Laboratorio di Sistemi Software
Scala Tower Defense

Ismam Abu - ismam.abu@studio.unibo.it
Hamado Dene - hamado.dene@studio.unibo.it

Giugno 2022

1 Introduzione

Scala Tower Defense è un gioco di strategia del sotto genere *"Tower Defense"*. L'obiettivo del gioco è impedire ai nemici di attraversare la mappa costruendo torri che, sparando automaticamente, colpiranno i nemici in avvicinamento. Per poter costruire torri servono punti che si ottengono uccidendo i nemici.

Questo gioco è stato implementato in Scala, cercando di sfruttare al meglio i vantaggi del linguaggio.

2 Processi di sviluppo

Abbiamo adottato un processo di sviluppo ispirato a Scrum: questo framework è adatto a team di piccole dimensioni (ideale per noi 3) e favorisce lo sviluppo di software che potrebbe presentare difficoltà inattese grazie al suo processo d'implementazione.

Il ruolo di "Product Owner" è stato ricoperto da Ismam Abu, che ha diretto i vari meeting, creato gli sprint e prodotto dei report alla fine di essi.

2.1 Meeting

Il processo "scrum" sfrutta dei meeting a breve e a lungo termine per discutere il breve e il lungo periodo, favorendo una maggiore comprensione di ogni componente dell'applicazione da parte di tutti i membri. Nel nostro caso, si è deciso di fare dei meeting giornalieri e settimanali, nel primo caso sono stati fatti per sincronizzarsi sul lavoro degli altri, mentre i meeting settimanali per gestire i merge e le release delle varie versioni dell'applicazione. Tutti questi meeting sono stati svolti tramite il servizio di voice chat "Discord".

2.1.1 Meeting giornalieri

Durante i meeting giornalieri, ogni membro ha discusso di:

1. funzionalità a cui stava lavorando quel determinato giorno, non entrando troppo nei dettagli
2. dubbi riguardo a delle scelte che aveva fatto o che avrebbe dovuto fare
3. proprie opinioni sul proprio lavoro e l'andamento generale dello sviluppo dell'applicazione

Dopo il meeting giornaliero, se serviva, si faceva una discussione più tecnica per gestire alcuni aspetti chiave dell'applicazione, in maniera da evitare che rimanessero dubbi in sospeso. Il meeting durava dai 30 minuti alle 2 ore, non è stato fatto tutti i giorni (principalmente per motivi lavorativi) ma per un buon 80% eravamo sempre presenti.

2.1.2 Meeting settimanali

Abbiamo svolto 4 sprint settimanali, della durata di una settimana ognuno, ogni sprint rappresenta una release della nostra applicazione. Ogni sabato, al posto di fare il meeting giornaliero si è provvisto a fare retrospective dello sprint concluso per andare a pianificare quello successivo. Oltre questo generalmente si procedeva a fare il merge del branch dev con il main cosicché venisse prodotta in automatico la release.

Lo sprint retrospective consiste in:

1. analizzare il livello di completamento delle issue associate allo sprint settimanale

2. individuare delle issue che si sono rivelati più complicati del previsto

Lo sprint planning consiste in:

1. decidere quali issue avremmo dovuto inserire nello sprint successivo
2. creare su git le nuove issue con le relative specifiche discusse

Tipicamente i meeting settimanali duravano una mezza giornata abbondante, spesso anche una giornata intera.

2.2 Suddivisione delle issue

Le issue che formavano poi le varie versioni, sono state individuate suddividendo gli obiettivi da raggiungere per categoria, una volta ottenuti venivano assegnati alla persona a cui spettavano in base alla disponibilità, chiaramente per molte issue si sono messe insieme due persone per poter sviluppare al meglio certe funzionalità.

2.3 Tools

Gli strumenti e le tecnologie di cui ci siamo avvalsi durante lo sviluppo sono:

1. Framework "ScalaTest" per realizzare le unit test
2. GitHub, GitHub Actions, GitHub Project
3. Docker (local testing)
4. Gradle

2.4 Repository GitHub

Per facilitare lo sviluppo dell'applicazione abbiamo utilizzato il DVCS "git" e organizzato il suo workflow nel seguente modo:

1. il branch "main" è stato dedicato alle release principali dell'applicazione, rilasciate al termine degli sprint settimanali
2. il branch "dev" è stato utilizzato per sincronizzare e unificare tutti gli altri branch
3. ogni feature ha avuto un proprio branch su cui hanno lavorato al più due persone contemporaneamente

La decisione di fare hosting della repository sul servizio "GitHub" ci ha permesso di utilizzare la tecnologia di "Continuous Integration" per automatizzare il workflow attraverso le "GitHub Actions". Di seguito le nostre pipeline:

1. alla creazione di un tag viene fatto in automatico il deploy sul nostro server (un tag generalmente viene creato quando si fa il merge del dev al main)

2. alla creazione di un tag viene fatta in automatico una realease (su github, allegando il JAR)
3. esecuzione dei test attraverso la creazione di Pull Request sui vari branch, i push falliscono se la build fallisce o i test non passano
4. quando c'è una libreria deprecata viene creata in automatico una Pull Request da Dependabot, se è una minor invece si automergia.

