Relazione
"Unibo-td"

Aurora Francesco
Casali Marco
Murvai Cristina
Pulga Luca

07 luglio 2024

Indice

Capitolo 1

Analisi

"Bloons Tower Defense" è una serie di videogiochi di tipo tower defense. In questi giochi, il giocatore deve posizionare diverse torri lungo un percorso per fermare ondate di palloncini (chiamati "bloons") che cercano di attraversarlo. Ogni "bloon" che raggiunge la fine del percorso fa perdere una vita al giocatore, e il gioco termina quando le vite scendono a zero.

Il software mira ad una imitazione di questo videogioco. La nostra implementazione presenta alcune limitazioni e semplificazioni rispetto alla complessità del gioco originale. Questo ci ha permesso di comprendere meglio le dinamiche e le sfide dello sviluppo di un videogioco tower defense, anche se con risultati meno sofisticati rispetto alla versione commerciale.

1.1 Requisiti

Il gioco è implementato a round di difficoltà incrementale in cui i giocatori cercano di impedire ai nemici di raggiungere la fine di un percorso stabilito, posizionando difese per eliminarli.
Eliminando i nemici si guadagnano monete che possono essere utilizzate per acquistare nuove difese.
Si hanno a disposizione N vite, che si decrementano quando i nemici raggiungono la fine del percorso.
Quando si arriva a 0 vite il gioco termina.

Requisiti funzionali

- Il gioco inizia mostrando la schermata di avvio, in seguito è possibile scegliere la tipologia di mappa per iniziare la partita.

- Scelta la mappa, viene mostrata l'area di gioco: questa consiste in una griglia in cui il giocatore può posizionare le proprie difese (torri) in un determinato tempo prima che inizi ad arrivare la prima ondata di nemici. A destra della griglia si trovano le torri posizionabili nella mappa, in alto, invece, vengono indicate le vite, il round, il timer e i soldi guadagnati.
- Il gioco è suddiviso in round con difficoltà incrementale.
- I nemici sono le entità che seguono il percorso sulla mappa alla stessa velocità con lo scopo di giungere al termine, senza essere uccisi dalle difese.
- Il giocatore può piazzare torri lungo il percorso nei punti permessi per difendersi dai nemici.
- Ogni difesa ha a disposizione una e una sola arma.
- Le torri colpiscono i nemici quando entrano nel loro raggio di azione.
- I nemici hanno un life point che viene decrementato ogni volta che una torre riesce a colpire il nemico. Quando il life point diventa zero, il nemico risulta sconfitto e si vince una ricompensa (denaro) che può essere utilizzata per acquistare altre difese.
- Viene quindi gestito un sistema di reward dei nemici sconfitti e perdita delle vite del giocatore.

Requisiti non funzionali

- Garantire fluidità del gioco su diverse piattaforme.
- Assicurare che il gioco sia compatibile con le versioni dei principali sistemi operativi (Windows, Linux, macOS).

1.2 Analisi e modello del dominio

Unibo TD avrà più entità che dovranno collaborare tra loro.

Gestione dei round

Il gioco sarà suddiviso in round di difficoltà crescente. Tra un round e l'altro, ci sarà una pausa per consentire al giocatore di riprogrammare la propria strategia. Il numero di round sarà limitato, permettendo così al giocatore di vincere. Ogni round determinerà i tipi di nemici che possono essere generati, il tempo di apparizione e l'ordine in cui compariranno.

Gestione della mappa

Il gioco si svolge su una mappa composta da celle bidimensionali di dimensione uniforme che sono rappresentate da una immagine. All'interno della mappa è presente un percorso fisso che porta i nemici dalla cella di inizio alla cella di fine percorso, fuori da questo percorso su alcune celle è permesso costruire le difese. Prima di iniziare la partita l'utente ha a disposizione più mappe tra cui scegliere ma una volta iniziato il gioco la mappa non è più modificabile.

Gestione delle difese

Il gioco sarà costituito da una mappa. All'interno della mappa saranno presenti delle celle edificabili su cui poter posizionare le proprie difese (*torri*). Le difese sono le unità incaricate a proteggere il punto terminale della mappa. Queste saranno quindi in grado di infliggere danni ai nemici attraverso specifiche logiche di targettamento e di attacco. Ogni tipologia difesa possiederà certe statistiche per poter garantire maggiore eterogeneità al gioco. Per costruire una difesa sarà necessaria una certa quantità di denaro, acquisibile tramite l'eliminazione di nemici nei vari round. Ogni difesa oltre ad avere relative speficifiche e caratteristiche, fra queste figurano anche le *armi* a disposizione di ogni torre, le quali gestiscono la *frequenza* di sparo dei *proiettili* per arrecare danno ai nemici. Ogni proiettile sparato da una determinata torre con una determinata arma, dispone di un certo danno. Ogni qualvolta una difesa sconfigge un nemico, verrà accreditato al **player** il denaro relativo al nemico sconfitto. Due feature interessanti sarebbero la rimozione e il riposizionamento delle difese sulla mappa nei punti permessi. Queste purtroppo non potranno essere effettuate all'interno del monte ore previsto e conseguentemente si rimanda tale feature in futuro.

Gestione dei nemici

Il gioco prevede tre diverse tipologie di nemici, ciascuno caratterizzato dal tipo (base, medium, advanced), dal nome, dai punti vita, dalla ricompensa, ecc. Tutti seguono lo stesso percorso sulla mappa alla stessa velocità. Se i nemici riescono ad arrivare alla fine del percorso senza essere uccisi dalle difese viene recato un danno alle vite del giocatore, in caso contrario, se le difese riescono ad eliminare i nemici prima che raggiungano la fine, il giocatore riceve una ricompensa. La generazione dei nemici è strutturata in "ondate", gestite dal round manager.

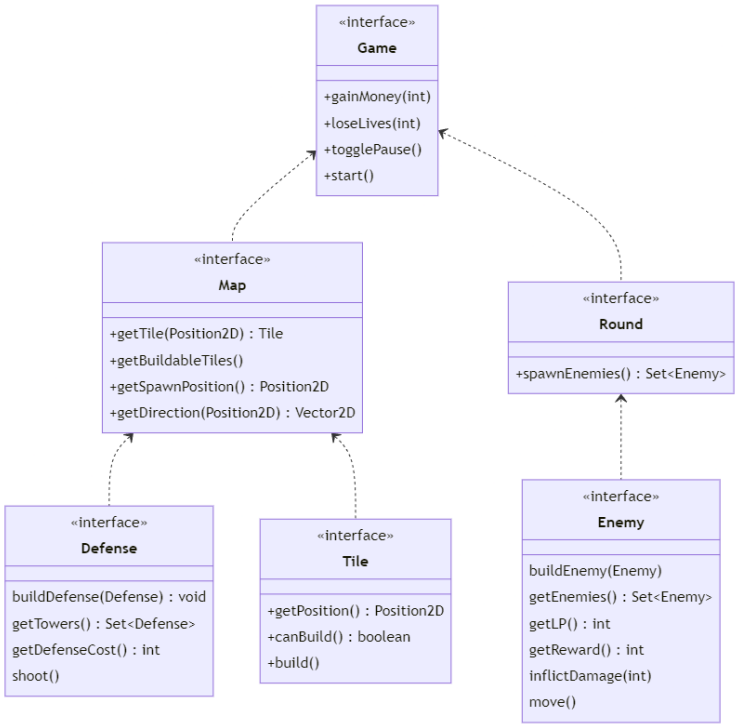

Figura 1.1: UML Modello di dominio

Capitolo 2

Design

2.1 Architettura

UNIBO-TD adotta il pattern architetturale MVC. Il pattern separa l'applicazione in tre componenti logici (Model, View, Controller). Rispettivamente, nel dettaglio, rappresentano:

• Controller: funge da intermediario tra Model e View, limitando l'accesso diretto al Model e fornendo solo oggetti di trasferimento dati. Ciò significa che gli altri componenti dell'applicazione, come la View, possono accedere solo a una rappresentazione immutabile dei dati nel Model, garantendo un maggiore controllo sull'integrità dei dati.

• Model: gestisce in modo trasparente la logica del gioco, come ad esempio la generazione e aggiornamento delle entità tramite il defense manager e l'enemy manager.

• View: si occupa di mostrare graficamente le diverse entità e le schermate di gioco. L'implementazione della View, così come è stata definita, è facilmente sostituibile poiché non dipende dalle classi sottostanti.

