Base Befender

Matteo Bambini, Davide Baldelli, Daniel Guariglia, Shola Oshodi

Aprile 2021

Indice

1 Analisi

1.1 Requisiti

Il software implementato mira alla costruzione di un videogame di tipo Tower Defense dal nome "Base Defender". Il giocatore per avanzare di livello dovrà fronteggiare invasioni di nemici acquistando e posizionando torri difensive sulla mappa allo lo scopo di proteggere la propria base.

1.1.1 Requisiti Funzionali

- Base Defender avrà più scenari di gioco, le Mappe, queste saranno composte da un percorso che i nemici nel corso dei loro attacchi cercheranno di completare. Avanzando di livello la difficoltà del gioco dovrà essere via via crescente. L'utente dovrà poter selezionare quale mappa di gioco utilizzare.
- L'uccisione di un nemico dovrà premiare l'utente con nuove monete, ogni nemico che completerà il percorso invece dovrà sottrarre al giocatore dei punti vita. Il Gioco dovrà terminare quando non si avranno più punti vita.
- Sarà presente uno Shop dal quale l'utente potrà acquistare nuove torri. Base Defender dovrà essere in grado di gestire in modo opportuno gli acquisti, gli upgrade delle torri e l'incremento dei soldi.
- Dovrà inoltre sempre essere possibile posizionare le torri nella mappa. E quest'ultime dovranno essere in grado di arrecare danno ai nemici in base alle caratteristiche con le quali sono definite.

1.1.2 Requisiti non funzionali

- Il software dovrà adattarsi alle dimensioni dello schermo in cui viene eseguito, di conseguenza tutta l'interfaccia deve essere scalabile in base alle dimensioni richieste.
- Il software dovrà gestire i thread in modo da non essere eccessivamente oneroso di risorse ma essere allo stesso tempo fluido e responsivo. Sarà inoltre necessario gestire in maniera ottimale accessi concorrenti a variabili non thread safe.

1.2 Analisi e modello del dominio

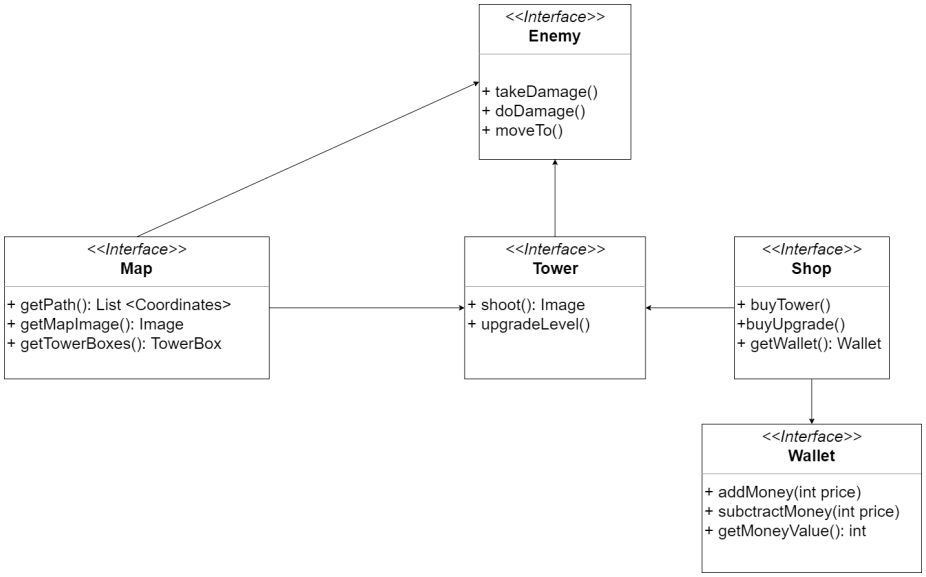


Figura 1.1: Schema UML del modello del dominio

Il software deve essere in grado di gestire la mappa di gioco, all'interno della quale sarà definito il percorso su cui si sposteranno i nemici e le celle in cui potranno essere piazzate le torri.

I nemici dovranno essere in grado di seguire il percorso determinato dalla mappa e dovranno avere dei punti vita. Sarà necessario gestire l'eventualità nella quale i nemici arrivano alla fine del percorso e di conseguenza sottraggono punti vita al giocatore.

Le torri dovranno avere una logica grazie alla quale selezioneranno e colpiranno il loro bersaglio. Dovranno essere inoltre potenziabili in modo da aumentarne l'efficacia. Ci saranno tre tipologie di torri con caratteristiche differenti.

Il gioco dovrà inoltre gestire lo shop in cui sarà possibile acquistare nuove torri e potenziamenti.

A ogni cambiamento apportato dei nemici (fine percorso o uccisione) dovremo essere in grado di ricalcolare vita e soldi e di gestire di conseguenza i round del gioco.

2 Design

2.1 Architettura

L'architettura di Base Defender segue il pattern architetturale MVC. A seguito della scelta della libreria Grafica (JavaFx) abbiamo identificato un unico entry point per la view e per il controller: la classe AppView (Figura 2.1). Il progetto può essere diviso in due macro sezioni: la sezione del menu: necessaria per iniziare la partita e visualizzare le statistiche di gioco (Figura 2.1) e la sezione del game (Figura 2.2).