Si è sfruttato il GitHub Projects, questo è un foglio di calcolo con delle tabelle dove sono raggruppate le issue, questo schema permette di tenere traccia del product backlog e degli sprint backlog. Le nostre colonne sono:

1. To do
2. In progress
3. Waiting for approval
4. Ready to merge
5. Done

3 Requisiti

3.1 Requisiti di business

Come progetto ci si è proposti di replicare un gioco che fosse realizzabile nel tempo specificato nelle regole d'esame, ovvero 60-80 ore di lavoro individuale, per la realizzazione dell'applicazione abbiamo imposto i seguenti requisiti:

1. L'applicazione sviluppata deve avere un giusto livello di complessità, deve dimostrare l'apprendimento delle conoscenze acquisite durante il corso e rientrare nel monte ore prestabilito.

2. Il gioco deve essere sviluppato in modo che, partendo da uno stato iniziale funzionante, si riesca ogni volta a creare nuovi blocchi di funzionalità e mano a mano unirli allo stato iniziale.

3. La componente grafica deve essere il più semplice possibile ma allo stesso tempo l'architettura deve rispettare i vincoli imposti dalla programmazione ad oggetti.

L'approccio utilizzato per la definizione del modello è basato sulla filosofia **Domain Driven Design** (DDD). Inizialmente è stata svolta una fase di *Knowledge Crunching* per esplorare il dominio ed avere una struttura visiva che possa poi essere facilmente trasferita all'implementazione effettiva.

3.2 Domain Driven Design

In questa sotto sezione verranno discusse le principali metodologie del DDD.

3.2.1 Intervista agli stakeholder

Le interviste agli stakeholder rappresentano una preziosa fonte di ispirazione all'interno di un'organizzazione, perché aiutano a scoprire aree di scostamento tra la strategia aziendale e i comportamenti o i processi decisionali attivati quotidianamente dagli stakeholder di progetto.

Quali aspetti dell'applicativo si ritengono fondamentali ? L'applicazione deve simulare in maniera accurata i giochi del genere "Tower Defense", implementando tutti quei meccanismi di base che fanno si che un gioco appartenga a tale categoria. L'interfaccia può essere minimale ma deve essere intuitiva, in maniera che anche un giocatore che non conosce per niente il genere possa giocare senza problemi. L'applicazione deve essere divertente ma al contempo difficile, per questo motivo si è deciso che non ha una fine, i giocatori giocano finchè non vengono uccisi per via dell'eccessiva difficoltà.

Quali sono le regole fondamentali in un tower defense ? Il gioco di per se è molto semplice: è composto da una serie di ondate di nemici, che attraversano una mappa e il gioco termina quando un certo numero di nemici

raggiungono il traguardo; l'obiettivo del giocatore è impedire ai nemici di attraversare la mappa, creando delle torri che a loro volta spareranno ai nemici nel loro raggio.

Come si vince ? L'applicazione non prevede che ci sia un vincitore, semplicemente si procede ad oltranza fino a che l'ultima vita non esaurisce. Sarebbe carino inserire una leaderboard con i giocatori che uccidono più nemici progredendo nel gioco.

Verrà implementata una modalità multi-player ? L'idea è ottima ma al momento non si prevede l'implementazione del multiplayer, nonostante questo si prevede che l'applicazione sia costruita in maniera sufficientemente modulare tale che si possa permettere di fare aggiunte anche di questo livello, ovvero cambiamenti che vanno a impattare pesantemente su tutto il flusso del gioco.

3.2.2 Ubiquitous Language

La metodologia DDD prevede, una volta ottenuta una buona conoscenza del dominio applicativo, la necessità di definire *l'Ubiquitous Language*. L'ubiquitous Language è l'output prodotto dalla fase di Knowledge Crunching e l'artefatto generato dalla comprensione condivisa del dominio. Viene utilizzato per rappresentare un linguaggio esplicito usato per la descrizione del modello del dominio e del problem domain, permette di estrarre dei termini che verranno usati nell'implementazione del codice sorgente. L'UL contiene terminologie specifiche del business a mano a mano raffinato e migliorato, andando di conseguenza a riflettere questo cambiamento sull'implementazione del codice. Questo avviene perché il knowledge crunching è un processo continuo che non viene svolto solo all'inizio ma anche durante l'intero ciclo di vita del progetto, in modo da rendere anche il modello stesso malleabile a futuri cambiamenti.

Termini	**Significato**
Enemy	Entità che rappresenta un nemico, ha una certa quantità di vita e il suo obiettivo è arrivare vivo alla fine del tracciato della mappa per infliggere danno al player
Enemy Type	Il tipo di nemico, il suo tipo rappresenta in genere le sue qualità, chi cammina più veloce e chi ha più vita. Al momento si hanno tre tipi di nemici: Simple, Normal e Hard
Grid	Rappresenta la griglia del gioco, è una matrice NxM e ogni singola cella è chiamata Tile, la mappa può essere di vari tipi, generalmente il tipo è associato al livello di difficoltà, tutte le mappe hanno i medesimi Tile ma disposti in maniera diversa
Tile	Rappresenta una singola cella della griglia di gioco, questa cella può essere di vari tipi, tra cui: path per indicare che fa parte del sentiero che seguono i nemici, grass per indicare che ci si può costruire sopra delle torri e gli altri tipi sono più che altro logici e non interessanti per il giocatore