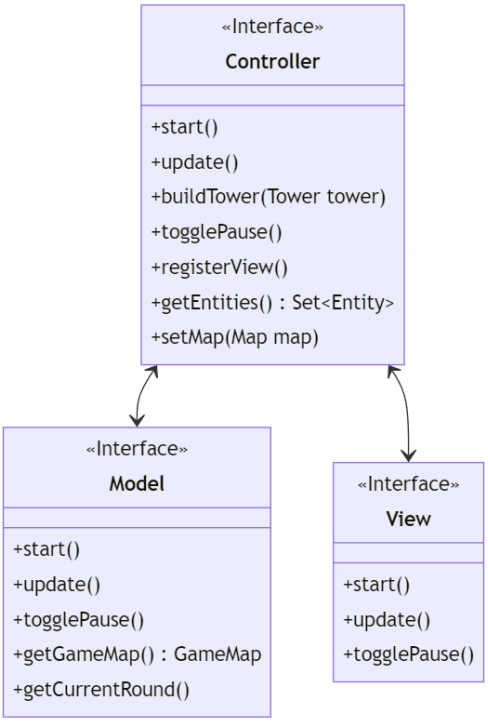

Figura 2.1: UML MVC di Unibo-td

2.2 Design dettagliato

2.2.1 Francesco Aurora

Gestione GUI e azioni utente, round, player

Gestione Round

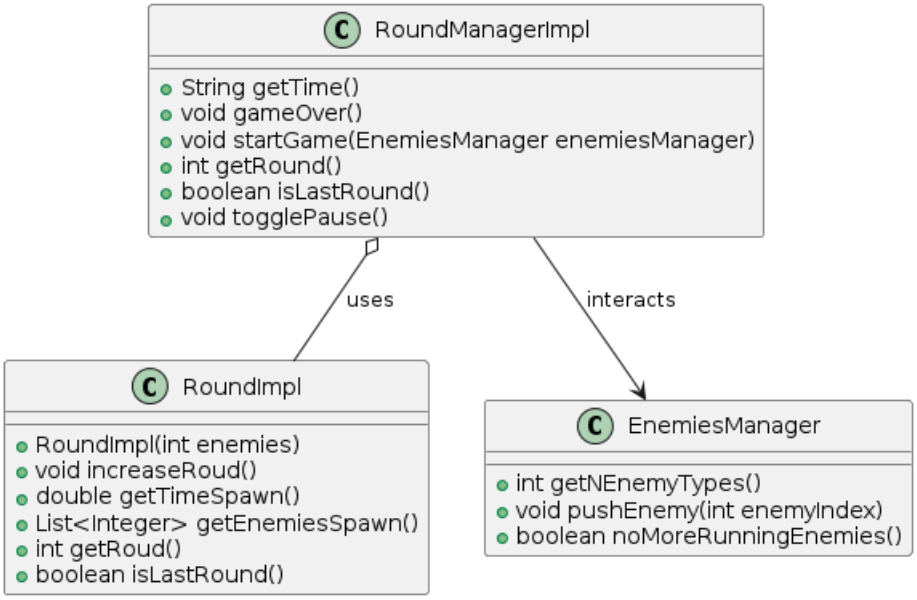

Figura 2.2: UML relazioni Round

Problema Nel contesto di un gioco a round, è essenziale gestire in modo efficace la progressione dei round e la generazione dei nemici. Il problema principale consiste nel determinare come incrementare il numero di round, regolare il numero di nemici generati, gestire i tempi di spawn dei nemici e differenziare i round normali dai "boss round". Inoltre, la gestione del gioco deve includere la capacità di mettere in pausa, riprendere e interrompere il gioco in modo sicuro, sincronizzando correttamente l'accesso ai dati condivisi tra i thread per evitare condizioni di gara e inconsistenze.

Soluzione Per affrontare questi problemi, ho progettato un sistema che utilizza due classi principali: `RoundImpl` e `RoundManagerImpl`.

`RoundImpl` gestisce la logica interna dei round, inizializzando il numero di nemici, il tempo di spawn e una lista di nemici. Il metodo `increaseRoud()`

gestisce l'incremento dei round, aggiornando la lista dei nemici e modificando il tempo di spawn a seconda del tipo di round.

`RoundManagerImpl` gestisce il flusso del gioco, inclusi il countdown e la generazione sequenziale dei nemici, utilizzando due thread distinti. Il metodo `togglePause()` gestisce la pausa e la ripresa del gioco in modo sicuro. La sincronizzazione dei dati tra i thread è garantita da blocchi di sincronizzazione.

Ho valutato l'alternativa di gestire tutti i round e i nemici in una sola classe, ma ho optato per la separazione per migliorare la modularità e la chiarezza del codice. Utilizzare thread distinti per il countdown e la generazione dei nemici ha migliorato la gestione del flusso di gioco e facilitato la manutenzione ed estensione del sistema.

Questo design offre un sistema robusto e flessibile, risolvendo efficacemente i problemi identificati e migliorando l'esperienza di gioco complessiva.

Gestione player

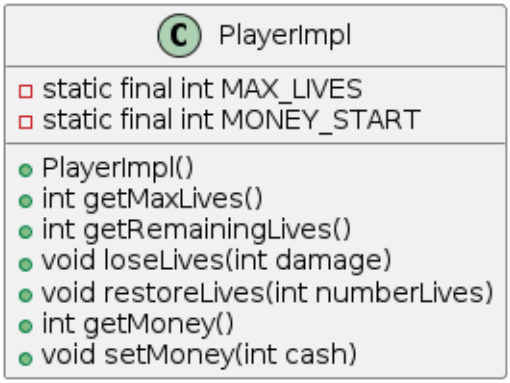

Figura 2.3: UML Player

Problema Nel contesto di un gioco, è essenziale gestire lo stato del giocatore, compreso il numero di vite rimanenti e il denaro disponibile. Il problema principale consiste nel mantenere aggiornati questi due attributi in base alle azioni del giocatore e agli eventi nel gioco.

Soluzione Per affrontare questo requisito, ho progettato la classe PlayerImpl. Questa classe implementa l'interfaccia Player, fornendo metodi essenziali per gestire le vite e il denaro del giocatore. Il costruttore inizializza il giocatore con un numero massimo di vite e un quantitativo iniziale di denaro. I metodi loseLives(int damage) e restoreLives(int numberLives) aggiornano

le vite del giocatore in base ai danni subiti o al ripristino delle vite. Allo stesso modo, i metodi getMoney() e setMoney(int cash) gestiscono il denaro del giocatore, consentendo di recuperare la quantità di denaro attuale e aggiornarla in base alle transazioni nel gioco.

Questa progettazione garantisce un controllo efficace e dinamico sulle risorse del giocatore, fondamentale per una esperienza di gioco coinvolgente e bilanciata.

Gestione della Schermata di Avvio e Selezione Mappa

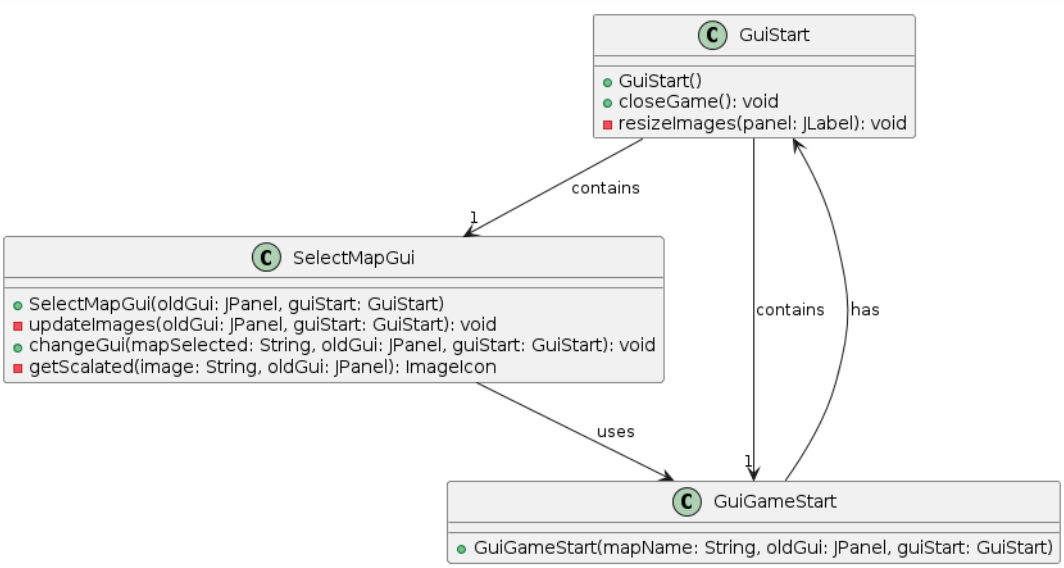

Figura 2.4: UML relazioni GUI

Problema Nell'implementazione di un'applicazione GUI per un gioco, è essenziale gestire efficacemente la schermata iniziale, includendo un pulsante di avvio ben posizionato e scalabile per adattarsi a diverse dimensioni di scherma, la selezione delle mappe ancora adattabile a diverse dimensioni dello schermo. Il problema principale consiste nel progettare un'interfaccia utente che permetta agli utenti di selezionare una mappa tra diverse opzioni, visualizzare anteprime delle mappe adattate alle dimensioni dello schermo e fornire un'esperienza utente coerente e intuitiva su diversi dispositivi.

Soluzione Le classi 'GuiStart' e 'SelectMapGui' sono componenti chiave nell'interfaccia utente del gioco. 'GuiStart' gestisce la schermata iniziale, impostando il titolo della finestra su "Unibo TD", utilizzando 'BorderLayout'

per organizzare i componenti e includendo un pulsante di avvio. 'SelectMapGui' gestisce la selezione della mappa, permettendo agli utenti di scorrere tra le mappe disponibili e avviare il gioco con la mappa selezionata.

Entrambe le classi sono progettate per essere responsive, adattandosi automaticamente alle dimensioni dello schermo per garantire una esperienza utente ottimale.

Questa soluzione offre un'interfaccia utente ben strutturata e facile da navigare, ottimizzata per migliorare l'esperienza dell'utente fin dai primi istanti di utilizzo del gioco. È progettata per consentire una transizione fluida tra diverse schermate, inclusa quella del gioco vero e proprio, all'interno della stessa finestra.

Come futura implementazione, si potrebbe considerare l'integrazione degli effetti sonori e la visualizzazione di più mappe simultaneamente sullo schermo, al fine di offrire agli utenti una visione più completa e coinvolgente delle opzioni di gioco disponibili.

