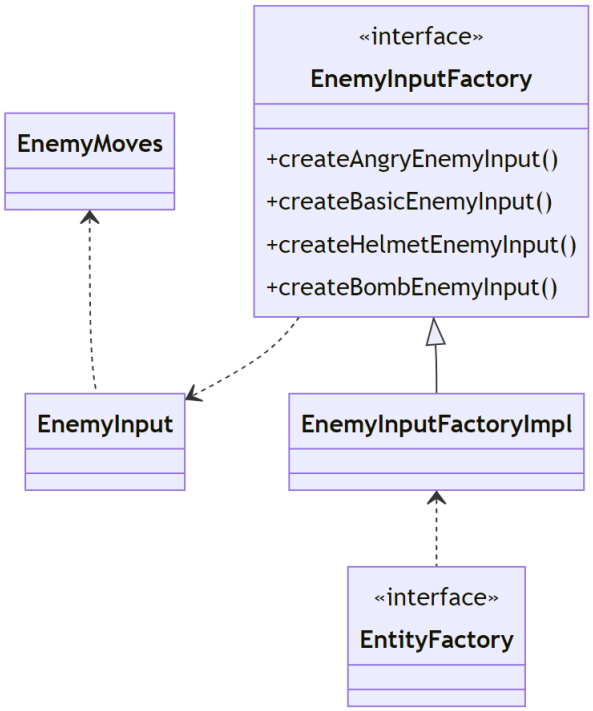

Figura 2.6: Schema UML rappresentante lo schema di ereditarietà della classe EnemyInput.

Problema Creare un intelligenza artificiale che scelga una posizione, dove non siano presenti altre entità al momento, in modo pseudo-casuale e che sia adattabile a 4 diversi tipi di entità. Inoltre essa deve effettivamente muoversi nella mappa in maniera uniforme.

Soluzione Per creare un tipo di intelligenza artificiale che stimoli il più possibile il giocatore con movimenti casuali e variabili ho deciso di dare alle entità nemiche un tipo di movimento che divida la mappa di gioco in quattro quadranti e casualmente scelga uno dei quadranti dove, ancora una volta casualmente scelga una posizione in esso non occupata da altre entità, così che la posizione futura di una talpa sia prevedibile il meno possibile. Per il problema del movimento uniforme della talpa ho deciso di effetuare dei calcoli per trovare il giusto passo che essa deve fare per muoversi nell'asse delle

ascisse e in quello delle ordinate [2] che le viene applicato dall'*InputComponent* attraverso l'*Entity* permettendole di modificare a mano a mano la sua posizione. Infine per la creazione degli input inizialmente ho creato una classe che veniva estesa da altre classi che definivano ognuno un tipo di talpa, in seguito rianalizzando il codice, ho notato di poter attuare il *Factory Pattern*, dando così la possibilità di creare i 4 tipi di input attraverso una sola classe rendendo più comprensibile e intuitiva la creazione di input nella *EntityFactory*, inoltre facilitando la possibile implementazione futura di un nuvo tipo di nemico.

Gestione delle collisioni

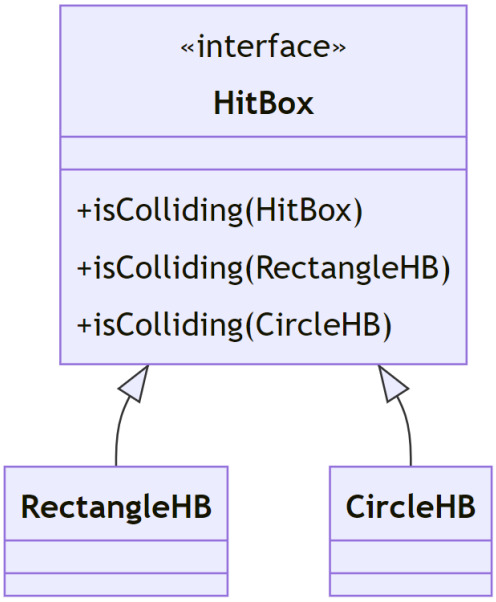

Figura 2.7: Schema UML rappresentante l'utilizzo del *Visitor Pattern* per il controllo delle collisioni tra diverse forme.

Problema L'oggetto su cui viene chiamato il metodo isColliding ha bisogno di conoscere di che tipo di HitBox è il parametro che gli viene passato.

Soluzione Per la soluzione di questo problema ho deciso di implementare il pattern *Visitor* permettendo così di facilitare una possibile implementazione futura di una nuova HitBox di forma diversa, in quanto il controllo

[2] `https://github.com/TheDarkRuler/OOP22-SMOL/blob/3d89bb7d9ef1c36e091a01eed47fa5c9fcbe6486/src/main/java/it/unibo/smol/controller/input/EnemyMoves.java#L68-L70`

della collisione viene affidato all'oggetto che viene passato alla chiamata del metodo, evitando cosi controlli manuali riguardanti lo stato del parametro.