Termini	**Significato**
Projectile	Entità che rappresenta un proiettile, viene sparato da una torre e il suo obiettivo è colpire i nemici per infliggere danni.
Projectile Type	I proiettili sono di due grandi categorie, quelli a colpo singolo e quelli ad area. I proiettili a colpo singolo sparano delle palline, di dimensioni diverse in base al loro sottotipo e colpiscono singolarmente un nemico. I proiettili ad area invece non dispongono di un proiettile fisico e fanno danno ad area, quest'area è tracciata prendendo come centro la torre che spara e infligge danno a tutto ciò che è nei dintorni
Player	Rappresenta il giocatore, inizia ad esistere quando si preme il tasto "start game" e inizialmente possiede un certo ammontare di vita e soldi, ulteriormente ha un nome che viene assegnato dal giocatore, nel caso non fosse assegnato viene assegnato randomicamente. L'obiettivo del player è andare avanti il più possibile all'interno del gioco.
Partita	Rappresenta una sessione di gioco, definita da una specifica mappa, una partita inizia alla pressione del bottone "start game"
Mappa Custon	E' possibile caricare delle proprie mappe da file system, queste sono rappresentate da dei file json (come anche le mappe standard) e se si vuole si può giocare con le proprie mappe, per caricarle è presente nella schermata iniziale un bottone che permette quest'operazione. Le mappe standard hanno un loro livello di difficoltà, le mappe custom invece sono a discrezione del creatore

Termini	Significato
Abbandono	Quando un giocatore esce forzatamente dal gioco, perdendo
Posizionamento torre	Quest'azione per essere fatta devono sussistere una serie di condizioni: il player deve essere all'interno di una partita, deve avere i soldi per poter posizionare la torre e la posizione scelta deve essere una posizione in cui si può costruire una torre, per poter posizionare una torre si seleziona la torre desiderata e successivamente si clicca la cella nella quale lo si vuole posizionare
Incominciare una wave	Durante una partita, per far partire lo spawn dei nemici è necessario cliccare sul bottone "start wave", a segiuto di ciò spawneranno i nemici di quella specifica ondata
Spawn	Comparizione sulla mappa di un nemico

3.2.3 Knowledge crunching

Questa è stata la fase iniziale del progetto, durante la quale si è cercato di ottenere una panoramica sul dominio che consiste nei vari giochi del genere "Tower Defense". La maggior parte delle terminologie inserite nella sezione Ubiquitous Language sono emerse durante questa fase e quindi verranno omesse in questo capitolo.

Il problema principale di questa fase è la comprensione della terminologia, dei concetti e delle relazioni del dominio, fortunatamente il nostro dominio non era complicato e quindi si è riusciti abbastanza facilmente a farlo comprendere a tutti i membri del gruppo. Le sessioni di analisi delle conoscenze con l'esperto di dominio è stato molto utile per i membri che erano poco informati su questo tema e sfruttando le domande e provando realmente qualche gioco del genere "Tower Defense" è stato semplice spiegarne il funzionamento e la terminologia A seguito di ciò è stato estremamente utile disegnare diagrammi e schematizzare i comportamenti di un gioco di questo genere. I diagrammi sono un casi d'uso per comprendere l'intento del gioco e un diagramma di flusso per esplorare gli scenari, sono stati usati anche altri diagrammi più specifici, ognuno per un certo caso, questi sono illustrati nelle parti implementative.

3.2.4 User stories

Al termine della fase di knowledge crunching si sono sviluppate le user stories con il fine di poter definire dettagliatamente il caso d'uso dell'applicazione da

parte di un utente finale.

Un utente che gioca a questo gioco esige di poter fare le seguenti operazioni.

Giocare un livello

Il giocatore vuole che l'esperienza di gioco sia esattamente come quella di un qualunque "Tower Defense", quindi vuole:

- avere a disposizione le torri (almeno quelle che si può permettere con i crediti che possiede)
- vedere, una volta iniziata la wave, come i nemici attraversano il percorso per essere uccisi o arrivare a destinazione
- fare click and place delle torri per posizionarle nella griglia di gioco
- resettare il game e incominciare a giocare nuovamente dal punto di partenza
- avere la possibilità di ritornare al menu iniziale
- avere la possibilità di passare alla wave successiva, nel caso ne esista una e quella corrente sia conclusa
- avere la possibilità di chiudere il gioco

Creare una mappa

Creare una mappa custom, caricandola attraverso un file, e, se possibile, generarla direttamente dal gioco.

Menu principale

L'utente, nella schermata di gioco, vuole:

- selezionare il livello di difficoltà desiderato
- inserire il proprio nickname
- aggiungere una mappa custom
- iniziare il gioco con le impostazioni specificate
- uscire dal gioco

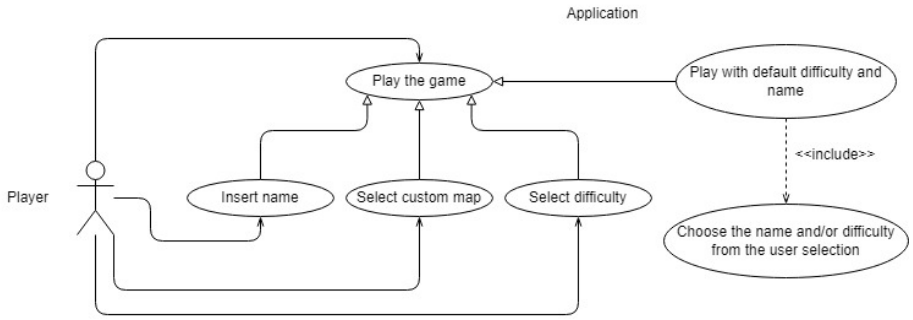

Figure 1: Diagramma dei casi d'uso

3.3 Requisiti funzionali

A partire dalle user story, sono stati formulati i seguenti requisiti funzionali:

1. Il giocatore può scegliere se giocare alle mappe fornite di default o alle proprie custom
2. Il gioco fornisce, come nei classici tower defense, tutte le varie tipologie di torri posizionabili
3. Il gioco deve permettere di caricare le proprie mappe
4. L'applicazione deve essere dotata di un menu

Di seguito una descrizione dettagliata dei punti descritti.

3.3.1 Gioco

1. Il gioco deve fornire all'utente la possibilità di giocare ad uno o più livelli
    - il giocatore può decidere se caricare una mappa o usare una difficoltà predefinita
    - se non c'è nessuna selezione, il gioco in automatico sceglie la difficoltà, il nome del giocatore e la mappa
    - ciascun livello di difficoltà è identificato dalla mappa, nel caso di mappa custom non è necessario scegliere la difficoltà
2. Tutte le mappe sono composte da una griglia 20x15 celle di gioco (che chiameremo tiles)
    - all'interno della griglia ci saranno dei tile di varie tipologie, che verranno descritti nel capitolo "dettagli implementativi"
    - i nemici camminano sui tiles che sono di tipo *"path"*

- le torri si possono posizionare nei tile di tipo *"grass"*

3. Il gioco è costituito dal susseguirsi di ondate di nemici (che chiameremo wave), fino a che il giocatore non muore
4. L'obiettivo del gioco consiste in:
   - partire con la wave
   - una volta partiti, è possibile inserire torri in qualunque momento
   - la partita termina quando le vite sono terminate
5. Il gioco non termina mai in quanto non ha una *"win condition"* ma semplicemente si va avanti finchè si hanno a disposizione le vite
6. La schermata di gioco dispone di bottoni che permettono di fare ripartire la partita dall'inizio, di andare al menu principale e di uscire dal gioco

3.3.2 Regole del gioco

Le regole del gioco sono molto semplici, in quanto le uniche azioni che si possono fare sono il posizionamento delle torri, queste hanno però dei vincoli legati al loro costo e al dove poterle posizionare.

La prima ondata di nemici spawnerà ed inizierà a muoversi solo alla pressione del relativo bottone e da li in poi le ondate saranno si susseguiranno in automatico. Il giocatore non "vince" mai, la sfida è semplicemente resistere il più a lungo possibile.

Per quanto riguarda le mappe custom, è possibile caricare da file system le mappe desiderate, il formato delle mappe è un file JSON formattato in un certo modo.

3.3.3 Menu principale

Al suo avvio, l'applicazione deve mostrare un menu con le opzioni tra cui il giocatore può scegliere le seguenti opzioni:

1. selezionare il livello di difficoltà
2. selezionare il proprio nickname
3. aggiungere una mappa customizzata
4. fare partire il gioco
5. uscire dal gioco

3.4 Requisiti non funzionali

1. L'applicazione dovrà essere *cross-platform*, cioè eseguibile su Windows versione 10 21H2 e 11 22H2, su Ubuntu 22.04 LTS e MacOS 12 Monterey (solo con processori Intel)
2. L'applicazione deve essere sufficientemente modulare in modo tale che sia possibile sia riusare i suoi elementi che aggiungerne di nuovi com semplicità. Non ci devono essere troppe dipendenze tra le classi.
3. L'applicazione non deve mai interrompersi qualora si verifichino errori di ogni sorta, deve mostrare al più messaggi di errore all'utente
4. L'applicazione deve essere fluida, cioè non freezare o laggare durante il gioco, il requisito minimo è una macchina con almeno 4gb di RAM e un processore Intel Core i5 di quinta generazione (o AMD equivalente) o superiore
5. Parti implementative quali:
   (a) creare un map editor direttamente dentro il gioco, lanciabile dal menu principale per creare le mappe custom dall'interno del gioco
   (b) gestire la musica con i suoi menu appositi e usare la propria musica da riprodurre durante il gioco

3.5 Requisiti implementativi

1. L'implementazione dell'applicazione deve essere fatta nel linguaggio "Scala", compatibile dalla 2.13.6
2. Deve essere sfruttato lo strumento di *build automation* Gradle v7.3.3
3. Lo sviluppo deve essere il più possibile conforme al paradigma della programmazione funzionale
4. L'andamento dello sviluppo deve essere fatta mediante lo spreadsheet di GitHub Projects
5. Le parti fondamentali dell'applicazione devono essere testate attraverso la libreria "ScalaTest"
6. La repository deve essere gestita attraverso il DVCS "git" e mantenuta su GitHub
7. Deve essere sfruttata la tecnologia offerta da GitHub per effettuare le procedure di *Continuous Integration*, ovvero le GitHub Actions.

4 Design

4.1 Design architetturale

L'architettura dell'applicazione sviluppata è basata sul pattern "Model-View-Controller", essendo questo uno dei maggiori pattern di riferimento per applicazioni dotate di interfaccia grafica. Questa scelta architetturale ci può consentire di poter cambiare uno dei tre componenti principali senza dover riscrivere l'intera applicazione.