Schermata di gioco Mappa

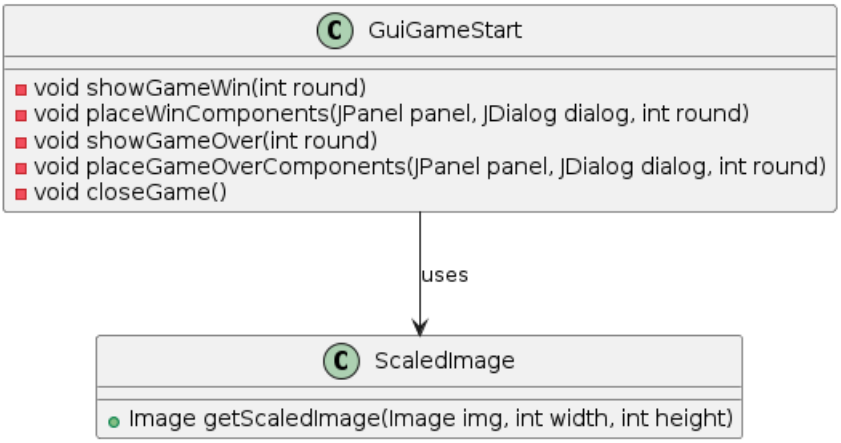

Figura 2.5: UML GUI mappa per area di mia pertinenza

Problema Si vuole gestione la vittoria e sconfitta, resize della mappa all'adattamento della finestra, bottoni pausa e settings.

Soluzione La mappa viene ridimensionata tramite l'evento di `resize` ogni volta che il contenitore delle immagini della mappa cambia dimensione. La gestione della vittoria e della sconfitta è implementata in modo tale che, quando il core rileva eventi di `GameOver` o `GameWin`, viene mostrato un `JDialog`.

Al clic sul pulsante `exit`, viene generato l'evento di chiusura del gioco. Poiché le pagine sono state incapsulate, l'evento deve essere richiamato sulla prima GUI mostrata. Il tutto deve essere scalabile tramite l'utilizzo di metodi in cascata e non ridondante. La suddivisione dei due metodi per `gameWin` e `gameOver` permette, in futuro, di avviare comportamenti diversi e di gestire in modo appropriato eventuali store di dati.

Nella schermata di gioco sono stati aggiunti i pulsanti `pausa` e `impostazioni`. Il pulsante `pausa` avvia una chiamata al core di gioco per mettere in pausa i vari componenti del gioco. Il pulsante `impostazioni` attualmente non è funzionante, poiché non sono stati ancora gestiti gli effetti sonori e la velocità di gioco. In futuro, si potrebbe implementare il pulsante `impostazioni` per mettere in pausa il gioco e permettere di configurare questi aspetti.

2.2.2 Marco Casali

Gestione mappa

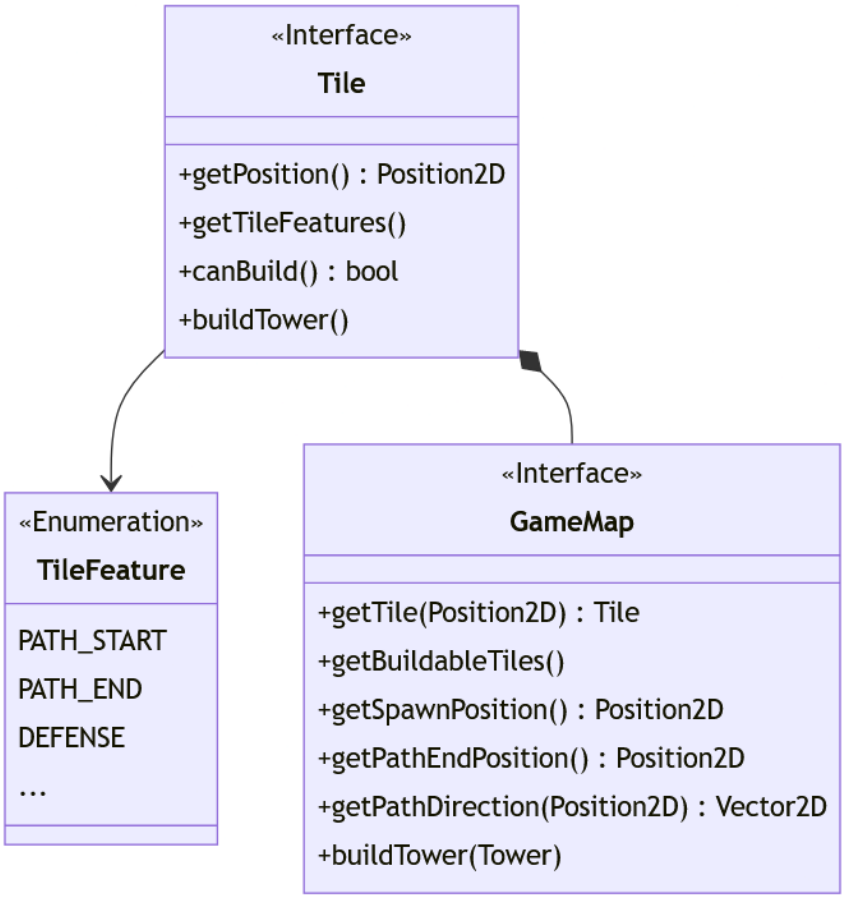

Figura 2.6: Schema UML della mappa di gioco

Problema La mappa è composta da una matrice bidimensionale di celle che hanno la stessa dimensione. Ogni cella però ha comportamenti diversi, ad esempio ci sono celle che compongono il percorso dei nemici o celle su cui possono essere costruite le difese.

Soluzione Ho semplificato il comportamento delle celle in proprietà definite in *TileFeature*, ogni cella quindi ha un insieme di *TileFeature* immutabile e la mappa identifica il comportamento delle celle tramite questo insieme.

Creazione mappa e celle

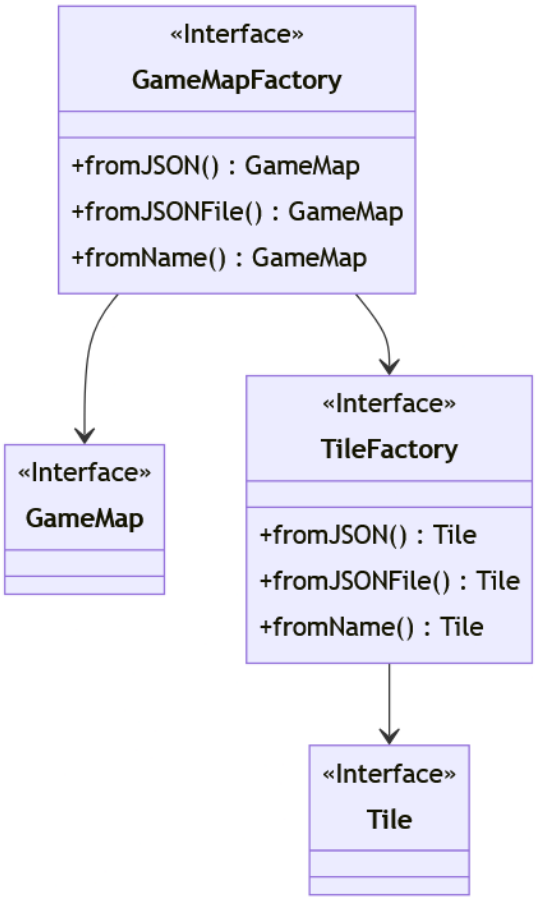

Figura 2.7: Schema UML MapFactory e TileFactory

Problema La mappa è un oggetto complesso e si vuole salvarla, come tutte le altre entità, su file formato JSON.

Soluzione Ho utilizzato il pattern *Factory* per la creazione di mappa e celle, ho creato due interfacce separate ma effettivamente i metodi sono gli stessi, tramite queste interfacce l'implementazione di mappa e celle è completamente incapsulata ed è possibile ottenere un'istanza solamente tramite il nome senza costruttori complessi.

Gestione del game loop

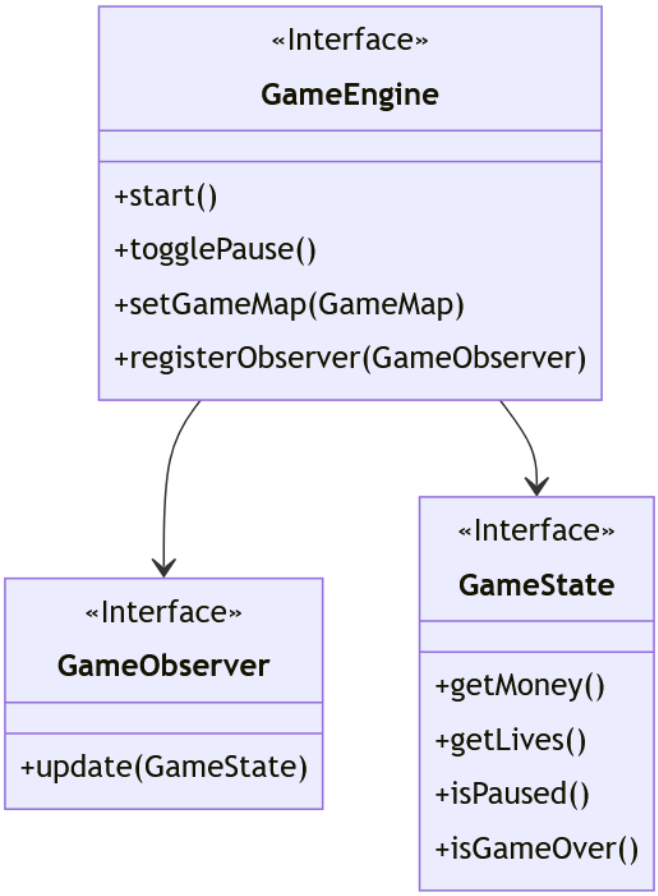

Figura 2.8: Schema UML del game loop

Problema In ogni momento del gioco vengono create e distrutte diverse entità tra cui nemici,difese e proiettili. Il movimento di queste entità deve essere fluido,uniforme e deterministico, quindi indipendente dalle capacità della macchina che esegue il gioco.