Marco Galeri

Gestione delle Entità

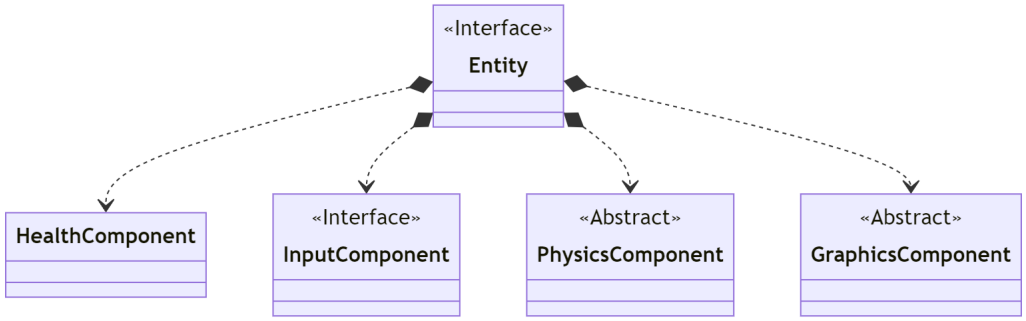

Figura 2.8: Rappresentazione UML di una entità e i suoi componenti

Problema L'interfaccia ENTITY deve implementare numerosi comportamenti diversi, l'uso dell'ereditarietà porterebbe alla creazione di molteplici sottoclassi ed a possibili problemi di eredità multipla aumentando la complessità del codice. Inoltre la classe ENTITY incapsula diversi aspetti del gioco violando il *Single Responsability Principle*.

Soluzione Adottando l'*Entity Component System*, ampiamente usato nell'ambito di Game Development, si ha una scissione dei vari aspetti di gioco in COMPONENTS. L'unione dei COMPONENTS quindi definisce il comportamento finale di ogni Entità. Con questo design che favorisce la composizione al posto dell'ereditarietà si ha un aumento di flessibilità del codice che viene suddiviso in classi più corte e meno complicate. Inoltre i componenti sono altamente riusabili rendendo plausibile la creazione di diverse tipologie di entità e lasciando libera la possibilità di aggiunte future.

Creazione delle Entità

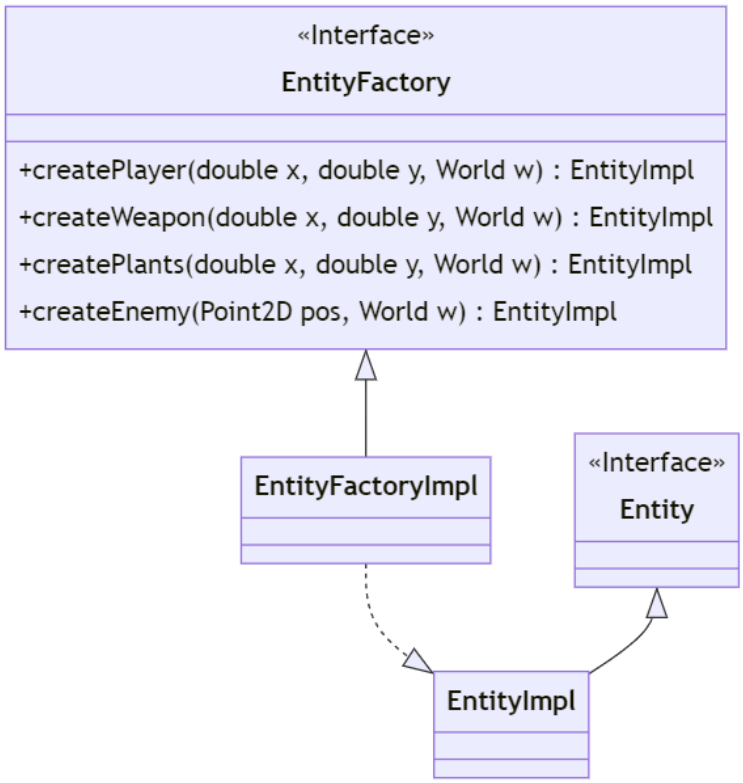

Figura 2.9: Rappresentazione UML della creazione di una entità attraverso l'utilizzo del Factory method

Problema Date le molteplici tipologie di componenti che una Entità può avere, la creazione di un'Entità direttamente all'interno del mondo è possibile ma poco flessibile e funzionale.

Soluzione Impiegando il *Factory design pattern*, la creazione dell'entità viene esternalizzata in una classe. Con questa classe, EntityFactory, si possono creare dei set di componenti predefiniti da usare per istanziare ogni tipo di entità. Attraverso questo metodo si diminuisce notevolmente la duplicazione di codice, inoltre si rimuove la responsabilità di creazione delle entità dalla classe chiamante.