Una delle caratteristiche maggiori del paradigma funzionale è senz'altro una vera e propria immutabilità dei dati, per questo motivo, si è cercato di favorire il principio di "favor immutability", questo si nota principalmente nelle classi di model, dove vengono prediletti oggetti immutabili. Nonostante la modellazione di alcuni elementi in maniera immutabile fosse banale e naturale in certe situazioni, altre volte invece risultava più difficoltosa, come ad esempio nella modellazione della salute di un nemico, questa purtroppo è destinata a cambiare con il passare del tempo, in questo e pochi altri casi simili abbiamo deciso di modellarli come elementi mutabili.

Essendo le entità del model immutabili, ci è venuto naturale modellare molti oggetti come case classe, relegando la logica complessiva in strutture di più alto livello, nel nostro caso sono i vari controller. E' presente una sorta di gerarchia tra i vari controller e nel più alto livello è presente il controller della view del main menù. Questo genere di disaccoppiamento ha reso facile aggiungere modificare nuove funzionalità all'applicazione, sarà sufficiente aggiungere il relativo controller che dovrà essere messo nel posto giusto.

Partendo dai requisiti abbiamo dapprima sviluppato un diagramma che dovrebbe rappresentare il prototipo dell'applicazione (vedi figura 2), così da notare eventuali scelte incongruenti o inconsistenti.
In figura 3 è invece illustrato un prototipo del flusso delle azioni che si possono fare all'interno dell'applicazione
Nella figura 4 abbiamo l'illustraizone di come avviene il principale processo del gioco: la selezione e il conseguente posizionamento di una torre.