Soluzione La gestione del game loop è affidata al *GameEngine* che viene eseguito su un thread separato e si occupa di aggiornare le altre entità tramite il pattern *Observer*, in questo caso gli *Observer* si registreranno e riceveranno periodicamente una chiamata al metodo *update()*, dove gli verrà passato un oggetto immutabile di tipo *GameState* che contiene tutte le informazioni sullo stato del gioco.

2.2.3 Cristina Murvai

Gestione delle diverse configurazioni di nemici

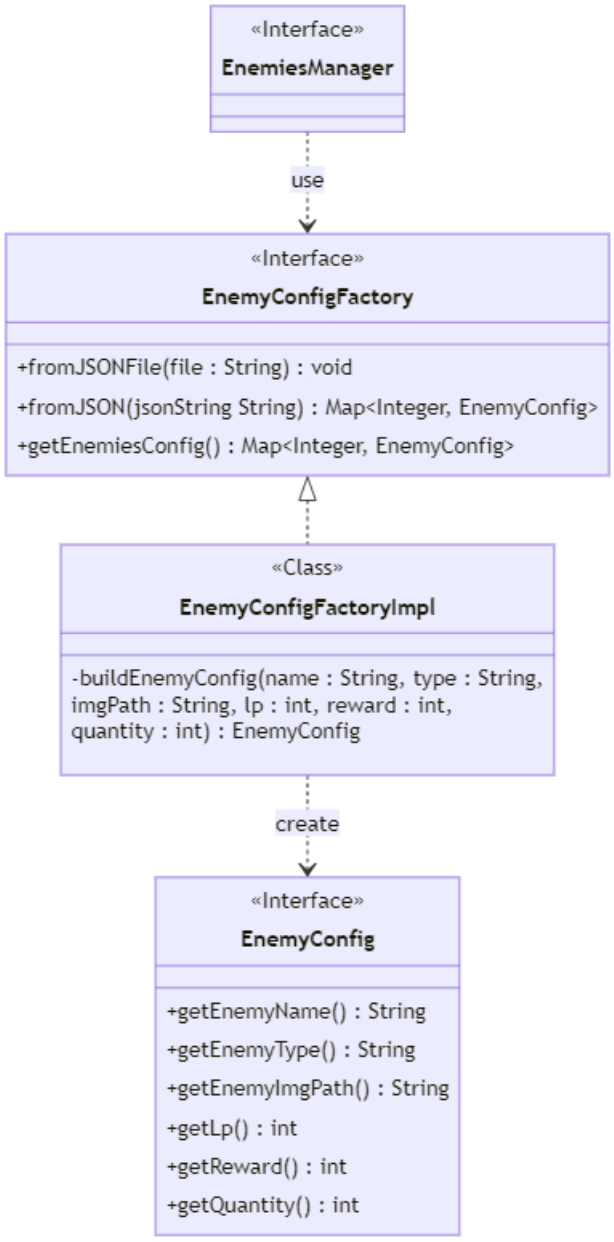

Figura 2.9: UML factory per caricamento di configurazioni di nemici

Problema Il gioco prevede attualmente tre tipologie di nemici, ciascuno con paramentri diversi (quali il nome, i punti vita, il reward assegnato con

l'uccisione, ecc.) che è necessario reperire ogni volta in cui un nemico di quella tipologia viene creato.

Soluzione Per garantire futura estendibilità ad ulteriori tipologie di nemici e facile modifica dei parametri è stato deciso di gestire la configurazione dei nemici attraverso un file JSON. Tuttavia, una lettura di questo file in seguito alla creazione di ciascun nemico sarebbe inutilmente dispendiosa del punto di vista delle risorse del sistema. Per questo motivo, si è deciso di sfruttare il pattern factory per creare e salvare in memoria i dati relativi alle varie tipologie di nemici, garantendo un accesso più rapido in fase di generazione degli stessi tramite il metodo *getEnemiesConfig()* dell'interfaccia *EnemyConfigFactory*. In particolare il metodo *fromJSONFile()* è sfruttato da *EnemiesManager* per leggere il file di configurazione e creare attraverso la factory i diversi *EnemyConfig*.

Gestione dei nemici: spawn, movimento e distruzione

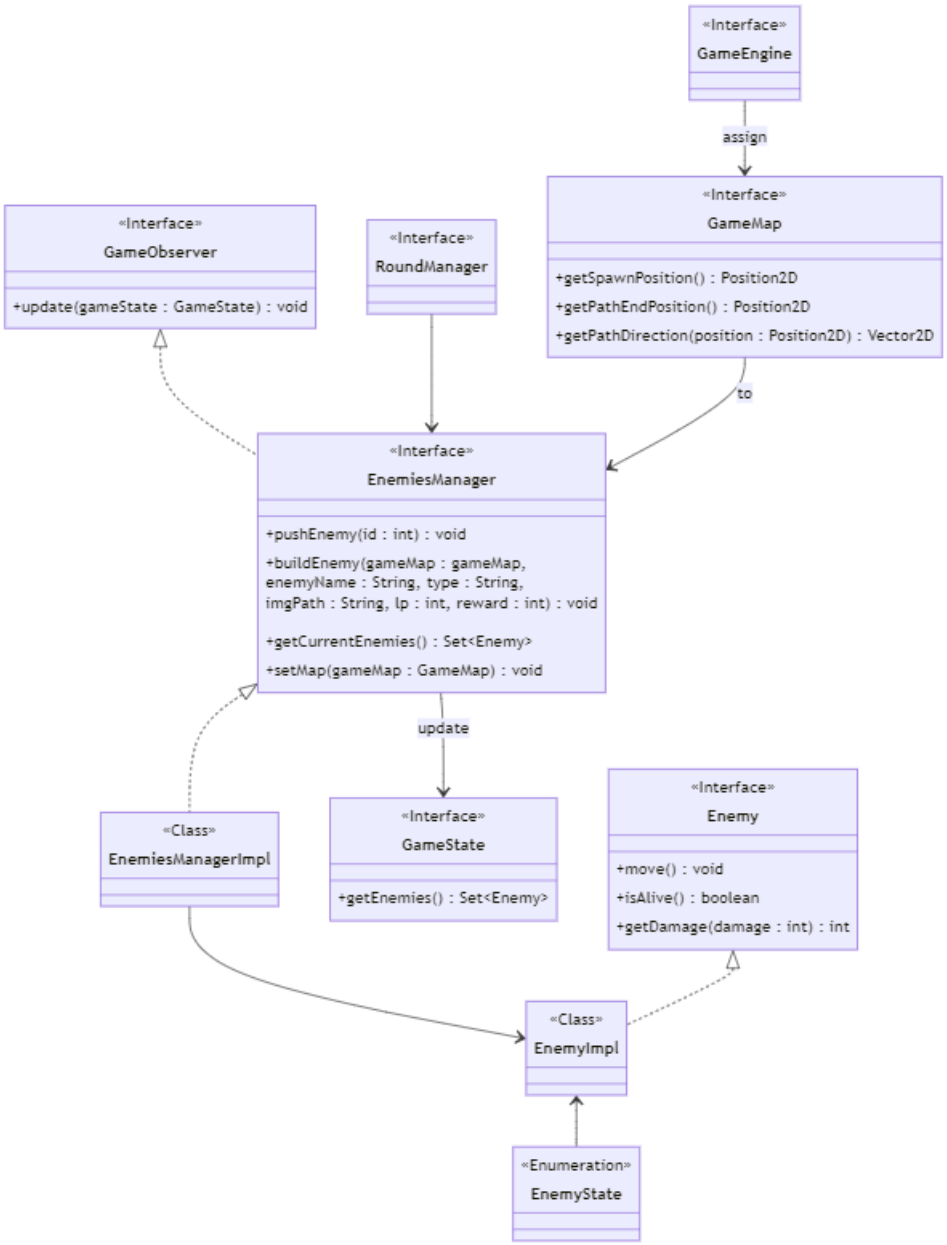

Figura 2.10: UML gestione dei nemici

Problema Per poter correttamente gestire la creazione e il movimento dei nemici è necessario avere informazioni riguardo la mappa quali posizione di partenza, percorso da seguire e fine del percorso.

Soluzione Dato che la scelta della mappa viene fatta in un secondo momento rispetto all'avvio del gioco, si è deciso di avviare comunque l'*EnemiesManager* in modo tale da eseguire immediatamente il load delle diverse configurazioni di nemici descritto al punto precedente. Per questo motivo si è scelto di modellare *GameMap* per il manager dei nemici con tipo *Optional*, il cui valore viene assegnato dal *GameEngine* appena l'utente effettua la selezione della mappa di gioco avviando la partita.

Problema Per la gestione delle interazioni difese-nemici si è reso necessario fornire al manager delle difese un modo efficace per monitorare quali nemici sono in vita e conoscere la loro posizione in tempo reale. Questo è cruciale per permettere alle torri di attaccare correttamente i nemici.