Gestione del movimento e risoluzione delle collisioni tra Entità

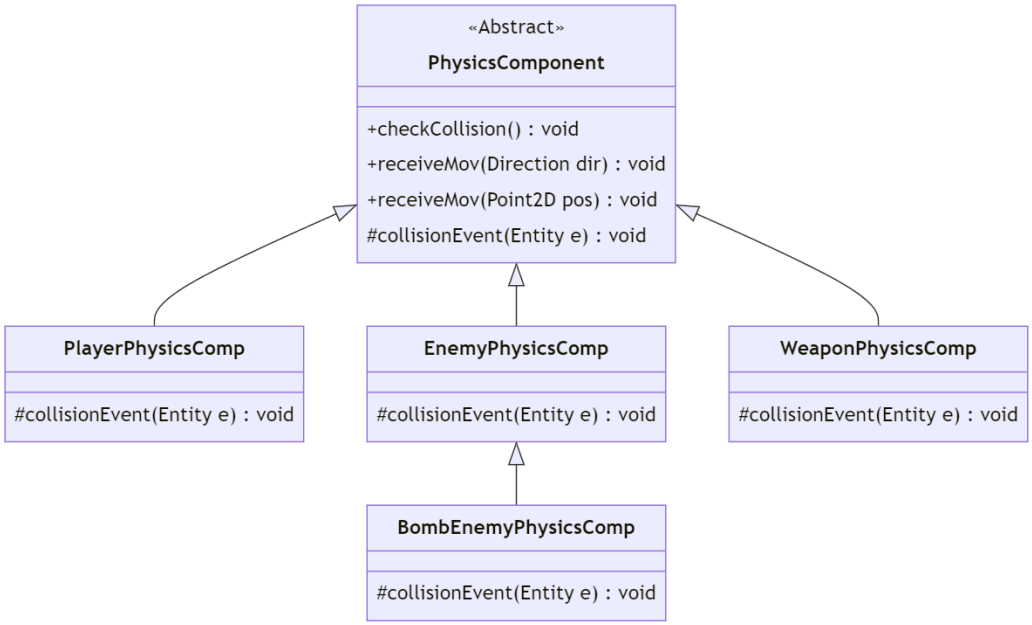

Figura 2.10: Rappresentazione UML delle diverse interazioni tra Entità

Problema

- Entità diverse ricevono comandi di movimento differenti rendendo complicato creare una interfaccia generale che gestisca la fisica delle entità.
- Le collisioni devono produrre eventi differenti in base alle tipologie di Entità che stanno collidendo.

Soluzione Utilizzando un *Template method pattern* si ha la possibilità di definire una struttura generale della classe, lasciando alle sottoclassi la scelta di dettagli specifici. Questo riduce drasticamente la duplicazione di codice in quanto i metodi template `checkCollision()` e `receiveMovement()` vengono scritti unicamente nella classe madre. Inoltre tramite l'overload del metodo `receiveMovement()` viene generalizzata anche la funzione di movimento.

Gestione di input multipli

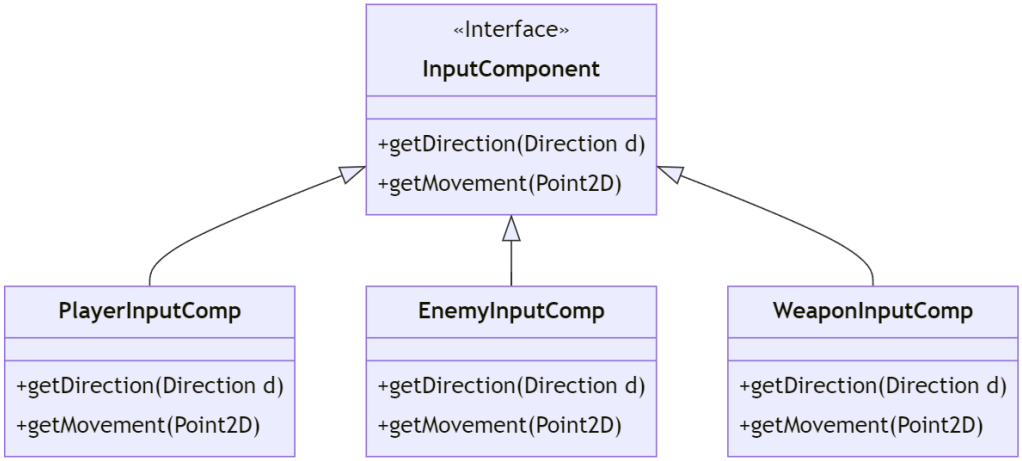

Figura 2.11: Rappresentazione UML della gestione degli InputComponent

Problema

- L'utilizzo di diverse implementazioni di input richiede la presenza di una interfaccia generale che li gestisca
- L'implementazione di nuove periferiche di gioco può risultare scomodo e macchinoso

Soluzione L'utilizzo di uno *Strategy pattern* consente di isolare le diverse implementazioni di InputComponent randendole facilmente intercambiali e propense ad aggiunte future.