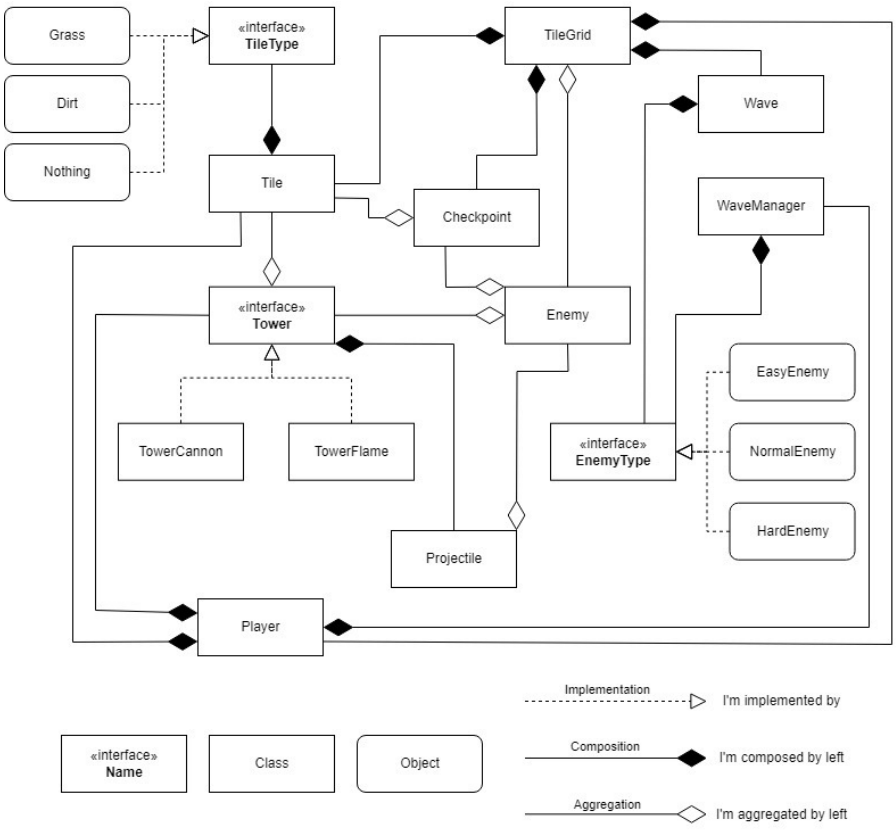

Figure 2: Diagramma del prototipo dell'applicazione

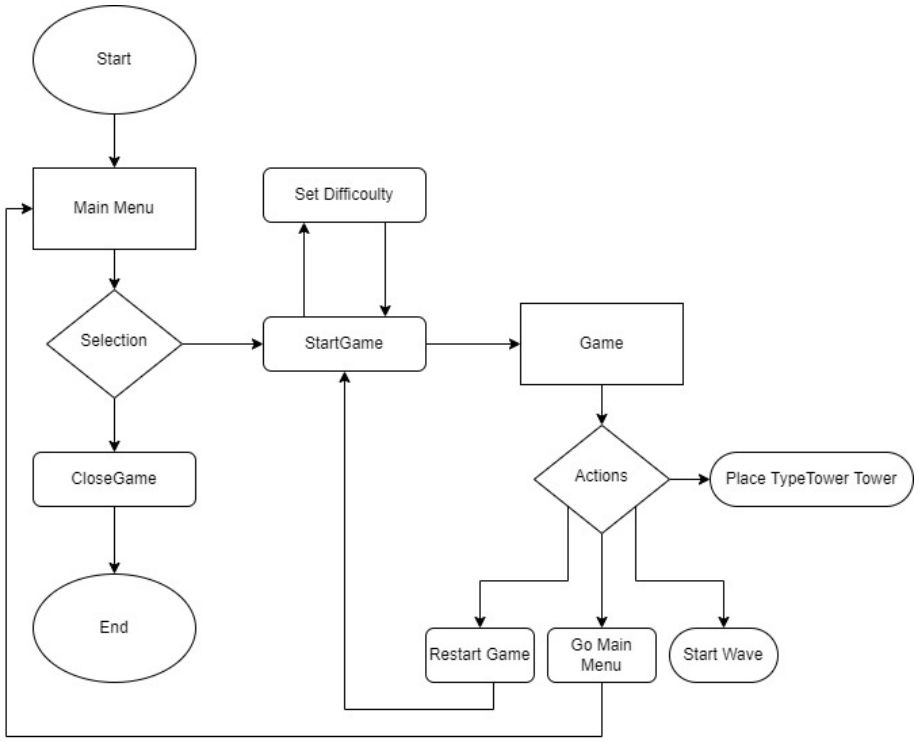

Figure 3: Diagramma di flusso dell'applicazione

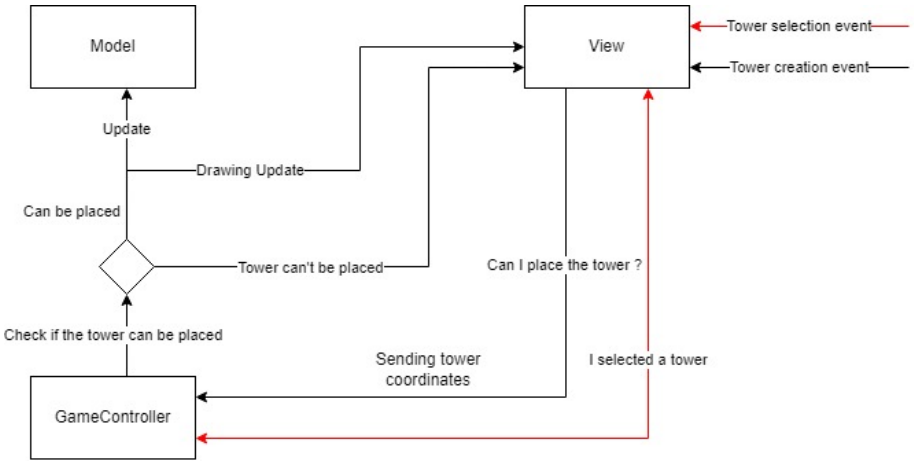

Figure 4: Processo di posizionamento di una torre

4.2 Design nel dettaglio

4.2.1 Organizzazione dei package

In figura 5 è illustrata la disposizione dei vari package all'interno dell'applicazione.

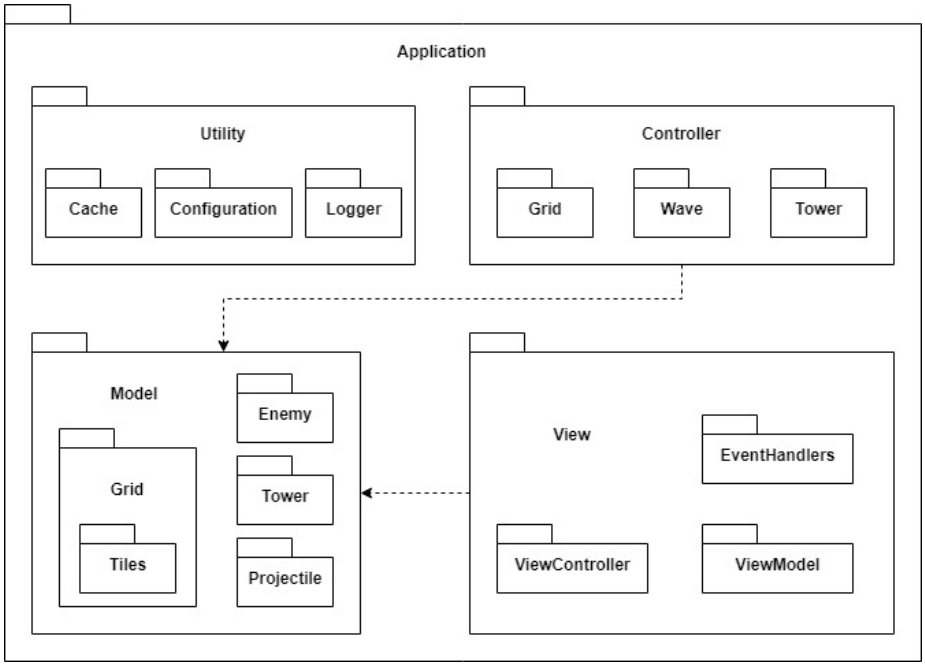

Figure 5: Package dell'applicazione

4.2.2 Componente "Model"

Questo componente contiene tutte le varie entità che sono presenti all'interno del gioco:

1. Tower
2. Enemy
3. Grid
4. Projectile
5. Player

Ognuna di queste classi contiene le definizioni di base di ogni entità che è presente all'interno dell'applicazione, ognuna modella il proprio comportamento in funzione di quello che deve fare e mantiene uno stato che sia capace di contenere le proprie informazioni più rilevanti.