Soluzione Si è deciso di sfruttare il pattern Observer per risolvere questo problema rendendo il set dei nemici osservabile. Questo set è disponibile nell'interfaccia *GameState*, in modo che qualsiasi componente, come il manager delle difese, che necessita di monitorare i nemici possa farlo. Periodicamente il *GameState* viene aggiornato con la versione più recente del set dei nemici (in cui ad esempio, un nemico viene eliminato o cambia posizione). Successivamente, gli *Observers* vengono notificati del cambiamento e possono conseguentemente aggiornare il loro stato. Ciò garantisce che tutte le parti del sistema che devono reagire ai cambiamenti dello stato dei nemici lo facciano immediatamente e in modo coerente. Inoltre, questo rende il codice scalabile, poiché è possibile aggiungere nuovi *Observers* dello stato dei nemici senza modificare il codice sorgente esistente.

Problema Il *RoundManager* si occupa della gestione dei vari round e quindi anche di stabilire il numero incrementale di nemici di diverse tipologie che popoleranno la mappa di gioco. Per questo motivo è necessario fornirgli un modo per generare i nemici in maniera efficace. Per fare ciò è necessario anche considerare che, visto che *EnemiesManager* estende *GameObserver*, il set dei nemici potrà essere modificato solo quando il manager avrà il controllo nel metodo *update()*.

Soluzione Per risolvere questo problema si è deciso di fornire al *RoundManager*, tramite *EnemiesManager*, il metodo *pushEnemy()* che, preso in ingresso l'identificativo di una tipologia di nemici, tramite *buildEnemy()* si occupa di creare un nuovo nemico. L'*EnemiesManager* inizializza poi la posizione del nemico, utilizzando i metodi forniti da *GameMap*, e lo aggiunge ad un set temporaneo di nemici da inserire nella partita. Una volta che il

manager ha il controllo sul set dei nemici, ovvero quando il *GameEngine* ne chiama il metodo *update()*, i nemici presenti nel set temporaneo vengono aggiunti a quello effettivo e il loro movimento sulla mappa inizia rendendoli anche colpibili dalle difese.

Gestione visualizzazione grafica dei nemici

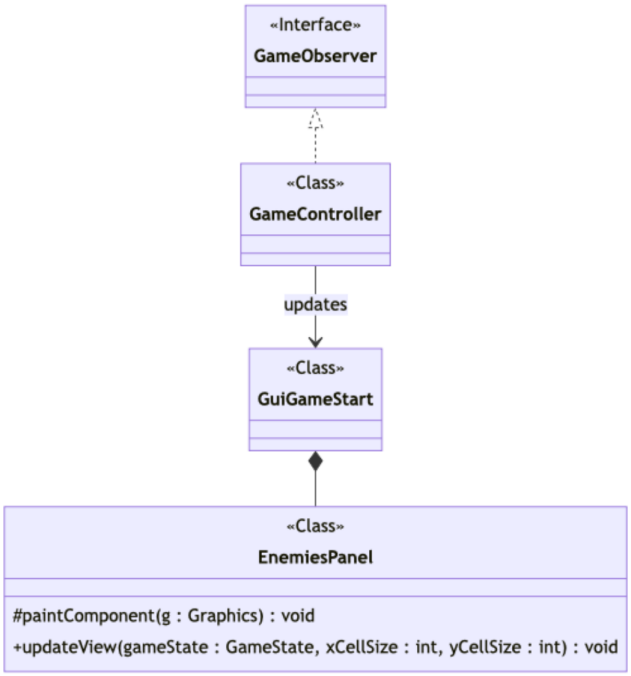

Figura 2.11: UML gestione view dei nemici

Problema Per la visualizzazione grafica dei nemici è stato necessario trovare un modo per far si che questa sia aggiornata in tempo reale in base allo spawn, alla morte e al movimento dei nemici sulla mappa.

Soluzione Si è deciso di integrare la view dei nemici all'interno dell'infrastruttura grafica implementata dai colleghi in modo tale da sfruttare il pattern observer. In particolare, il *GameController* chiama il metodo *update()* su *GuiGameStart* che a sua volta chiama *updateView()* su *EnemiesPanel* che ridisegna i nemici sulla schermata di gioco basandosi sulla loro nuova posizione.

2.2.4 Luca Pulga

Gestione delle difese

Problema Si vuole gestire in maniera scalabile le varie entità presenti all'interno del gioco, come le difese (torri, armi, proiettili) e i nemici, evitando l'eccessiva scrittura di codice duplicato.

Soluzione E' possibile suddividere le varie caratteristiche delle varie entità in sotto-entità, in modo da attribuire le relative proprietà ad ogni tipologia di entità che è presente o che potrà essere presente in futuro, rendendo la struttura scalabile. Sono state realizzate interfacce e classi astratte, a cui sono state delegate le assegnazioni delle varie proprietà, fornendo la condivisione del codice comune e contratti parziali, implementabili dalle sottoclassi. Un'ulteriore soluzione poteva essere quella di utilizzare solo interfacce e come implementazione di esse utilizzare i record, in quanto fornirebbero un modo decisamente più coinciso per dichiarare classi immutabili ed eliminerebbero molta della verbosità associata alle classi standard.

Figura 2.12: Modello per la gestione delle entità.

Problema Disporre di un oggetto in grado di poter creare qualsiasi tipo di entità, centralizzando la parte di creazione degli oggetti che verranno caricati e successivamente utilizzati durante il gioco.

Soluzione Si utilizza il *Factory Method Pattern* in modo da sfruttare la sua capacità di separare la costruzione degli oggetti dal loro utilizzo. Dunque questa separazione ci consente di avere una maggiore flessibilità e modularità del codice, rendendolo più facile da mantenere e aggiornare anche introducendo in futuro nuove entità o separare in sotto-entità quelle già esistenti. La *Factory* provvederà a leggere i files json all'interno di specifiche cartelle per caricare le entità, come le *torri*, che saranno poi successivamente gestite dal *Defense Manager*.

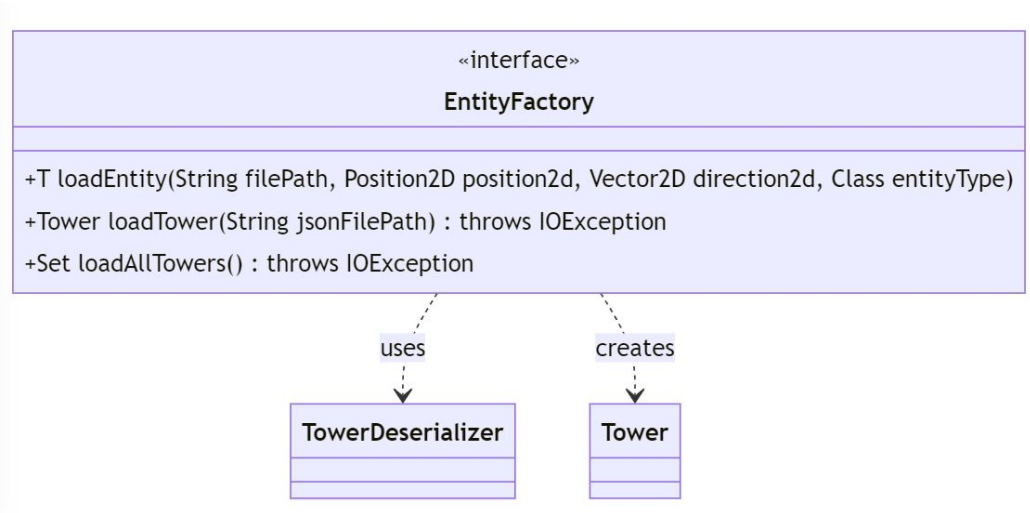

Figura 2.13: Factory Method Pattern per il caricamento delle entità.

Problema Gestione delle istanze delle torri senza avere un'entità centralizzata.

Soluzione La soluzione prevedere l'implementazione di un Defense Manager in grado di gestire le entità difesa. Questa soluzione centralizzata permette di gestire all'interno di un unica classe tutte le entità difesa, fornendo anche un modo comodo all'esterno per ottenere informazioni con altri oggetti.

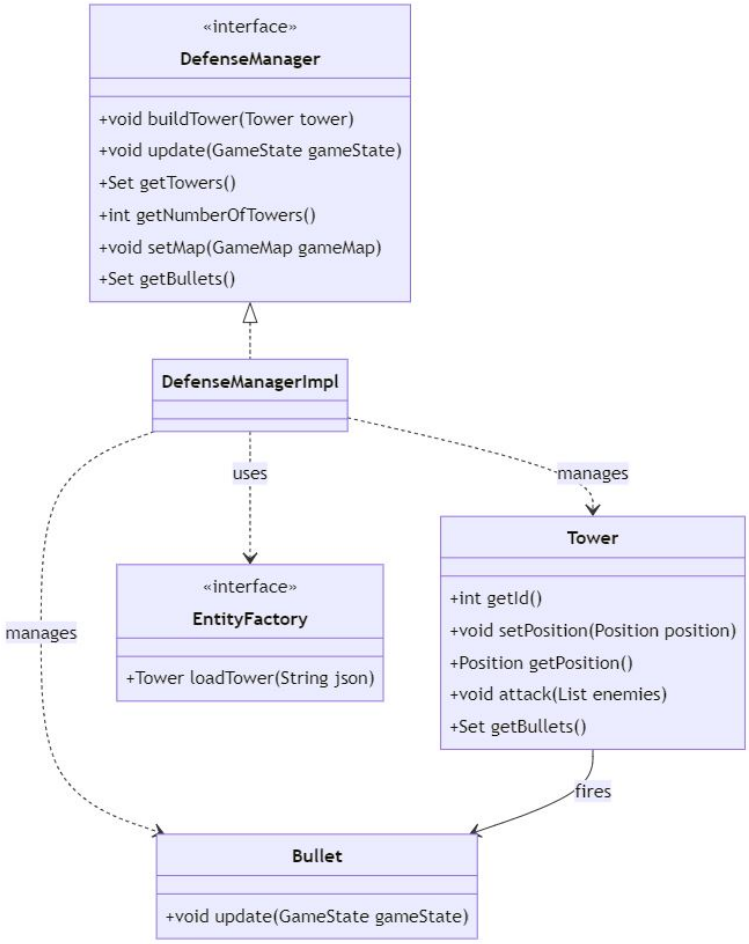

Figura 2.14: Defense Manager per la gestione delle difese.