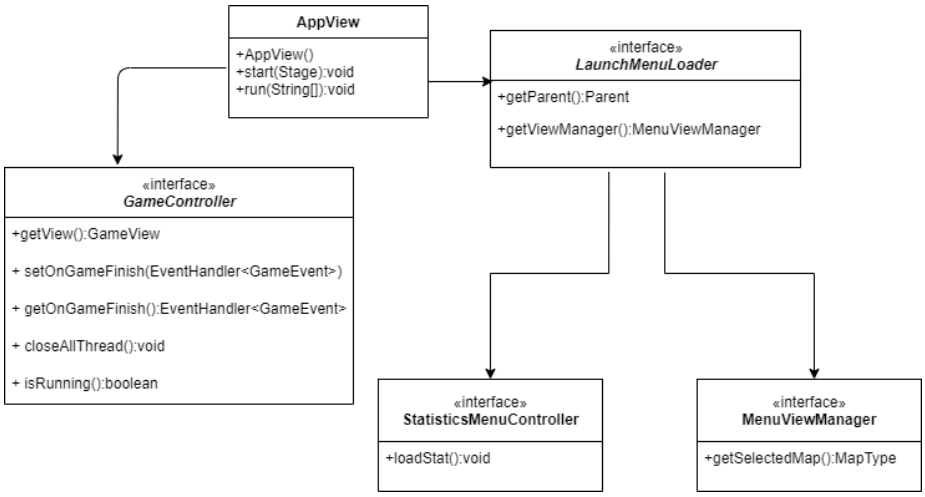


Figura 2.1: Schema UML architettura Menù

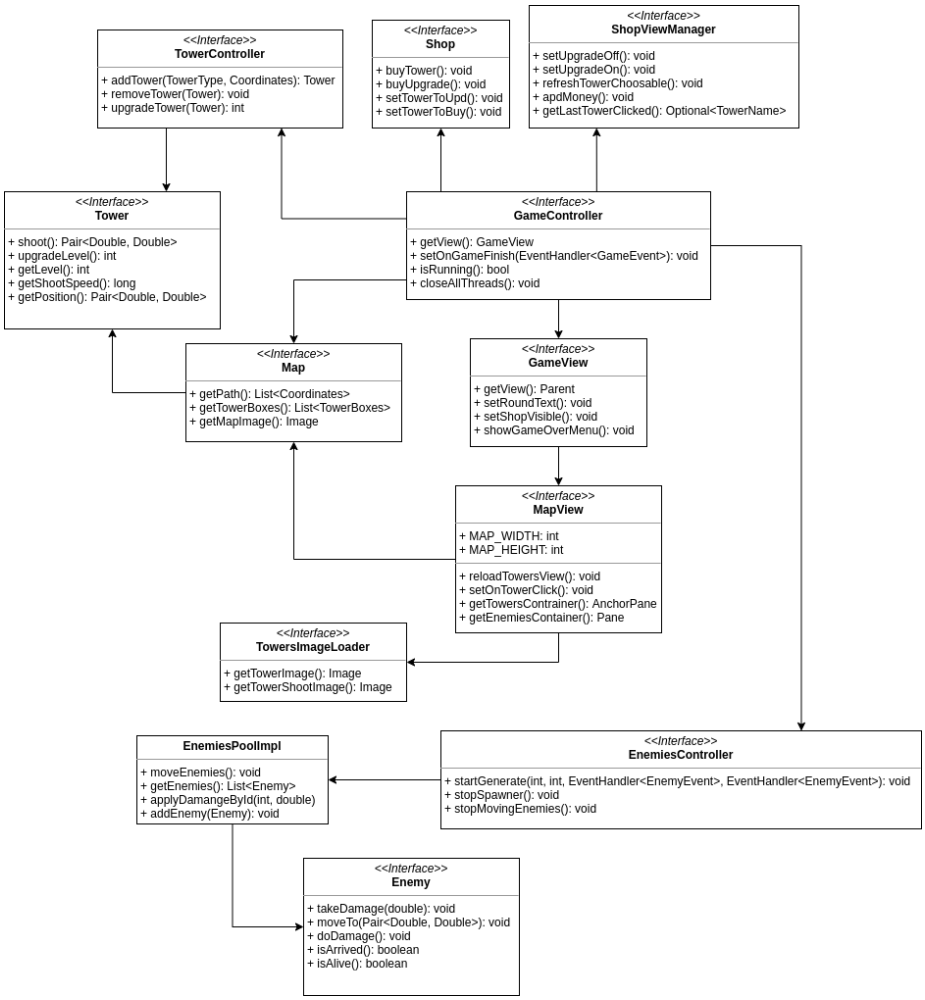
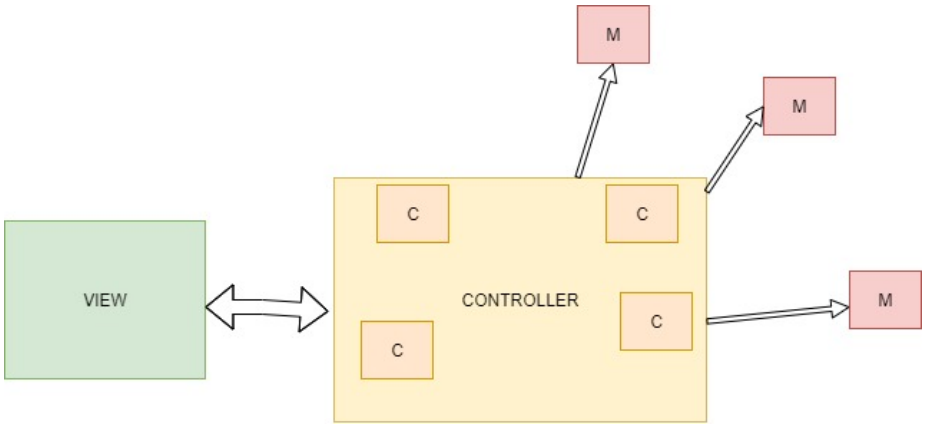


Figura 2.2: Schema UML architettura game

Base Defender non possiede un entry point per Model in quanto non è presente un'unica classe centralizzata ma piuttosto diversi model indipendenti che vengono caricati ed usati nei rispettivi controller. In un'ottica MVC possiamo identificare come controller principale la classe `GameController` che comunica con `GameView` che rappresenta la View principale della nostra architettura. `GameController` gestisce tutti i controller delle principali sezioni del programma che a loro volta interagiscono con i Model di loro competenza. 2.1



Ogni componente del gruppo si è impegnato nel rispettare il più possibile il pattern architetturale MVC per garantire una maggiore riusabilità delle classi e maggiore indipendenza dalla libreria grafica. A seguito di diversi accorgimenti sono state progressivamente eliminate le dipendenze dirette alla libreria grafica dal controller di Gioco (`GameControllerImpl`) permettendo di sostituirla mantenendo la parte logica inalterata. `GameControllerImpl` infatti non utilizza mai direttamente metodi o costrutti di JavaFX, ma delega ogni operazione di tipo grafico alle classi implementate che come anticipato precedentemente si pongono l'obiettivo massimizzare l'intercambiabilità.

2.2 Design di Dettaglio

2.2.1 Shola Oshodi

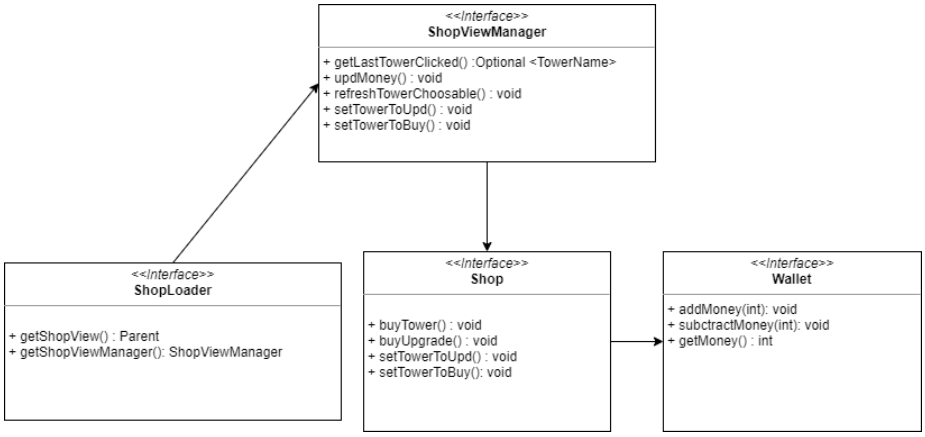


Figura 2.3: Schema UML delle Classi Shop e Wallet

Shop e Wallet
Successivamente alla fase di analisi è iniziato lo sviluppo individuale, dovendo io gestire principalmente la parte riguardante l'acquisto delle Torri, degli Upgrade e dell'accredito delle monete a seguito dell'uccisione dei nemici, ho iniziato progettando logicamente la struttura delle classi per poi iniziarne l'implementazione. Ho ritenuto importante separare le operazioni svolte sul credito utente dalle operazioni che riguardavano più nello specifico la logica dello Shop, per questo ho creato la classe Wallet che sarà gestita in modo opportuno dallo Shop e che ne richiamerà i metodi qualora si desiderasse incrementare-decrementare le monete o verificare se si dispone di credito a sufficienza per compiere determinate operazioni. In seguito mi sono concentrata sulla realizzazione dell'interfaccia grafica, non conoscendo in maniera approfondita la libreria scelta è stato necessario studiare più attentamente JavaFX arrivando infine al completamento della GUI modellandola grazie all'uso del FXML. Ho cercato nella sviluppo di questa parte di progetto di rispettare quanto più possibile il pattern MVC gestendo la suddivisione come mostrato in figura 2.4, aumentandone così la modularità e in modo che anche in futuro si possano riutilizzare le classi e/o modificare la libreria grafica. **Shop e Wallet**