4.2.3 Componente "Controller"

Questo componente contiene tra le cose più importanti:

1. GameController (è il controller di un game all'interno dell'applicazione, al suo interno vengono usate quasi tutte le entità presenti nel model attraverso i propri controllers)

2. UpdateManager: si occupa in generale del game loop, quindi effettuare gli aggiornamenti all'interno del game

Tutte le altre classi di controller sono utilizzate all'interno di questi due controller. Tutti i controller delle entità del model wrappano quelle stesse entità e offrono metodi avanzati per passare informazioni all'esterno. L'unica eccezione è DrawingManager, questa classe serve solamente per disegnare le entità all'interno della griglia e viene usata solamente da UpdateManager. Come per il model, anche qui si è cercato di seguire, quando possibile, il principio di "favor immutability"

4.2.4 Componente "View"

Questo componente è particolare, è presente un nuovamente un livello del pattern MVC, quindi abbiamo:

1. ViewModel: rappresenta tutta la grafica dell'applicazione e contiene tutte le definizioni delle interfacce grafiche

2. ViewController: rappresenta il controller del model, al suo interno contiene principalmente tutti gli event listener associati ai vari elementi del ViewModel e permette di eseguire le operazioni che l'utente fa sull'interfaccia grafica.

C'è un package, chiamato EventHandlers che serve a gestire nello specifico poi ogni singolo evento. Come il resto, queste classi sono state scritte sempre seguento il principio di "favor immutability", in particolare in questo caso ulteriormente tutte le classi sono private e l'unico modo per poter ottenere un tipo è sfruttando il trait, mentre per istanziare un'istanza è necessario sfruttare l'apply del companion object.

5 Implementazione

5.1 Ismam Abu

5.1.1 Setup del repository

Prima di iniziare a lavorare effettivamente sul progetto è stato necessario fare il setup del repository per quanto riguarda le azioni di CI/CD e di deploy. E' stato fatto anche il setup di gradle e aggiunte le librerie che sicuramente sarebbero state utilizzate. Queste operazioni sono state fatte da me in collaborazione con Hamado. Successivamente a questo ho provvisto a fare la modellazione dell'applicazione mediante l'uso dei diagrammi più noti (illustrati precedentemente nel design), grazie a ciò abbiamo creato le classi delle entità di base che sarebbero servite (mettendo inizialmente solo le entità che sarebbero state utili per il primo sprint).

5.1.2 View

Nella prima settima ho lavorato sulle classi di view per fornire, già dopo il primo sprint, un'applicazione con cui l'utente potesse interagire, ho utilizzato le librerie "ScalaFX" e "JavaFX", realizzando le seguenti classi:

1. Package ViewModel:
    - ApplicationViewModel: è un trait che viene esteso da tutti i vari model della view (ogni model ha anche un trait specifico che eredita da questo trait generico)
    - GameViewModel e MainMenuViewModel: sono le due classi di model che rappresentano le due scene dell'applicazione, la prima quella di gioco e la seconda quella del menù principale. Entrambe le classi contengono al loro interno le definizioni di tutte le varie entità grafiche del gioco, nello specifico sono bottoni, label e nel caso del GameViewModel abbiamo una griglia che è rappresentata da un canvas
2. Package ViewController:
    - ViewModelController: è un trait che viene esteso da tutti i controller della view (ogni controller ha anche un trait specifico che eredita da questo trait generico)
    - MainMenuViewController e GameViewController sono i controller dei model del package descritto in precedenza, hanno tutti in comune il metodo hookupEvents, questo si occupa di agganciare i vari action listener agli elementi del model associato.
3. Package EventHandlers:
    - EventHandlers: è un triat che viene esteso da tutti i vari event handlers della view, di base contiene le definizioni dei metodi di cui ha

poi bisogno ogni classe che estende (ogni gestore diverso possiede un trait specifico, che eredita questo trait generico)

- GameEventHandlers e MainMenuEventHandlers: gestiscono gli eventi, ogni controller della view ha una propria classe per gestire i propri eventi. Gli eventi che abbiamo sempre sono setScene e nothing, tutti gli altri sono specifici della singola scena di gioco.

L'unico modo per accedere a queste classi è attraverso l'apply del proprio companion object. L'entry point dell'intera applicazione è la classe GameLauncher, da essa viene creato il controller del main menù e successivamente lanciato. Ogni volta che si vuole cambiare schermata, viene creato nell'action listener ad esso associato il controller della scena desiderata e successivamente viene lanciata.

5.1.3 Grid

Ho lavorato alla creazione dell'entità Grid, questa rappresenta la griglia di gioco.

Grid Model

Per implementare in maniera ottimale questa entità ho realizzato un'interfaccia con un'unico metodo che restituisce la griglia di gioco, questa griglia viene creata all'interno della classe che però è accessibile solamente dal proprio companion object. Questa classe, internamente, si occupa solamente di generare la griglia di gioco e questa è rappresentata da un array bidimensionale. Per generare la mappa di gioco viene utilizzata la classe PathMaker.