Problema Gestione di varie tipologie di entità in tempo reale come torri, proiettili, nemici e tutte le varie entità che potrebbero essere introdotte in futuro che hanno necessità di reagire a certi eventi.

Soluzione A livello di difese, si sottoscrive il Defense Manager all'Observer, il quale consente di definire un meccanismo di sottoscrizione per notificare a più oggetti gli eventi che accadono all'oggetto che stanno osservando, tutto ciò rispettando il principio Open/Closed, migliorando così la modularità e la manutenibilità del codice.

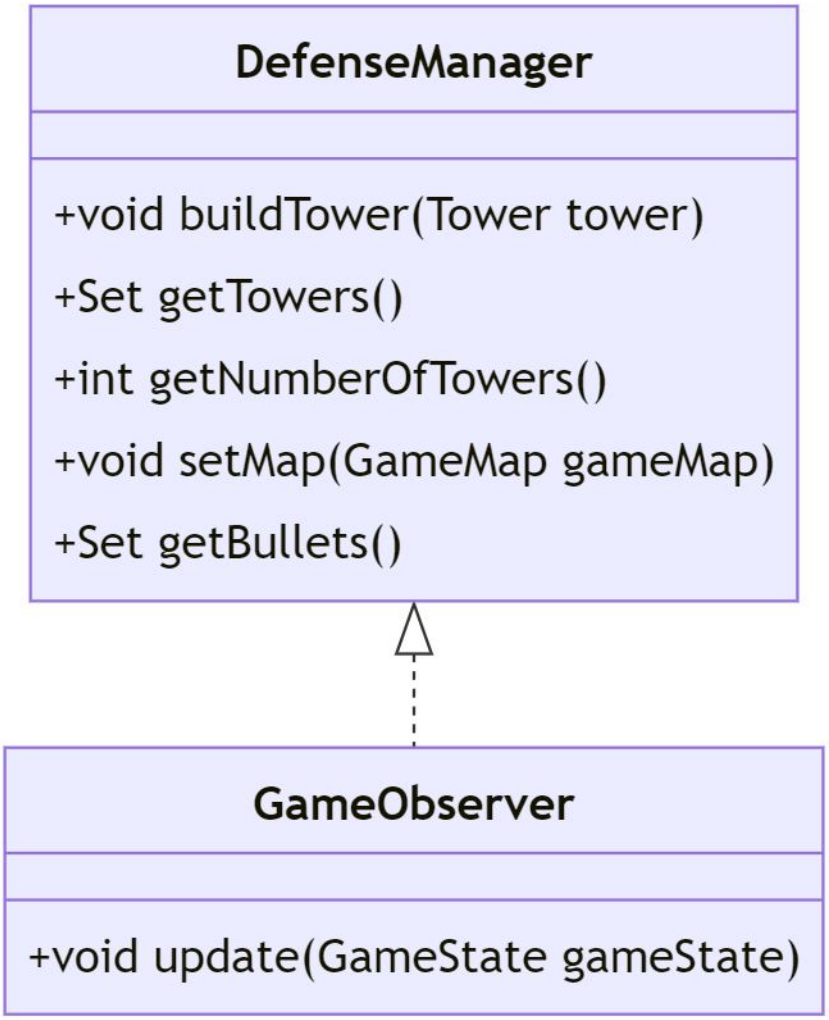

Figura 2.15: Figura 2.5: Defense Manager e Observer.

Problema Implementazione di diverse tipologie di targettamento dei nemici da parte delle torri a seconda della loro tipologia.

Soluzione Si implementa lo Strategy Pattern, in questo modo il contesto diventa indipendente dalle strategie concrete di targettamento dei nemici, per cui è possibile aggiungere/modificare gli algoritmi di targettamento senza modificare il codice all'interno delle varie classi che potrebbero implementare la relativa strategia di attacco.

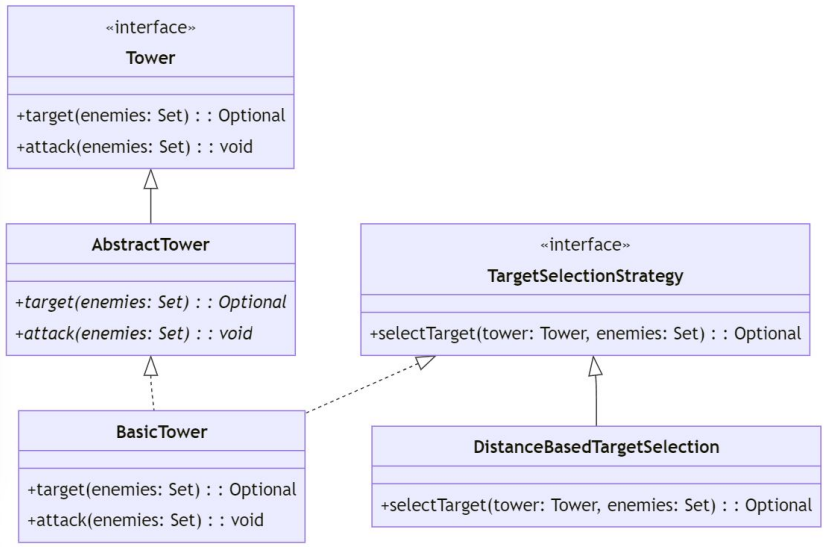

Figura 2.16: Pattern Strategy per targettamento dei nemici.

Problema Implementazione di diverse tipologie di attacchi verso i nemici da parte delle torri a seconda della loro tipologia.

Soluzione Si implementa lo Strategy Pattern, in questo modo il contesto diventa indipendente dalle strategie concrete di attacco verso i nemici, per cui è possibile aggiungere/modificare gli algoritmi di attacco senza modificare il codice all'interno delle varie classi che potrebbero implementare la relativa strategia di attacco.

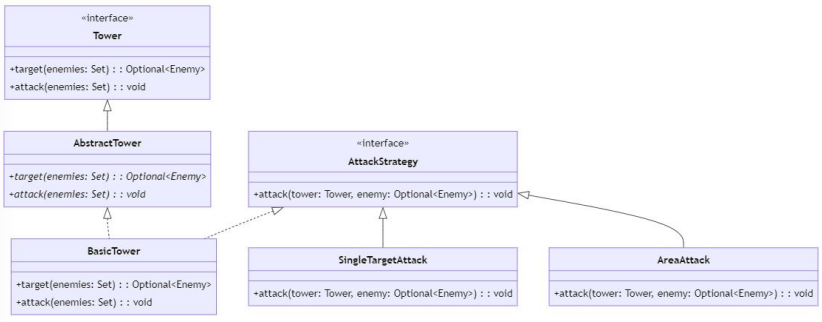

Figura 2.17: Pattern Strategy per attacco ai nemici.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per verificare il corretto funzionamento del gioco Tower-Defense, sono stati creati appositi test automatizzati utilizzando JUnit. I test automatizzati sono necessari per poter testare se le logiche pensate durante la fase analisi, di progettazione e di implementazione del Model, e delle classi affini di utility, sono funzionanti.

- ***TestBasicTower:*** Test automatizzato per garantire la correttezza dei metodi di get e set delle difese torri per poter gestire durante il gioco le sue proprietà.
- ***TestBulletImpl:*** Test automatizzato per garantire la correttezza dei metodi di get e set della posizione e direzione del proiettile, fondamentale per la gestione grafica del gioco.
- ***TestDefenseFactory:*** Test automatizzato per garantire la correttezza di lettura da file JSON delle entità torri attraverso l'utilizzo della **??**EntityFactory
- ***TestTargetStrategy:*** Test automatizzato per garantire la correttezza dei calcoli effettuati durante la fase di targettamento di nemici da parte delle torri.
- ***TestEnemiesConfigFactoryImpl:*** Test automatizzato per verificare la corretta funzionalità di caricamento delle configurazioni dei nemici da file JSON.
- ***TestEnemyImpl:*** Test automatizzato per garantire il funzionamento di vari metodi della classe Enemy, come ad esempio verificare se lo

stato iniziale, i punti vita, la ricompensa e il percorso dell'immagine sono impostati correttamente, oppure controllare se il nemico si muove lungo la direzione giusta, aggiornando il suo stato una volta raggiunta la fine del percorso.