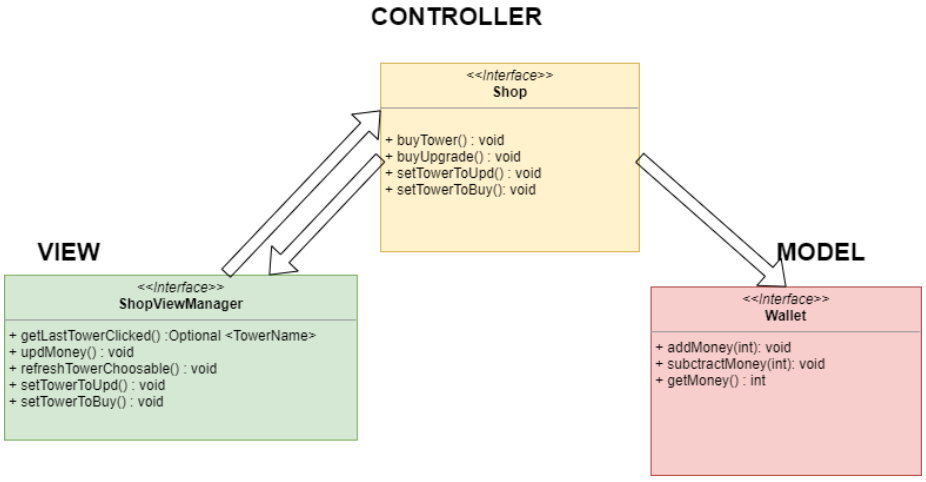


Figura 2.4: Divisione MVC dello Shop

L'interfaccia della parte View è ShopViewManager la cui implementazione si occupa della gestione di label e bottoni e che comunica con il controller Shop al fine di notificargli il verificarsi dei trigger eventi successivamente ai quali Shop svolgerà diverse operazioni che andranno aggiornare la classe di Model. Avendo necessità di conoscere le azioni utente nell'interfaccia grafica (per conoscere il tipo di torre scelta ad esempio) si è scelto di utilizzare il pattern Observer definendo la classe ShopManagerViewImpl come observable e Game controller come observer, 2.5 . si era scelto inizialmente di utilizzare la gestione eventi di JavaFX, essendoci poi accorti della forte dipendenza che avremmo creato nel controller abbiamo pensato ad un alternativa che ci permettesse di modificare il codice prodotto il meno possibile. Per questo motivo dopo svariate i potesi abbiamo deciso di utilizzare una nostra classe EventHandler.

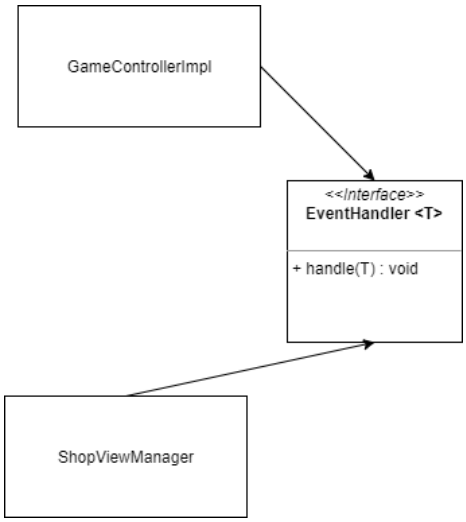


Figura 2.5: Divisione MVC dello Shop

Nello schema UML mostrato in figura possiamo inoltre visionare l'interfaccia funzionale ShopLoader per mezzo della quale viene caricato l'FXML, il cui unico metodo restituisce la View caricata con lo scopo di aggiungere il Pane dello Shop alla UI di gioco senza appunto inserire direttamente riferimenti a JavaFX in Game Controller.

Game Controller Un volta completata la parte individuale mi sono dedicata, in collaborazione con Daniel Guariglia, all'implementazione del controller di Gioco, tra i nostri obiettivi c'erano quelli di sfruttare al meglio le Interfacce di Torri, Nemici, Mappa e Shop in modo che il controller fosse quanto meno caotico possibile e di inserire tutti gli aspetti necessari alla gestione e alla meccanica di gioco. Avendo ogni membro del gruppo cercato di incapsulare al meglio la logica implementativa della sua sezione e di offrire interfacce e metodi che fossero di facile comprensione siamo riusciti a coordinare i diversi componenti utilizzando nel controller solo poche classi principali. GameController nello specifico comunica la classe gameView per generare l'interfaccia grafiche di gioco dotata di tutte le sezioni principali. Si è scelto di gestire gamecontroller con le seguenti associazioni. [2.6] Come anticipato e mostrato precedentemente Game Controller sfrutta il pattern Observer avendo un ruolo centrale nella gestione degli eventi.

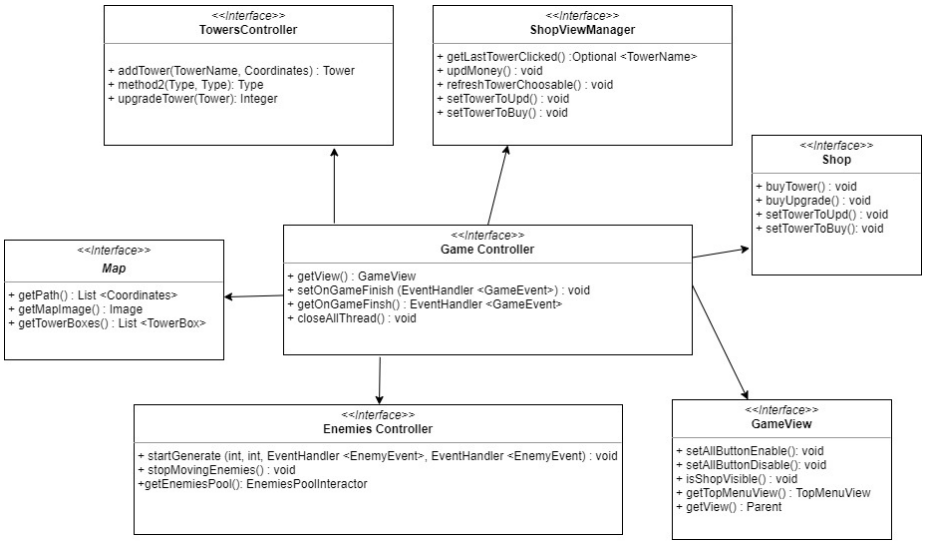


Figura 2.6: Schema UML GameController e relazioni

2.2.2 Daniel Guariglia

Menù:
Inizialmente mi sono occupato di aspetti non implementativi, legati all'aspetto grafico del programma, data l'inesperienza le prime giornate sono state dedicate alla ricerca di una metodo efficace per la creazione delle mappe, un modo per codificarne il percorso, lo studio di JavaFX e del linguaggio di markup FXML.

Seguendo la suddivisione del lavoro mi sono poi occupato della creazione del menù, diviso in menù principale e menù delle statistiche, adottando come pattern architetturale MVC, per realizzare le parti di View ho fatto uso del software SceneBuilder, ogni View in FXML ha un rispettivo manager che ho realizzato adottando il pattern `Observer` per notificare i click sull'interfaccia grafica, posso identificare come Observed MenuManagerImpl e come Observer AppView.