Giovanni Paradisi

Gestione della grafica delle entità

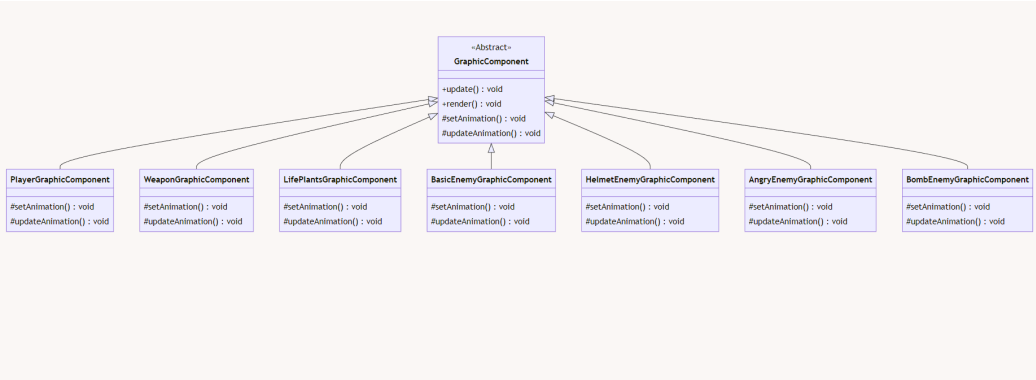

Figura 2.12: Rappresentazione UML dei GraphicComponent

Problema

- Gestire le diverse immagini delle entità in base al loro comportamento senza dover riscrivere più volte il codice in comune.

Soluzione Grazie all'uso del *Template method pattern* si definiscono dei metodi template come `setAnimation()` e `updateAnimation()` nella classe astratta e implementati nelle sottoclassi in modo da soddisfare le singole esigenze, mantenendo il codice privo di ripezioni.