- ***TestPlayer**:* Test automatizzati per verificare la corretta inizializzazione di soldi e vite, nonché il corretto decremento e incremento di questi valori.
- ***TestRound**:* Test automatici per verificare l'incremento dei round e della complessità del gioco.
- ***TestRoundManagerImpl**:* Test automatici per verificare l'avvio del conto alla rovescia e l'avvio, oltre alla pausa del gioco.
- ***TestGameMapFactory**:* Test per verificare la corretta creazione della mappa.
- ***TestTileFactory**:* Test per verificare la corretta creazione della cella.

3.2 Note di sviluppo

3.2.1 Francesco Aurora

Utilizzo di lambda expressions

Utilizzate in vari punti. Un esempio: `https://github.com/PulgaLuca/OOP23-unibo-td/blob/f84040bdc412c3abbb06a344ec0ae82e5af06e6f/src/main/java/it/unibo/model/round/RoundManagerImpl.java#L107`

Utilizzo della libreria SLF4J

Utilizzata in vari punti. Un esempio: `https://github.com/PulgaLuca/OOP23-unibo-td/blob/f84040bdc412c3abbb06a344ec0ae82e5af06e6f/src/main/java/it/unibo/model/round/RoundManagerImpl.java#L225C17-L225C81`

Uso di libreria di terze parti

Utilizzata nelle GUI per effettuare il resize della schermate. Link alla libreria generata come classe final. `https://github.com/PulgaLuca/OOP23-unibo-td/blob/main/src/main/java/it/unibo/model/utilities/ScaledImage.java` Link di esempio utilizzo. `https://github.com/PulgaLuca/OOP23-unibo-td/blob/f84040bdc412c3abbb06a344ec0ae82e5af06e6f/src/main/java/it/unibo/view/GuiStart.java#L130`

3.2.2 Marco Casali

Utilizzo della libreria SLF4J

Utilizzata in vari punti. Un esempio: https://github.com/PulgaLuca/OOP23-unibo-td/blob/4601e6548d51942648ab1356e79d8f62a87682a8/src/main/java/it/unibo/model/map/GameMapFactoryImpl.java#L81C13-L81C75

Utilizzo di lambda expressions e Stream

Utilizzate in vari punti. Un esempio: https://github.com/PulgaLuca/OOP23-unibo-td/blob/4601e6548d51942648ab1356e79d8f62a87682a8/src/main/java/it/unibo/model/map/GameMapFactoryImpl.java#L123-L125

Utilizzo della libreria JSON

Un esempio: https://github.com/PulgaLuca/OOP23-unibo-td/blob/74113621fccddd231d338
src/main/java/it/unibo/model/map/GameMapFactoryImpl.java#L52-L56

3.2.3 Cristina Murvai

Utilizzo di Optional

Un esempio: https://github.com/PulgaLuca/OOP23-unibo-td/blob/396dd9934c61e72d3e4ca
src/main/java/it/unibo/model/entities/enemies/EnemiesManagerImpl.java#L168-L171

Utilizzo di Stream

Utilizzati in vari punti, ad esempio: https://github.com/PulgaLuca/OOP23-unibo-td/blob/396dd9934c61e72d3e4ca269508d5e62017b6ce6/src/main/java/it/unibo/model/entities/enemies/EnemiesManagerImpl.java#L188-L196

Utilizzo della libreria Json per interazione con file JSON

Un esempio: https://github.com/PulgaLuca/OOP23-unibo-td/blob/396dd9934c61e72d3e4ca
src/main/java/it/unibo/model/entities/enemies/EnemiesConfigFactoryImpl.java#L67-L85

Utilizzo della libreria SLF4J

Un esempio: https://github.com/PulgaLuca/OOP23-unibo-td/blob/396dd9934c61e72d3e4ca
src/main/java/it/unibo/view/enemies/EnemiesPanel.java#L61-L63

3.2.4 Luca Pulga

Utilizzo di Optional

Utilizzati in vari punti. Un esempio: https://github.com/PulgaLuca/OOP23-unibo-td/blob/12d7eec3c7e4b474e6f315461642e1c6a2d6c693/src/main/java/it/unibo/model/entities/defense/tower/target/DistanceBasedTargetSelect
java#L24

Utilizzo di generici

https://github.com/PulgaLuca/OOP23-unibo-td/blob/12d7eec3c7e4b474e6f315461642e1c
src/main/java/it/unibo/model/entities/EntityFactoryImpl.java#L40

Utilizzo di lambda expressions

Utilizzate in vari punti. Un esempio: https://github.com/PulgaLuca/OOP23-unibo-td/blame/12d7eec3c7e4b474e6f315461642e1c6a2d6c693/src/main/java/it/unibo/model/entities/defense/manager/DefenseManagerImpl.java#L96

Utilizzo della libreria SLF4J

Utilizzata in vari punti. Un esempio: https://github.com/PulgaLuca/OOP23-unibo-td/blame/12d7eec3c7e4b474e6f315461642e1c6a2d6c693/src/main/java/it/unibo/model/entities/defense/manager/DefenseManagerImpl.java#L29C43-L29C43

Utilizzo della libreria Jackson per interazione con file JSON

Un esempio: https://github.com/PulgaLuca/OOP23-unibo-td/blob/12d7eec3c7e4b474e6f3
src/main/java/it/unibo/model/entities/EntityFactoryImpl.java#L66

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Francesco Aurora

Sono soddisfatto del mio lavoro svolto all'interno del gruppo. Nonostante i miei impegni lavorativi, sono riuscito a contribuire in modo significativo e a dare una mano quando necessario. Tuttavia, ci sono stati diversi aspetti negativi dal punto di vista del coordinamento del gruppo. Pur essendo abituato a lavorare in team, questa è stata un'esperienza particolarmente difficile, probabilmente a causa delle diverse esperienze dei membri del gruppo. Nonostante ciò, sono estremamente soddisfatto, soprattutto perché non conoscevo nessuno del gruppo e siamo comunque riusciti a contribuire in modo significativo al progetto. Guardando al futuro, le competenze che ho appreso mi saranno molto utili.

Per quanto riguarda lo sviluppo del codice, è stato interessante vedere tutti gli errori che sono stati commessi e che venivano generati durante il build. Questo mi ha fatto capire quanti errori possono essere generati da una programmazione non corretta, e non me lo aspettavo. È stato anche molto interessante risolvere errori di bug, soprattutto perché questi richiedevano che il codice venisse a volte completamente stravolto. Ad esempio, la creazione dei thread tramite parametro lambda per il passaggio di parametri non duplicati nella classe è stata una sfida significativa.

Sono soddisfatto del lavoro che ho realizzato. In ottica futura, ci sono molti miglioramenti che si potrebbero apportare per rendere il gioco più accattivante e performante, almeno graficamente. Ad esempio, la gestione degli effetti sonori e altri dettagli. Sei mesi per lo sviluppo del progetto sono molti, ma, da quel che ho capito, non è realistico pensare di sviluppare tutto quello che è stato fatto in sole 80 ore.

Marco Casali

Il risultato del progetto è abbastanza soddisfacente, siamo riusciti a implementare i requisiti minimi in tempo e a parte alcune imprecisioni grafiche il gioco gira come previsto. Sicuramente non è stata sufficiente la parte iniziale di progettazione e questo è risultato in un'architettura poco efficace, ad esempio per la lettura dei file JSON o più in generale delle risorse ogni membro ha implementato per sè e questo ha portato a ripetizione di codice. Comunque è stata un'esperienza molto formativa dal punto di vista dello sviluppo in team e del design che mi ha portato ad affrontare nuovi progetti con idee migliori.

Cristina Murvai

Questo progetto è stata sia la mia prima esperienza di teamwork che di sviluppo di un'applicazione complessa in java, che prima di questo corso non avevo mai utilizzato. In particolare sono contenta di come sono riuscita a individuare e sfruttare alcuni dei pattern visti a lezione come Factory e Observer e di aver sfruttato in modo utile diversi elementi più avanzati del linguaggio java quali Stream e Optional. Non sono invece troppo soddisfatta della resa grafica in quanto avrei voluto trovare un modo per far si che i nemici seguissero più precisamente il percorso sulla mappa. Una della challenge principali è stata quella di confronto e coordinazione con gli altri membri del gruppo che, prima di questo progetto, non conoscevo personalmente. A mio parere avremo potuto comunicare di più per gestire alcuni aspetti comuni in maniera più efficiente. Complessivamente sono discretamente soddisfatta del lavoro svolto, ma sono consapevole che alcuni aspetti si sarebbero potuti gestire in maniera diversa. In futuro cercherò di migliorare le mie skill di teamworking e programmazione object oriented.

Luca Pulga

Prendere parte a questo progetto è stata un'attività non semplicissima, per via degli impegni lavorativi e della coordinazione nel gruppo, ma decisamente formativa, soprattutto in ottica futura. La progettazione della parte difensiva del gioco ha ricoperto un ruolo massivo dell'applicazione e ho cercato non solo di cimentarmi nella mia parte di difese, ma anche nel cercare una generalizzazione relativamente alle varie tipologie di entità presenti nel gioco. La soluzione adottata rende estendibile il codice in caso di futuri sviluppi, attraverso l'astrazione delle entità e all'utilizzo di Design Patterns fondamentali per rendere il codice maggiormente leggibile e scalabile. Purtroppo non sono riuscito a sfruttare a pieno la programmazione funzionale e altre tecniche di

programmazione illustrate durante il corso, che mi sarebbe interessato implementare e progettare, ma che porto ora con me nel mio bagaglio formativo. Infine direi che sono comunque soddisfatto, non solo di aver acquisito nuove competenze, ma anche di aver lavorato in un team nuovo.