Ispirandomi al pattern Proxy ho creato una interfaccia (LaunchMenuLoader) e una sua implementazione che si occupa di caricare dalle risorse il file FXML e di istanziare il rispettivo manager passandoli gli eventi da richiamate, questa soluzione viene anche usata per caricare il file fxml del menù delle statistiche e il rispettivo controller, permettendo così di caricare il menù completo usando solo la classe LaunchMenuLoaderImpl. 2.7

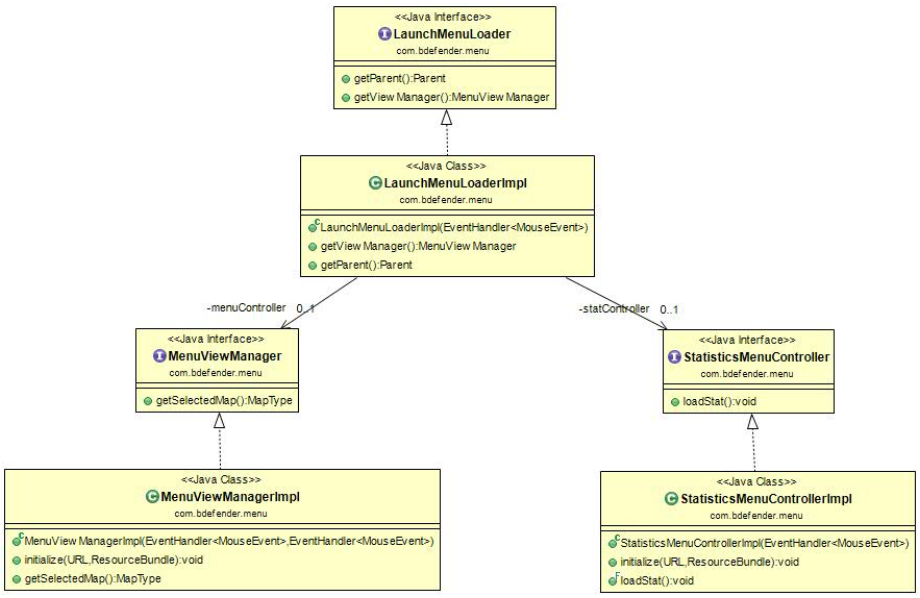


Figura 2.7: Schema UML gestione menù principale e menù delle statistiche

Game Controller:
La realizzazione del controller di gioco è senza dubbio la parte che ha richiesto più comunicazione e collaborazione con i membri del gruppo, per questo in fase di analisi abbiamo assegnato questo lavoro a più di una persona, nello specifico della sua realizzazione ce ne siamo occupati principalmente io e Oshodi.

Dopo aver aspettato che i membri del gruppo avessero realizzato le rispettive interfacce abbiamo iniziato a realizzare l'interfaccia GameController e la sua implementazione GameControllerImpl, in questa fase io e Oshodi ci siamo divisi i metodi da realizzare, nello specifico io mi sono occupato dell'implementazione della mappa, la logica di posizionamento delle torri, aspetti di gameplay e la gestione dei punti vita del giocatore. Per rispettare il pattern MVC è stata necessaria la creazione della classe GameView che incapsula la parte di view ed espone metodi per interagire con essa, quest'ultima ci consente di isolare la dipendenza da JavaFX non importando nel controller di gioco.

Il GameController di gioco funge da Obderver per svariate classi che implementa, nello specifico per: shopManagerView per definire l'azione per chiudere lo shop, GameView per definire molte azioni del top menu e EnemiesController per definire le azione da svolgere alla morte o al completamento del percorso di un nemico. Figura 2.6

Statistiche:
Dopo aver sviluppato con i miei colleghi una prima versione del GameController sopra citato, ho sviluppato la logica di salvataggio delle statistiche di gioco,

decidendo di dividerla in lettura (StatisticsReader) e scrittura (StatisticWriter), aspetti che sono necessariamente implementati separamente in quanto la lettura sarà effettuata al caricamento del menù e la scrittura a inizio e fine partita. Tra i miei obiettivi c'era quello di memorizzare permanentemente le statistiche delle partite, così da poterne estrarre informazioni elaborate come il tempo totale di gioco e la mappa più giocata, per questo ho ritenuto necessario l'uso della lettura e scrittura di file. Ho iniziato implementando la classe StatisticWriterImpl che ha il compito di riportare su file le statistiche calcolate, qui ho sfruttato il pattern creazionale `|Builder|`per garantire la completezza dei dati al momento del salvataggio, evitando così il salvataggio di partite senza inizio, senza round di fine etc. 2.8

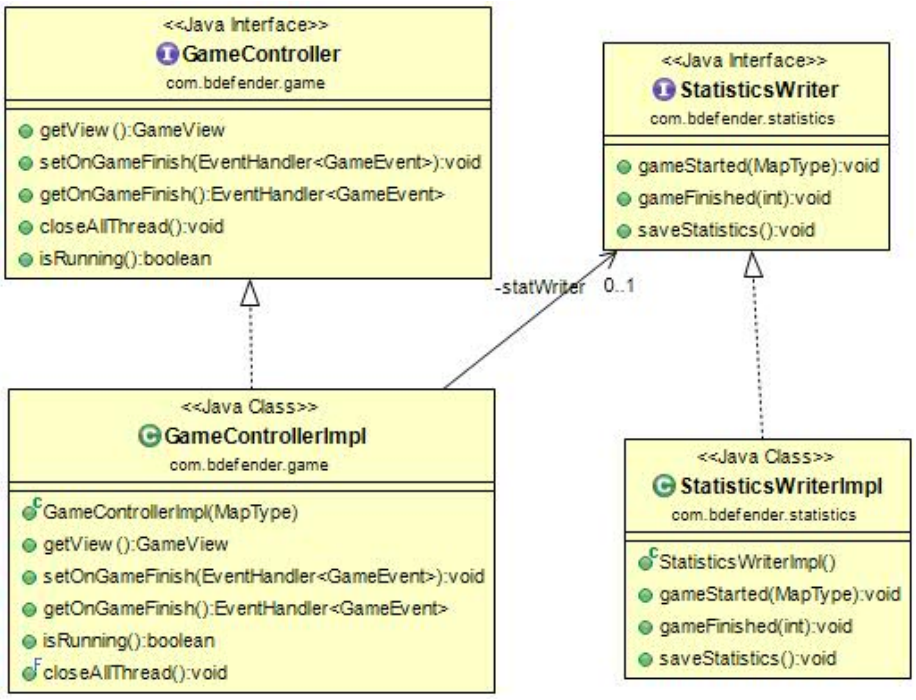


Figura 2.8: Nello schema UML è rappresentato il Writer delle statistiche e l'uso di GameControllerImpl

Per quanto riguarda la fase di lettura e rielaborazione ho realizzato l'interfaccia StatisticReader e la sua implementazione StatisticReaderImpl, quest'ultima realizzata con l'uso del pattern `|Singleton|`,usato per risolvere il problema di un potenziale accesso concorrente al file. 2.9

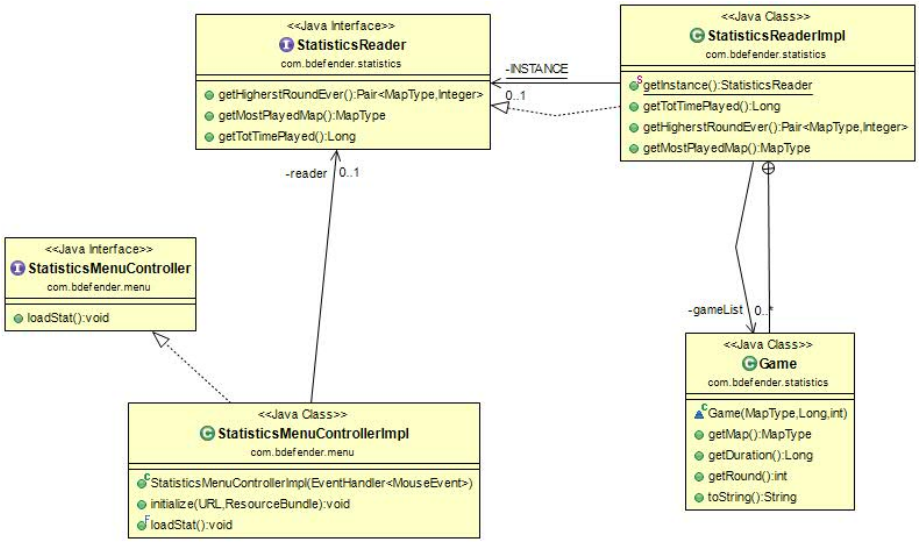


Figura 2.9: Nello schema UML è rappresentato il reader Singleton e il suo uso nel menu delle statistiche

2.2.3 Matteo Bambini

Mappa

La prima parte di cui mi sono occupato è quella relativa alla gestione della mappa. L'interfaccia `Map` è quella che descrive la struttura della mappa e che conserva tutti i dati relativi ad essa, ovvero è il model che descrive la mappa. Tutte le informazioni della mappa devono essere caricate da risorse sul filesystem e, per eseguire questa operazione, ho implementato la classe `MapLoader` (Figura 2.10) utilizzando il pattern Singleton. Per memorizzare le torri che vengono piazzate sulla mappa ho implementato la classe `TowerBox` (Figura 2.10), che permette di associare la torre alle coordinate.