Generazione delle Entità Ostili

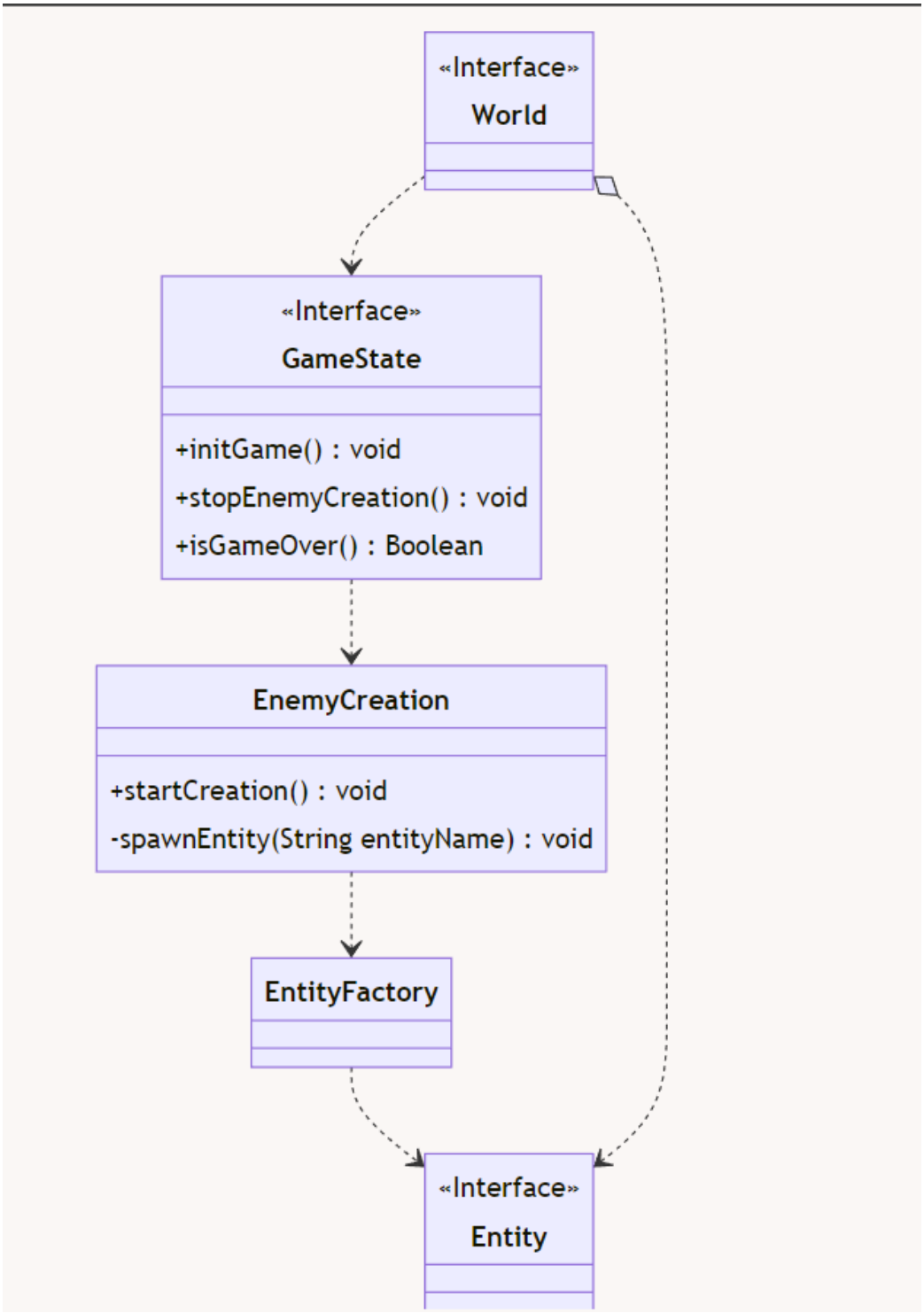

Figura 2.13: Rappresentazione UML di EnemyCreation

Problema

- Gestire la generazione continua e progressiva di entità nemiche.

Soluzione Viene creato un timer con un intervallo di tempo (tmin e tmax) tra i quali estrae il delay per la creazione delle entità, e una mappa che associa ad ogni nemico una percentuale di spawn. Con l'avanzamento del punteggio di gioco cambiano sia il delay che la percentuale dell'entità più complesse aumentando così la difficoltà di gioco in modo progressivo.

Mounir Samite

Gestione delle finestre

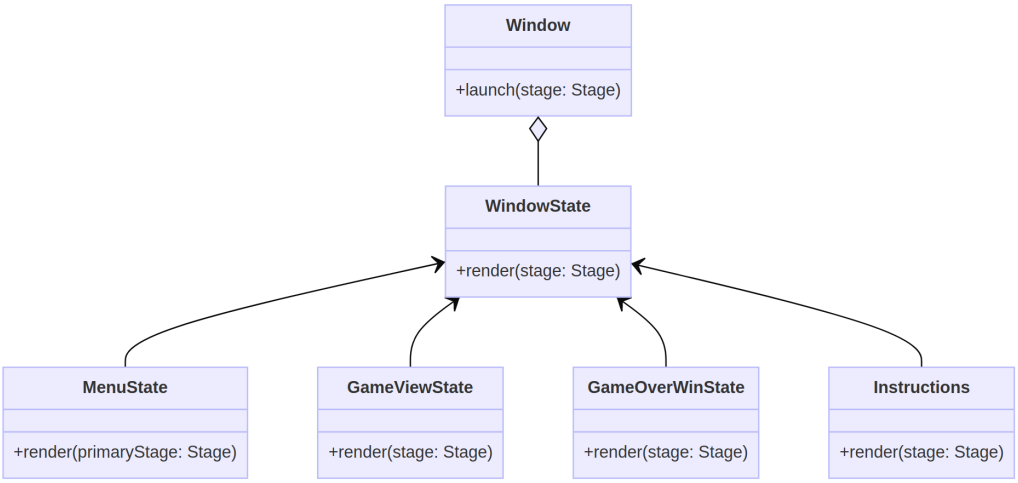

Figura 2.14: Schema UML della gestione delle finestre del gioco usando lo state pattern

Problema gestire le finestre del software, favorendo la modularità.

Soluzione Il pattern State è utile per gestire le finestre in un videogioco con JavaFX poiché consente di gestire lo stato di una finestra in modo dinamico e di cambiare facilmente lo stato della finestra a seconda delle esigenze dell'applicazione. In un videogioco, ad esempio, ci possono essere diverse finestre con diverse funzionalità, come la finestra di gioco, la finestra del menu, la finestra del game over e così via.
Utilizzando il pattern State, si può creare una classe per ogni stato possibile

della finestra e gestire i comportamenti specifici di ogni stato all'interno della sua classe. In questo modo, si può garantire che la finestra si comporti in modo corretto a seconda dello stato in cui si trova.
In questo modo, è possibile cambiare lo stato della finestra in modo dinamico a seconda delle azioni dell'utente o del gioco stesso, senza dover scrivere codice ripetitivo o complicato. Inoltre, questo approccio favorisce la modularità del codice, rendendolo più facile da mantenere e riutilizzare.

Mondo di gioco

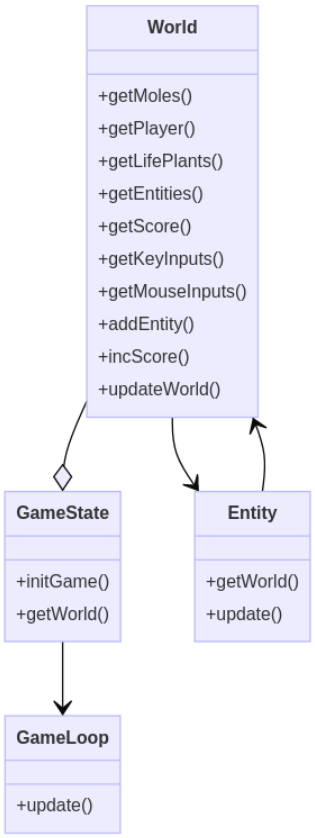

Figura 2.15: Schema UML della gestione del funzionamento del mondo di gioco

Problema serve un contenitore per le entità e le componenti del gioco, che possa aggiornare lo stato di questi ultimi.