Capitolo 5

Appendice A - Guida Utente

5.0.1 Launcher

All'avvio dell'applicazione si aprirà la schermata di anteprima del gioco. Cliccare il bottone "Start" per iniziare.

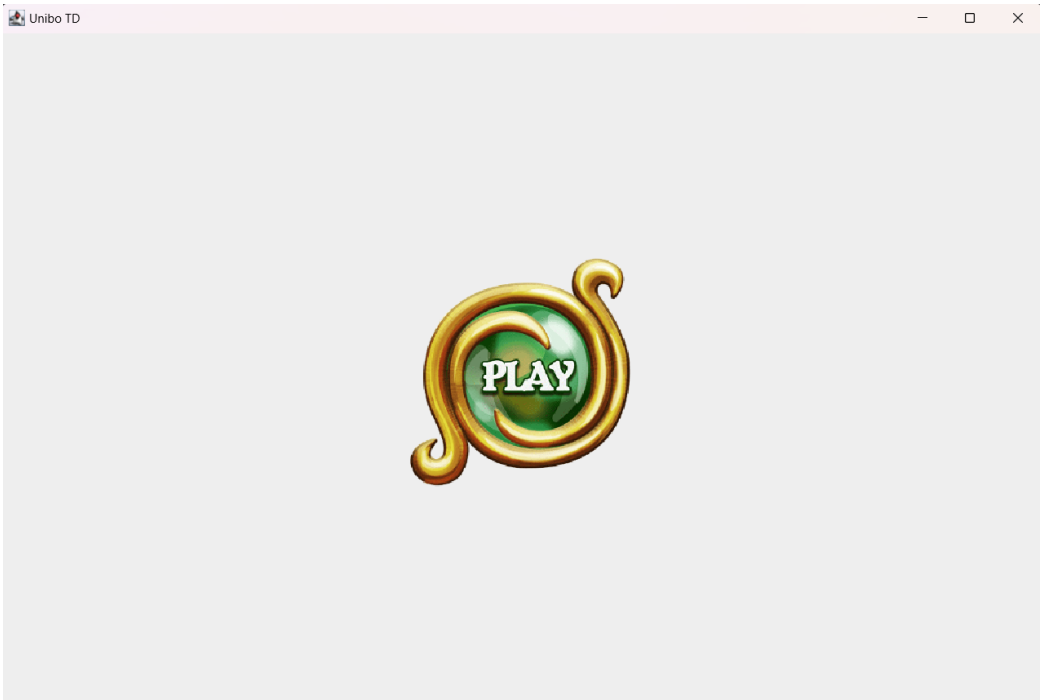

Figura 5.1: Schermata di gioco iniziale - Bottone Start

5.0.2 Selezione Mappa

E' possibile selezionare la mappa in cui si desidera giocare. Cliccare sulle frecce per scorrere tra le mappe disponibili. Per selezionare una mappa, cliccare sulla mappa desiderata.

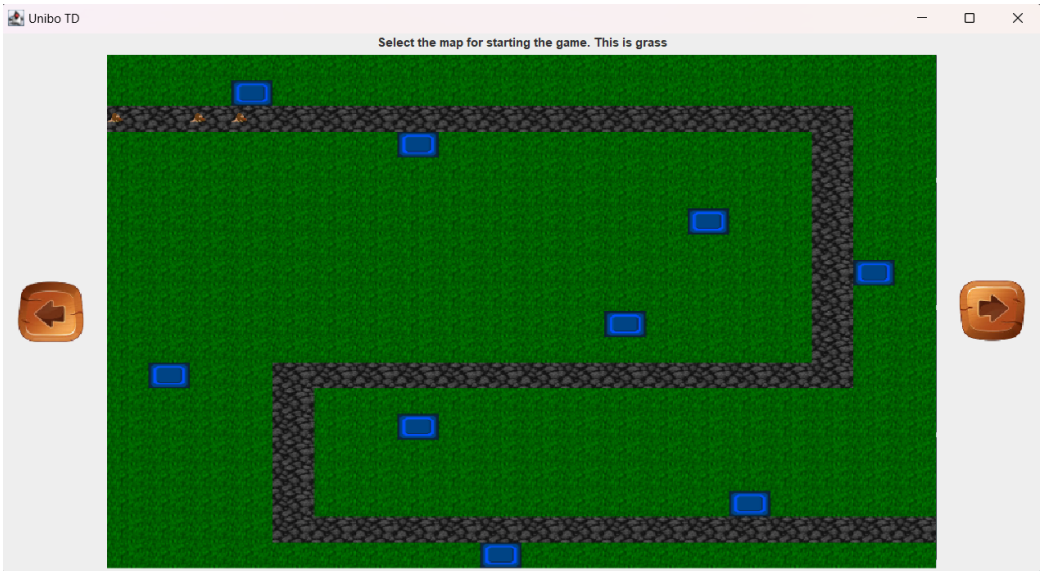

Figura 5.2: Schermata di selezione mappa

5.0.3 Schermata di inizio gioco

Verrà caricata la mappa su cui sarà possibile posizionare le torri per sconfiggere le ondate di nemici. Per posizionare una torre, cliccare sulla torre desiderata. La torre selezionata verrà segnalata con un rettangolo rosso. Cliccare su un rettangolo blu della mappa per posizionare la torre correntemente selezionata.

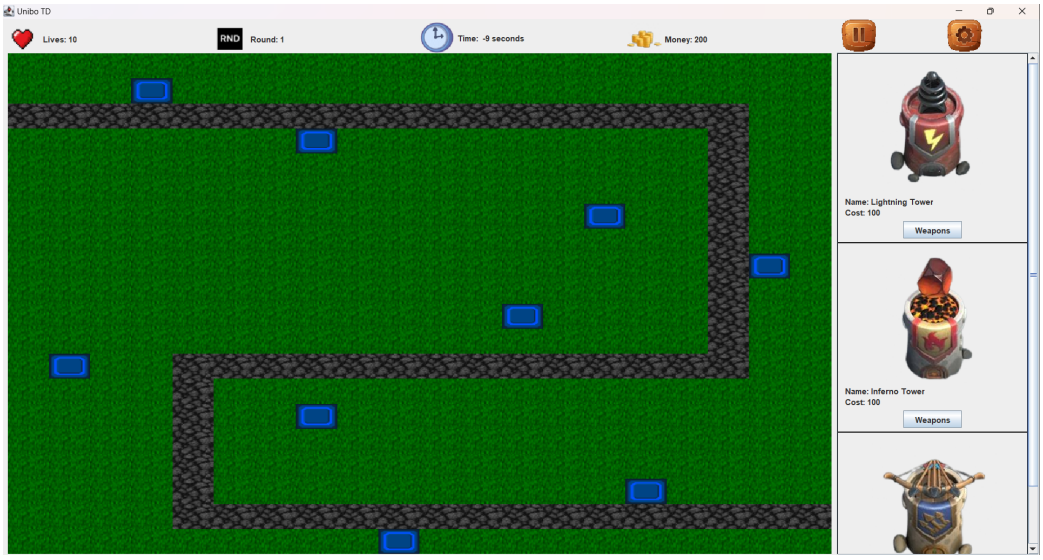

Figura 5.3: Schermata di inizio gioco

5.0.4 Posizionamento torri

Esempio di posizionamento torri e ondata di nemici in corso.

Figura 5.4: Schermata di inizio gioco

5.0.5 Armi a disposizione

Ogni torre dispone di armi diverse con cui sferrare i propri attacchi. Ogni torre dispone di più armi ma, per semplicità, i proiettili sparati da ognuna di esse, attualmente, arrecano lo stesso danno ai nemici. L'arma di ogni torre decreta la frequenza di sparo dei proiettili.

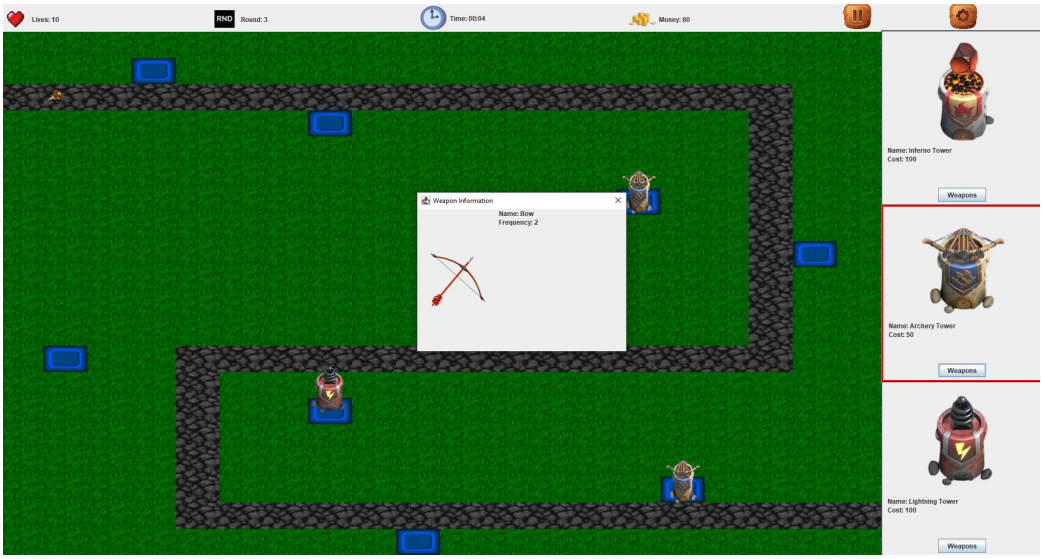

Figura 5.5: Armi a disposizione