Per semplificare la gestione delle posizioni degli elementi sulla mappa ho creato la classe `Coordinates` (Figura 2.10) che permette di gestire il posizionamento degli elementi sulla mappa in maniera semplice dal punto di vista dei controller ma contemporaneamente offre dei metodi per convertire le coordinate in pixel in modo che la view possa renderizzare il contenuto nel punto giusto.

La classe `MapView` (Figura 2.10) rappresenta la view principale della mappa. Dentro essa ci sono i container che conterranno le view delle torri e quelle dei nemici e inoltre contiene `TowerPlacementView`, ovvero l'interfaccia che permette di selezionare in quale `TowerBox` piazzare la torre. `MapView` si interfaccia con il controller utilizzando il pattern Observer. In particolare `MapView` è la classe observable e `GameControllerImpl` è la classe observer.

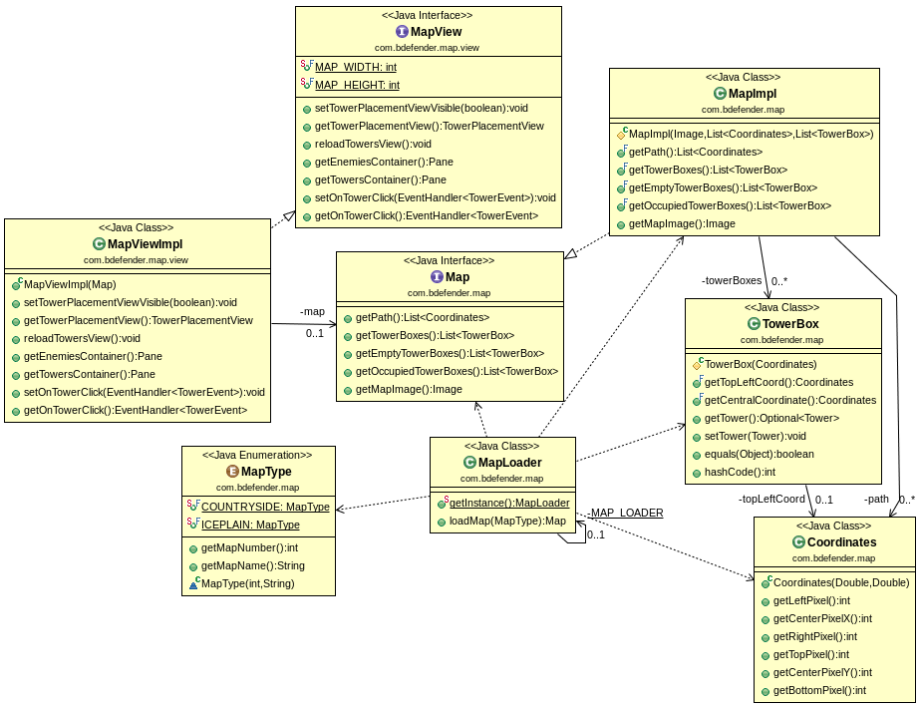


Figura 2.10: Schema UML gestione mappa

Eventi

Durante lo sviluppo ci siamo resi conto che le classi e le interfacce che stavamo usando per la gestione degli eventi erano dipendenti dalla libreria grafica (JavaFX). Di conseguenza, per rendere i controller indipendenti da quest'ultima, ho implementato il sistema di gestione degli eventi descritto in Figura 2.11. Questo sistema è strutturato esattamente come quello di JavaFX: ha l'interfaccia `Event` che identifica l'evento, l'interfaccia funzionale `EventHandler<T extends Event>` che rappresenta l'handler dell'evento e la classe `EventType<T extends Event>` che identifica il tipo dell'evento. La creazione di questo sistema mi ha anche permesso di creare classi di eventi dedicate a torri e a nemici in modo che l'oggetto (torre o nemico) che ha generato l'evento sia accessibile direttamente tramite i metodi delle classi `TowerEvent` o `EnemyEvent`.

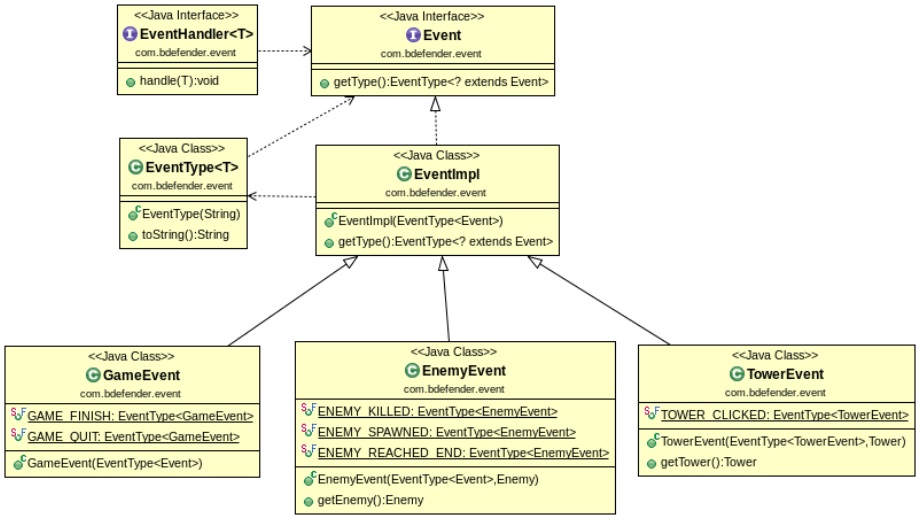


Figura 2.11: Schema UML gestione eventi

2.2.4 Davide Baldelli

Torri:
Per la realizzazione delle torri ho deciso di porre particolare attenzione alla riusabilità e all'estensibilità. Nelle prime fasi della progettazione mi sono posto due questioni fondamentali : la prima era quella di riuscire a creare versioni differenti delle torri che potessero comportarsi in maniera diversa l'una dall'altra, per questo ho utilizzato **Strategy** (Figura 2.12), nello specifico: il metodo per individuare il target migliore (nel caso di una torre a colpo diretto il nemico più vicino, nel caso di una torre con colpo a zona il nemico con più nemici attorno ecc.), e il metodo per applicare i danni (al singolo nemico, a più nemici).