Soluzione la classe *World* fa da contenitore, in particolare ha il compito di dividere i tipi di entità in base al *type* e di aggiornare lo stato delle entità.

- Per ritornare le entità usa dei semplici getter che cercano di ritornare delle copie dove è possibile.
- Per aggiornare le entità è stato usato una sorta di Iterator Pattern in cui tutte le entità vengono iterate con un *forEach* e aggiornate con il loro metodo interno *update()* per poi essere passate al GameLoop attraverso il GameState: il GameLoop andrà ad aggiornare lo stato delle entità "FPS"[2] volte al secondo.

[2]FPS è una costante nel GameLoop che servirà per decidere quante volte si aggiorna al secondo il gioco ed è anche un indicatore classico nei videogiochi che indica i fotogrammi per secondo

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per effettuare i test abbiamo utilizzato la libreria JUnit 5.9.1
I test automatizzati che abbiamo effettuato sono:

- **HitBoxTest**: in questa classe si testano le collisioni tra le diverse forme presenti nel gioco assicurandosi il corretto funzionamento.
- **EnemyInputTest**: in questa classe si verifica se, alla creazione di un EnemyInput, le variabili vengono correttamente inizializzate e se *EnemyTimesSpawn* incrementa.
- **KeyInputsTest**: in questa classe si controlla che, quando viene premuto o rilasciato un tasto, vengono correttamente inserite le *Direction* nella coda assicurandosi invece che non vengano aggiunti elementi in caso il player risulti bloccato.
- **MouseInputsTest**: in questa classe si controlla che tutte le variabili della classe *MouseInputs* assumono correttamente determinati valori quando necessario.
- **GameStateTest**: in questa classe si testa il sistema di punteggio, che dovrà aumentare e diminuire come previsto senza mai scendere sotto lo zero. Inoltre verrano testate l'inizializzazione del gioco, assicurandosi della creazione dei corretti elementi e la condizione di Game Over.
- **EntityTest**: in questa classe si controlla la corretta creazione delle entità controllando la presenza di componenti specifici per ogni tipologia.

- **HealthTest**: in questa classe vengono testate le funzionalità di aumento o perdita di vita e la condizione di morte delle Entità assicurandosi il corretto funzionamento.
- **PhysicsTest**: in questa classe vengono testate le interazioni tra le diverse entità assicurandosi il corretto funzionamento degli eventi scatenati da esse.
- **WorldTest**: in questa classe si controlla la corretta inizializzazione del mondo e si testano le funzioni di manipolazione della lista di Entità assicurandosi il corretto funzionamento.

3.2 Metodologia di lavoro

Siamo partiti inizialmente con un'attenta analisi del dominio del gioco e di cosa potesse essere aggiunto, cercando di rendere il gioco adeguatamente estendibile. Dunque è stato fatto un UML generale per avere in mente il funzionamento del gioco e sono state definite tutte le principali interfacce. Invece per quanto riguarda il workflow è stata adottata la metodologia di DVCS semplice, spiegata a lezione, con pull e push.

Ettore Farinelli

Mi sono occupato di:

- Gestione delle collisioni (**package smol.common, smol.common.hitbox**).
- *EnemyInput* e *EnemyInputFactory* (**package smol.controller.api**).
- Gestione degli input e movimento talpe (**package smol.controller.input**).
- *plantsCreation* (**package smol.model.impl**).
- *GraphicsDraw* e *GameViewState* (**package smol.view.impl**).

Ho inoltre contribuito a:

- *EnemyCreation* (**package smol.model.impl**).
- *PhysicComponent* implementando il receiveMovement del *WeaponPhysicsComponent* (**package smol.model**).
- *LoadImgs* aggiungendo la mappa per la scelta del pacchetto di skin (**package smol.view**).

Marco Galeri

Mi sono occupato di:

- *GameEngine* e *GameLoop* (**package smol.core**)
- *Entity* ed *EntityFactory* (**package smol.model**)
- *PhysicsComponent* e le sue sottoclassi (**package smol.model**)
- *HealthComponent* (**package smol.model.impl**)
- *InputComponent* (**package smol.controller**)
- rendere la grafica proporzionale allo schermo (**package smol.view.GameMap**)

Ho inoltre contribuito a:

- *GameViewState* Aggiungendo il punteggio (package smol.view)

Giovanni Paradisi

Mi sono occupato di:

- *GameState* (**package smol.controller**)
- *GraphicComponent*, le sue sottoclassi e la classe *LoadImgs* (**package smol.view**)
- Implementazione delle istruzioni di gioco nella classe *InstructionsState* (**package smol.view**)
- *EnemyCreation* (**package smol.model**)
- Salvataggio in locale del record *ScoreLocalStorage* (**package smol.model**)
- Creazione dei *bounding box* con l'entità *Wall*