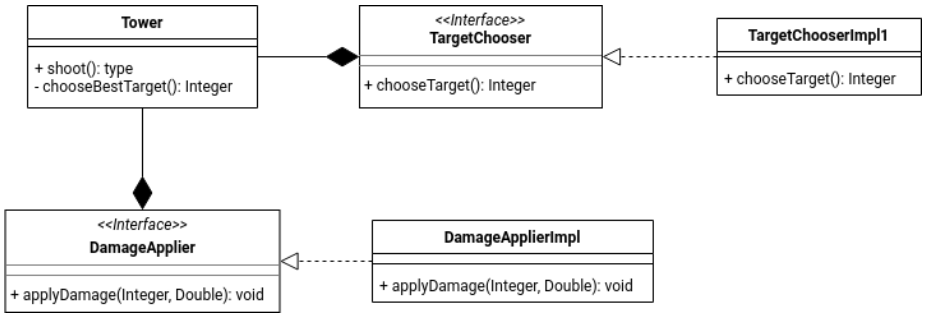


Figura 2.12: Schema UML della realizzazione di **Strategy** nelle torri

La seconda questione che mi sono posto era quella di fare in modo che l'implementazione della torre fosse indipendente dal quella dei nemici, perciò ho deciso di isolare le due implementazioni tramite il principio di **Dipendecy Inversion**, attraverso delle classi Interactor (Figura 2.13). Nello specifico queste classi si occupano di offrire metodi di utility alla torre, ad esempio, quello di restituire le posizioni di tutti i nemici, che verranno poi usate per calcolare il target migliore, oppure quello di restituire la posizione di un nemico dato il suo id. Gli Interactor sono utilizzati, inoltre, nelle loro implementazioni, per applicare i danni ai nemici.

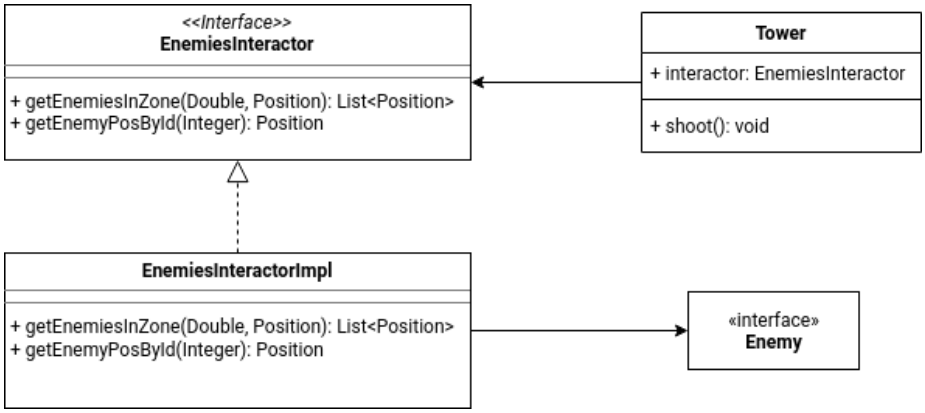


Figura 2.13: Schema UML della **Dependency Inversion** realizzata tra torri e nemici

Per rispettare il pattern **MVC** nel controller **TowerController** mi sono servito di un interfaccia funzionale (Figura 2.14), passata dal costruttore che mi permetteva di implementare le view delle torri senza importare dipendenze relative, appunto, all'implementazione della view.

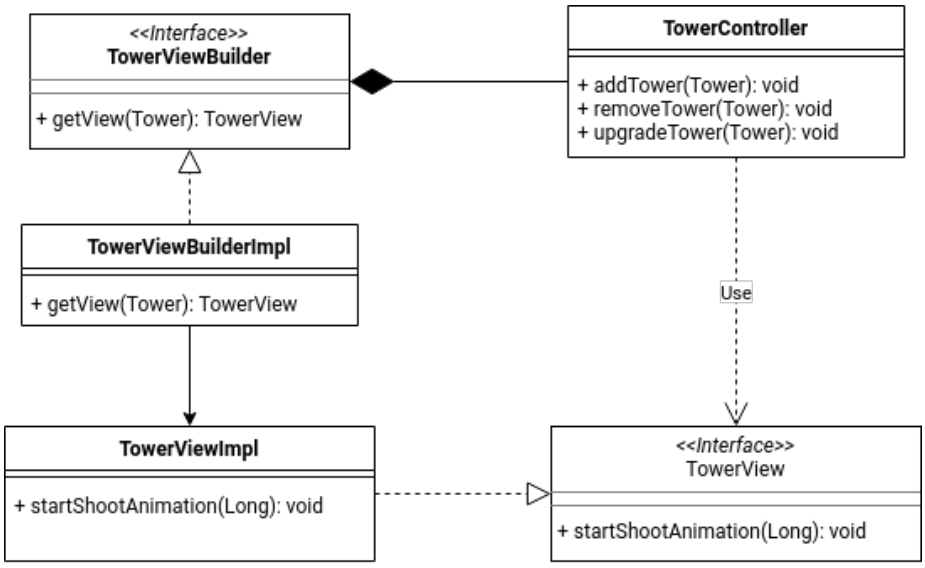


Figura 2.14: Schema UML della separazione della View e del Controller delle Torri

Nemici: Per la realizzazione dei nemici mi sono subito reso conto che per avere una gestione ottimale dei nemici avrei dovuto realizzare una classe con la quale potevo operare su tutti i nemici insieme in un unico punto. Ho scelto un approccio "seriale" rispetto a quello "parallelo" perché avere un thread per ogni nemico sarebbe stato molto difficile da gestire, e la limitata quantità dei nemici non avrebbe portato dei vantaggi in termine di prestazioni. Il lato negativo di questo approccio però era quello di creare una interfaccia piena di metodi che venivano utilizzati da parti diverse del programma per scopi diversi. Per questo motivo ho deciso di utilizzare la Interface Segregation (Figura 2.15), una tecnica che serve per esporre ai utilizzatori solo i metodi che utilizzano, aggiungendo così flessibilità e riusabilità al codice.

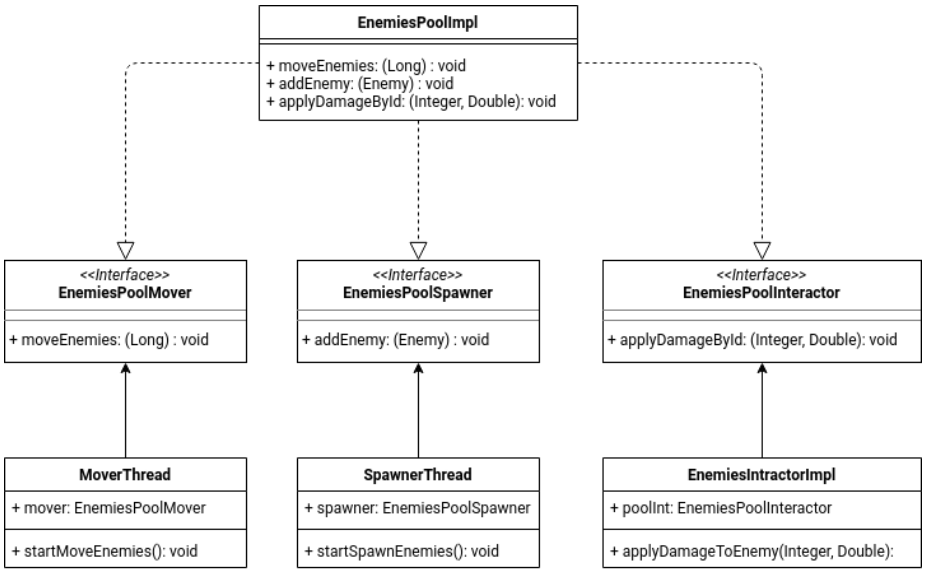


Figura 2.15: Schema UML delle Intrface Segregation applicata alla classe che gestisce le iterazioni coi nemici

Per rispettare il pattern MVC nel controller **EnemiesContoller** mi sono servito dell'interfaccia **EnemiesGraphicMover** (Figura 2.16), passata al costruttore, per implementare il movimento grafico dei nemici.