Ho inoltre contribuito a:

- *GameViewState* aggiungendo il record (package smol.view)

Mounir Samite

Mi sono occupato di:

- Progettazione dell'utilizzo delle finestre con *Window* e *WindowState* con cui verranno renderizzate tutti i tipi di finestre come Menu, Gioco e Game Over (**package it.unibo.smol.view**);
- Implementazione del Menu di gioco nella classe *MenuState* (**package it.unibo.smol.view**);
- Implementazione del GameOver (fine gioco) nella classe *GameOver-WinState* (**package it.unibo.smol.view**);
- Progettazione del Mondo di gioco ovvero del contenitore di tutte le entità, degli input e dello score andando a gestire alcune loro funzionalità in *WorldImpl* (**package it.unibo.smol.model**);
- Progettazione della barra della vita del player in *HealthBarTankImpl* (**package it.unibo.smol.view**).

Ho inoltre contribuito a:

- *PlayerInputComponent* (**package it.unibo.smol.controller**);
- *WeaponInputComponent* (**package it.unibo.smol.controller**);
- *GameViewState* andando a gestire la barra della vita anche li (**package it.unibo.smol.view**);
- *GameEngine* aggiungendo le skins (che in questo caso sono intese come diversi tipi di grafica) al gioco: vettoriale e pixelata per ora (**package it.unibo.smol.core**)

3.3 Note di sviluppo

Ettore Farinelli

Utilizzo di stream

- Scelta, di una talpa, della pianta su cui andare: `https://github.com/TheDarkRuler/OOP22-SMOL/blob/7c96b119ea8cc75403c5656491ac352c0466de73/src/main/java/it/unibo/smol/controller/api/EnemyInput.java#L197-L217`

Utilizzo di Lambda Expressions

- https://github.com/TheDarkRuler/OOP22-SMOL/blob/7c96b119ea8cc75403c5656491a
  src/main/java/it/unibo/smol/view/impl/GameViewState.java#L147

Marco Galeri

Il GameLoop è stato realizzato ispirandosi al capitolo "GameLoop" presente in *Game Programming Patterns*[1] di Robert Nystrom.

Utilizzo di Stream

- Controllo delle collisioni: https://github.com/TheDarkRuler/OOP22-SMOL/blob/757228125cb2531fccca35b35f7645ceba9d5ae9/src/main/java/it/unibo/smol/model/api/PhysicsComponent.java#L34-49

Giovanni Paradisi

Utilizzo di Stream

- SpawnEnemies: https://github.com/TheDarkRuler/OOP22-SMOL/blob/7c96b119ea8cc75403c5656491ac352c0466de73/src/main/java/it/unibo/smol/model/impl/EnemyCreation.java#L110-L116

Utilizzo di JavaFX

- Utilizzo di FXML nelle *Instructions*: https://github.com/TheDarkRuler/OOP22-SMOL/blob/master/src/main/java/it/unibo/smol/view/impl/InstructionsState.java#L61

Mounir Samite

Utilizzo di Stream per gestione delle entità

- Prendere le talpe dalla lista di entità: https://github.com/TheDarkRuler/OOP22-SMOL/blob/dac2df26a7ed60c955c6ea339cb908958a450955/src/main/java/it/unibo/smol/model/impl/WorldImpl.java#L49-L55
- Prendere le piante dalla lista di entità: https://github.com/TheDarkRuler/OOP22-SMOL/blob/dac2df26a7ed60c955c6ea339cb908958a450955/src/main/java/it/unibo/smol/model/impl/WorldImpl.java#L63-L69

[1] https://gameprogrammingpatterns.com/

- Rileva il player dalla lista di entità: https://github.com/TheDarkRuler/OOP22-SMOL/blob/dac2df26a7ed60c955c6ea339cb908958a450955/src/main/java/it/unibo/smol/model/impl/WorldImpl.java#L56-L62

Utilizzo di JavaFX

- Utilizzo di javaFX senza FXML per la gestione della health bar in *GameViewState*: https://github.com/TheDarkRuler/OOP22-SMOL/blob/1a17f10567507032ae1308ff28c6147b7e08646a/src/main/java/it/unibo/smol/view/impl/GameViewState.java#L164-L201
- Utilizzo di FXML nel *MenuState* e nel *GameOverWinState*: https://github.com/TheDarkRuler/OOP22-SMOL/blob/1a17f10567507032ae1308ff28c6147b7e08
src/main/java/it/unibo/smol/view/impl/MenuState.java#L80
- Utilizzo basilare di CSS con FXML: https://github.com/TheDarkRuler/OOP22-SMOL/blob/1a17f10567507032ae1308ff28c6147b7e08646a/src/main/resources/css/Menu.css