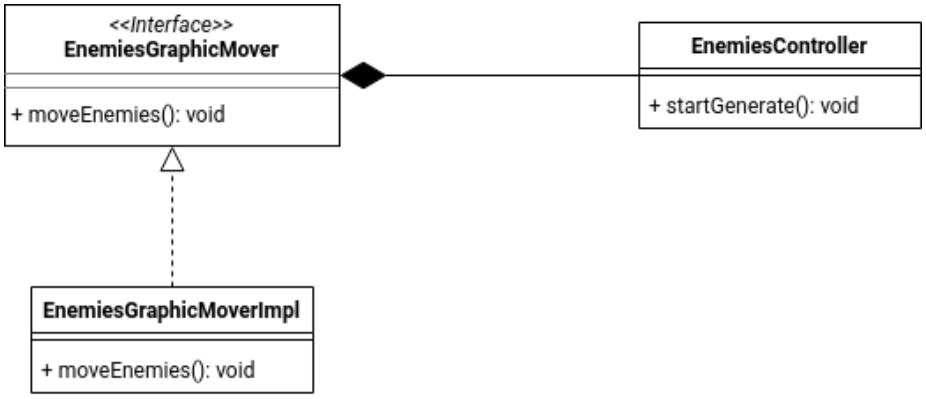


Figura 2.16: Schema UML di MVC applicato a **EnemiesController**

3 Sviluppo

3.1 Testing automatizzato

3.1.1 Mappa

- loadMapCountryside: testa il caricamento della mappa con indice 0
- loadMapIcePlain: testa il caricamento della mappa con indice 1
- testTowerPlacementBoxClick: testa il click su una casella to TowerPlacementView per verificare che ritorni il TowerBox corretto

3.1.2 Shop e Wallet

Per testare il corretto funzionamento della classe Shop e della classe Wallet sono stati implementati i seguenti test automatizzati.

- testBuyTower: verifica che l'acquisto delle torri si svolga correttamente e che non sia possibile acquistare torri quando non si possiedono monete a sufficienza.
- testWalletSubtraction: controlla che la Classe Wallet scali correttamente i soldi, senza raggiungere un saldo negativo.
- testWalletAddition: controlla che la Classe Wallet incrementi correttamente i soldi.
- testMixedOperation: si alternano operazioni di addizione e sottrazione per verificare che non ci siano errori nella gestione monete.
- testTowerIsCorrect (test grafico) : ci si accerta graficamente che i bottoni dello Shop aventi le immagini delle torri siano associati al giusto tipo di Torre.

3.1.3 Tower

- testTowerShoot: controlla che la torre individui i nemici correttamente e che colpisca il nemico più vicino.

3.1.4 Enemies

- testMovement: controlla che il nemico segua il percorso della mappa, eseguendo correttamente i cambi di posizione.
- testReachEnd: controlla che il nemico venga aggiornato correttamente lo stato del nemico esattamente quando arriva all'ultimo key point dell percorso

3.1.5 Statistics

Per testare il corretto funzionamento delle statistiche di gioco sono stati implementati i seguenti test automatici junit: StatisticsReaderTest

- **testDefaultValues:** testa il corretto funzionamento dei valori de default da caricare nel caso non esista nessuna statistica da caricare.

WriterThrowExcTest

- **testMissingStart:** testa il corretto funzionamento delle eccezioni nel caso si provi a salvare le statistiche di una partita senza prima averne dichiarato l'inizio.
- **testMissingFinish:** testa il corretto funzionamento delle eccezioni nel caso si provi a salvare le statistiche di una partita senza prima averne dichiarato la fine.

3.2 Metodologie di Lavoro

A seguito di un'approfondita analisi abbiamo definito la suddivisione del lavoro in modo che ognuno di noi potesse sviluppare una sezione di progetto in autonomia prevedendo poi in fase finale una più stretta collaborazione durante la quale avremo poi gestito le diverse integrazioni e modellato la logica di GamePlay. Il DVCS che abbiamo utilizzato è Git, e nello specifico GitHub: inizialmente per lo sviluppo individuale abbiamo lavorato tutti in branch separati facendo push e merge non appena si fosse realizzato qualcosa di pronto e funzionante, nella fase di integrazione abbiamo poi utilizzato un unico branch al fine di evitare il più possibile eventuali conflitti. Durante la fase di integrazione e test è stato fondamentale l'utilizzo di Jira, che ci ha permesso di tener traccia dei bug e dei task da completare prima del rilascio del software, permettendoci così di avere maggiore organizzazione e controllo. Per permettere una più facile collaborazione soprattutto nella realizzazione delle classi riguardanti nemici e torri si è cercato di incapsulare il più possibile la logica di gestione interna al fine di proporre poche interfacce che tutti i membri del gruppo potessero utilizzare senza preoccuparsi di ulteriori dettagli implementativi.

3.2.1 Matteo Bambini

Realizzazione della gestione degli eventi e del componente ImageButton. Realizzazione della maggior parte del model e della view della mappa e dell'interfaccia di posizionamento delle torri. Realizzazione in collaborazione con Davide Baldelli di EnemiesGraphicMoverImpl. Di seguito la lista delle classi realizzate da me nello specifico.
In autonomia:

- com.bdefender.event.*
- com.bdefender.map.Map

- com.bdefender.map.MapLoader
- com.bdefender.map.MapView
- com.bdefender.map.Coordinates
- com.bdefender.map.TowerBox
- com.bdefender.component.ImageButton
- com.bdefender.shop.TowerPlacementView

In collaborazione:

- com.bdefender.enemy.view.EnemiesGraphicMoverImpl

3.2.2 Daniel Guariglia

Ho realizzato in autonomia le classi per l'implementazione di Menu e Statistiche, in collaborazione per la creazione del Controller e View di Gioco e per la parte di model della mappa. Inoltre ho collaborato con i componenti del gruppo per l'implementazione di pattern, risoluzione di bug e Refactor. Di seguito la lista delle classi nello specifico:
In autonomia:

- com.bdefender.menu.*
- com.bdefender.statistics.*
- com.bdefender.map.MapType

In collaborazione:

- com.bdefender.game.GameControllerImpl
- com.bdefender.app.AppView
- com.bdefender.game.TopMenuView

3.2.3 Shola Oshodi

Implementate sia logicamente che graficamente tutte le parti riguardanti lo Shop, gli acquisti delle torri e degli upgrade e gestione dell'incremento delle monete (a seguito della sconfitta di un nemico). Implementata la classe Wallet, creato TowerName. Svolto in collaborazione il processo di integrazione e di sviluppo del Controller di Gioco. Collaborato con i componenti del gruppo nell'eseguire operazioni di refactoring, risoluzione di bug e definizione di dettagli implementativi. Di seguito la lista delle classi nello specifico:
In autonomia:

- com.bdefender.shop.*

- com.bdefender.wallet.*

In collaborazione:

- com.bdefender.tower.TowerName
- com.bdefender.game.GameControllerImpl
- com.bdefender.game.TopMenuView
- com.bdefender.game.GameView

3.2.4 Davide Baldelli

Realizzazione delle classi per la gestione logica dei nemici e le torri, nello specifico il movimento dei nemici e l'interazione con le torri, e le relative classi controller per la gestione delle funzionalità di gioco. Dal punto di vista della UI ho realizzato il caricamento delle risorse grafiche delle torri e dei nemici in autonomia, mentre la classe per il movimento grafico è stata realizzata in collaborazione. Di seguito la lista delle classi realizzate nello specifico:
In autonomia:

- com.bdefender.tower.interactor.*
- com.bdefender.tower.view.*
- com.bdefender.Tower
- com.bdefender.TowerFactory
- com.bdefender.enemy.pool.*
- com.bdefebder.enemy.Enemy
- com.bdefebder.enemy.EnemyFactory
- com.bdefebder.enemy.EnemyName
- com.bdefender.enemy.view.EnemiesViewLoader
- com.bdefebder.enemy.view.EnemiesGraphicMover

In collaborazione:

- com.bdefender.tower.TowerName
- com.bdefebder.enemy.view.EnemiesGraphicMoverImpl

3.3 Note di Sviluppo

3.3.1 Matteo Bambini

- **Progettazione con generics**: utilizzato per gestire gli EvenType
- **Uso di lambda expressions**: usate come event handler per la gestione degli eventi e all'interno degli stream per filtrare e mappare.
- **Uso di stream**: usati all'interno di Map per filtrare i towerbox e all'interno di MapView per mappare i towerbox in elementi renderizzabili.
- **Uso di Optional**: utilizzati in towerbox per identificare se la torre fosse presente oppure no evitando di utilizzare null.
- **Utilizzo di Build System**: Gradle
- **Librerie di terze parti**: JavaFX per la GUI, TestFX per i testi automatizzati della GUI

3.3.2 Daniel Guariglia

- **Uso della Lambda Expression:** usate per implementare gli eventi di gioco.
- **Uso delle Stream:** usate per il calcolo delle statistiche e per la chiusura dei Thread delle torri.
- **Uso di Optional:** Usati nella scrittura delle statistiche.
- **Utilizzo di Build System:** Gradle
- **Librerie di terze parti:** JavaFX

3.3.3 Davide Baldelli

- **Uso di Lambda Expressions:** utilizzate nei controller per implementare le view, evitando così di importare dipendenze relative al framework grafico scelto.
- **Utilizzo di stream:** utilizzati per filtrare la mappa contenente tutti i nemici in base al loro stato attuale.
- **Utilizzo di librerie esterne:** JavaFx per le implementazioni delle view dei nemici e delle torri.

3.3.4 Shola Oshodi

- **Utilizzo di lambda expressions**: per gestire gli eventi di gioco collegandoli opportunamente ai bottoni, e più in generale per compiere operazioni sui bottoni.
- **Utilizzo di Stream**: per filtrare l'enumeratore di Torri ed estrarre il costo dell'upgrade.
- **Utizzo di Optional**: per la gestione delle torri da comprare e su cui fare l'upgrade.
- **Utilizzo di Build System** Gradle
- **Librerie di terze parti** JavaFX, TestFX che ha permesso di simulare le interazioni grafiche in modo da poter sviluppare test automatizzati,
- **Uso di parti di libreria non spiegate a lezione**: FXML

4 Commenti Finali

4.1 Autovalutazione e lavori futuri

4.1.1 Davide Baldelli

Questo progetto per me è stato un occasione di imparare e migliorare l'attitudine al lavoro di gruppo. Spesso mi è capitato di portare avanti progetti personali,ma collaborare allo stesso progetto è un esperienza che mi ha arricchito e migliorato. Nelle fasi iniziali è stato difficile conciliare i vari punti di vista, ma dalle discussioni sono nate ottime soluzioni. Personalmente posso ritenermi soddisfatto del lavoro che ho svolto anche se forse avrei potuto produrre più funzionalità, come torri di tipi diversi o animazioni migliori, ma penso, tutto sommato di avere creato del codice di qualità, flessibile, e riutilizzabile. Nelle fasi di sviluppo ho cercato di essere sempre propositivo e penso che ognuno di noi abbia messo in gioco tutte le sue carte. Un punto dove sento di avere fallito è quello di avere avuto poca costanza, ho prodotto molto all'inizio e poi mi sono perso. Ideare e realizzare un progetto di queste dimensioni inoltre mi ha aiutato anche a migliorare la mia capacità di analisi, e grazie ad un buon lavoro fatto prima di scrivere effettivamente il codice, siamo riusciti portare avanti il progetto senza troppe difficoltà dal punto di vista architetturale.

4.1.2 Daniel Guariglia

Questo progetto rappresenta per me il primo software sviluppato con un gruppo e penso anche per gli altri componenti. Durante lo sviluppo abbiamo avuto molte occasioni di esporre le nostre opinioni su come organizzare/gestire le classi e come dividerci le parti, questa è probabilmente l'aspetto che più ci ha arricchiti, infatti ritengo che saper esprimere a voce la propria idea di organizzazione del

codice sia quasi più difficile dello sviluppo di esso. Sono sicuramente soddisfatto del prodotto finale, mentre per quanto riguarda il mio operato mi critico indubbiamente l'organizzazione del tempo, infatti ho aspettato troppo tempo prima di iniziare la mia parte e mi sono ritrovato a scrive codice all'ultimo.

4.1.3 Matteo Bambini

Credo di aver fatto, in generale, un buon lavoro considerando l'entità del progetto e le difficoltà nel coordinarsi con un gruppo di lavoro. Le parti che mi soddisfano di più sono quelle relative al model della mappa e alla gestione degli eventi. Credo che una cosa che avremmo potuto progettare meglio sia l'architettura e credo che avremmo potuto effettuare meglio la separazione tra libreria grafica e controller, anche se utilizzando JavaFX è molto complesso separare questi due elementi.

4.2 Shola Oshodi

Ritengo che questo progetto sia stato molto formativo soprattutto nell'apprendere come organizzare e gestire al meglio un progetto di dimensioni un po' più importanti di quelli su cui siamo solitamente abituati a lavorare e nel riuscire a coordinarsi con diversi membri del gruppo. Complessivamente sono soddisfatta della realizzazione del progetto anche perchè non pensavo saremmo effettivamente riusciti a realizzare un programma funzionante. Penso che un punto di debolezza possa essere l'organizzazione dell'architettura generale, questo forse perchè è stato un po' difficile avere da subito una visione d'insieme corretta, anche nella mia parte personale ho spesso cambiato idea su come gestire le diverse classi. Penso però nel complesso di aver prodotto un lavoro abbastanza pulito e comprensibile, ma sicuramente migliorabile, un po' anche a causa della pressione nel cercare di rispettare il termine di consegna. Mi sarebbe piaciuto sviluppare qualche funzionalità opzionale in più di quelle che ci eravamo proposti di implementare.

4.3 Difficoltà incontrate e commenti per i docenti

5 Appendice A

5.1 Guida Utente

Aprendo il gioco la prima schermata visualizzabile sarà quella del menu principale, dal quale sarà possibile: Iniziare una nuova partita, visionare il tutorial o selezionare una tra le possibili mappe di gioco oppure visualizzare le statistiche relative alle precedenti partite.

Figura 5.1: Menu di gioco

Premendo sul bottone "Inizia Partita" la schermata che si presenterà sarà la seguente

Figura 5.2: Inizio Partita

Qui potremo distinguere Mappa, Shop e Barra del Menù grazie alla quale sarà possibile rispettivamente aprire lo Shop, avviare il round, tornare al menù principale. Per poter fronteggiare gli attacchi i nemici sarà necessario costruire nuove torri e posizionarle strategicamente sulla mappa, per farlo sarà sufficiente cliccare su una torre dallo Shop e poi nel punto in cui si desidera collocarla. Se si desiderasse potenziare le tue torri in modo che infliggano danno maggiore ai nemici si potrà sceglierne una tra quelle già posizionate cliccandoci sopra e poi selezionare l'ultimo bottone dello Shop (upgrade). Una volta iniziata la partita (cliccando il bottone nella barra del menu) i nemici inizieranno l'attacco. La difficoltà dei livelli crescerà in modo progressivo.

Posizionando il cursore sopra a una torre comparirà un numero. Questo numero rappresenta il livello della torre (Figura 5.3).

Figura 5.3: Livello Torre