Utilizzo di Lambda Expressions

- https://github.com/TheDarkRuler/OOP22-SMOL/blob/1a17f10567507032ae1308ff28c
src/main/java/it/unibo/smol/view/impl/HealthBarTankImpl.java#L71-L76
- https://github.com/TheDarkRuler/OOP22-SMOL/blob/1a17f10567507032ae1308ff28c
src/main/java/it/unibo/smol/model/impl/WorldImpl.java#L132-L136

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Ettore Farinelli

Sono molto soddisfatto del risultato ottenuto dal nostro lavoro di squadra. Questo progetto è stata una grande occasione per imparare quanto è importante la comunicazione e la condivisione di idee in un team, siccome in certe occasioni la mancanza di quest'ultima ha portato a diverse controversie. Grazie a questo lavoro mi sento di aver sviluppato diverse capacità utili per affrontare progetti futuri, come l'importanza di scrivere codice facilmente comprensibile e l'organizzazione generale di un team. Mi ritengo molto contento del risultato finale e sicuramente effettuerò varie modifiche e aggiunte al gioco in futuro.

Marco Galeri

Ho apprezzato molto la possibilità di realizzare un simile progetto, essendo la prima volta in cui ci cimentavamo in un progetto di questa portata le difficoltà non sono mancate ma siamo riusciti comunque a consegnare un programma di cui mi ritengo abbastanza soddisfatto. Questo progetto mi ha fatto realizzare quanto non sia scontato un buon lavoro di squadra. Comunque nonostante una partenza poco cooperativa nella fase di analisi e progettazione siamo riusciti a collaborare in modo efficiente nelle fasi successive. Detto ciò penso che il progetto abbia del potenziale e spero di riuscire a migliorare in futuro.

Giovanni Paradisi

Ritengo che questa esperienza sia stata molto formativa, sia dal punto di vista dello sviluppo del software che dal punto di vista della gestione di un team. Questo progetto mi ha dato la possibilità di realizzare un'applicazione che inizialmente avevamo solo sotto forma d'idee. Non è stato semplice realizzare per la prima volta un progetto con queste specifiche suddividendo il lavoro in modo equo e coerente all'interno del gruppo, anche a causa dell'inesperienza, ma sono contento del risultato. In conclusione mi sento molto fiducioso in modifiche future che realizzeremo ampliando le nostre conoscenze.

Mounir Samite

Inizierò dicendo che ho apprezzato molto il nostro lavoro di squadra su questo progetto. Tutti abbiamo dimostrato un grande impegno e dedizione nel raggiungere gli obiettivi prefissati, e siamo stati in grado di collaborare efficacemente per superare le sfide che abbiamo incontrato lungo il percorso.
Detto ciò, riconosco che una fase analisi e progettazione non sufficientemente dettagliata all'inizio del progetto ha comportato alcune difficoltà durante lo sviluppo del software. Ci siamo trovati ad affrontare alcune complessità che avremmo potuto evitare con una pianificazione più accurata e un'analisi dettagliata dei requisiti.
Nonostante questi problemi, sono soddisfatto del risultato finale del progetto. Abbiamo creato un software che superava le nostre aspettative. Tuttavia, credo che il processo sarebbe stato ancora più efficiente e produttivo se avessimo investito più tempo nella fase di analisi e progettazione.
In futuro, cercherò di fare in modo che questi aspetti siano considerati in modo più approfondito fin dall'inizio del processo di sviluppo del software, in modo da massimizzare l'efficienza del nostro lavoro di squadra e ottenere risultati ancora migliori.

Capitolo 5

Guida utente

All'avvio dell'applicazione l'utente si ritroverà nel menu di gioco composto da diversi bottoni:

- **Start**: per avviare la partita.
- **Instructions**: per visionare le istruzioni di gioco in un'altra schermata (compresa di bottone menu per tornare nella scena iniziale).
- **Quit**: per chiudere la finistra.
- **ListBox**: per la scelta della grafica di gioco.

I comandi di gioco sono i seguenti:

- **F11**: Attiva/disattiva la visuale a tutto schermo.
- **W**: Movimento verso l'alto.
- **A**: Movimento verso sinistra.
- **S**: Movimento verso il basso.
- **D**: Movimento verso destra.
- **Movimento del mouse**: Spostamento del mirino.
- **LMB** (tasto sinistro mouse): Colpo del martello (a seguito di una pressione prolungata di questo tasto il raggio di azione dell'arma sarà maggiore).

Bibliografia

- Abbiamo utilizzato come ispirazione durante le fasi iniziali
  - la repository di gameAsALab di A. Ricci `https://github.com/pslab-unibo/oop-game-prog-patterns-2022`
  - il seguente progetto degli anni precedenti `https://github.com/unibo-oop-projects/oop22-t2s-game`
- Abbiamo preso ispirazione dal seguente sito per fare le animazioni dei bottoni nel *MenuState* e *GameOver*: `https://genuinecoder.com/javafx-animation-tutorial/`