Grid Controller

Questa classe wrappa al proprio interno il model della griglia e offre all'esterno tutta una serie di funzionalità per ottenere informazioni dalla griglia. Ho scritto il controller in maniera che non restituisse con nessun metodo la griglia effettiva di gioco ma solamente le cose di cui un utilizzatore abbia effettivamente bisogno, quindi si possono leggere per dei tile partendo dalle loro coordinate, si può conoscere il numero di tile presenti e combinando queste informazioni è comunque possibile ottenere tutte i tile della griglia, ma comunque, ho preferito evitare di rendere pubblica direttamente la griglia.

Tile

L'unità alla base della griglia è il *tile*, questa è un'entità che rappresenta un blocco all'interno della griglia di gioco. E' stato realizzato un trait che ne racchiude le funzionalità che possono essere usate dall'esterno e l'unico modo per istanziarlo è attraverso l'apply del companion object. Al suo interno, come campo di rilievo ha un'altro oggetto denominato *TileType*, questo è una classe che serve per poter gestire la tipologia dei vari tile e le informazioni associate ad essa (ad esempio i campi *buildable* e color sono direttamente dipendenti dalla tipologia di tile). Per creare un TileType è necessario passare dall'apply del

companion object, durante la creazione si passa il tipo desiderato (ottenuto da un'enumerazione) e viene generato l'oggetto richiesto. Tutti i tipi di tile sono nel package **Tiles**. Aver progettato in questa maniera il tile (ovvero con il tipo come oggetto complesso) rende molto più modualare la griglia, in quanto è facile aggiungere nuove tipologie di tiles, essendo anche tutte case classes è molto comodo in quanto si può utilizzare il pattern matching.

PathMaker

Questa è una classe utilizzata dal model della grid per generare la matrice che rappresenta la griglia, il trait espone una serie di metodi per generare il tipo di griglia desiderata. L'utilizzatore chiama metodo execute() che prende come parametro uno dei metodi per generare la mappa ed esegue in più delle azioni comuni a tutte le mappe (esegue i controlli sulla validazione di una certa mappa). Questa classe si occupa anche di leggere da file le mappe, in quanto le tre di base sono salvate nelle resources e quelle custom nel file system del giocatore. Per leggere da file system utilizziamo come formato il json, quindi il file deve essere prima letto e poi validato, per questi ed altri motivi ho deciso di utilizzare la libreria Gson. Il file viene letto tramite il suo path all'interno del file system attraverso un BufferedReader, i dati poi vengono prima serializzati come json e successivamente mappati e restituiti come array bidimensionale.

Player

Ho gestito l'entità che rappresenta un giocatore all'interno dell'applicazione. Questa classe è abbastanza semplice e non ha oggetti complessi al proprio interno, viene chiamata quando viene creato e valorizzata quando viene creato un nuovo game. Viene utilizzata dalle altre classi principalmente per leggere le informazioni riguardo il giocatore, come nome, vita, soldi e il contatore delle uccisioni.

Test

Ho scritto i test relativi alla Grid e alla Configuration.

5.2 Hamado Dene

Il contributo sul progetto si è incentrato sull'implementazione delle torri e dei proiettili. Le torri si dividono principalmente in due tipologie:

- Shooter tower
- Circular radius tower

Le differenza principale tra le due tipologie è che le torri di shooter identificano un nemico nel range e gli sparano usando un proiettile. Se questo proiettile entra in collisione con il nemico, quest'ultimo subisce un danno. Le torri di tipo circular radius invece non utilizzano proiettili, ma fanno un danno ad area in un

raggio circolare (Per esempio una torre che spara fiamme). Quando un nemico entra nell'area, se la torre spara un colpo ad area, i nemici prendono danno.

I proiettili hanno una logica comune, ovvero che nel momento in cui vengono sparati dalla torre, escono con una determinata velocità e se entrano in collisione con un nemico, gli infliggono un danno. Le differenze quindi stanno nella velocità in cui si muovono, dalla dimensione del proiettile e dal danno che possono influire.

5.2.1 Implementazione tower

L'implementazione della torre è suddivisa nel seguente modo:

- Implementazione del model che definisce la logica delle varie tipologie di torri.
- Un controller dalla quale tutte le torri vengono derivati.

5.2.2 Model

Nel model viene definito la logica delle due tipologie di torri attraverso le due classi ShooterTower e CircularRadiusTower. Queste due classi implementano il trait presente in TowerType. Il trait definisce le caratteristiche principali di una torre.

ShooterTower

Questa classe definisce i metodi di tutte le torri che hanno come obiettivo quello d'individuare un obiettivo e sparare. Nel costruttore prende come parametro un tipo di proiettile. Qualunque torre che spara dovrà dunque estendere questa classe.

Sono definiti i seguenti metodi:

- findDistance(enemy: Enemy): Dato un nemico, calcola la distanza tra la posizione della torre e il nemico stesso. Ritorna un valore che corrisponde alla distanza dal nemico.
- inRange(enemy: Enemy): Dato un nemico, verifica se il nemico si trova nel range che la torre può coprire.
- chooseTarget(): Prende la lista di nemici vivi e presenti sulla mappa del gioco e dato la distanza calcolata con findDistance(enemy: Enemy) e dall'esito della funzione
- inRange(enemy: Enemy), seleziona un nemico tra quelli presenti nel suo range
- attack(): Dato il nemico scelto con chooseTarget(), se quest'ultimo risulta ancora in vita, chiama il metodo fireAt(enemy: Enemy) che disporrà il proiettile per lo sparo al nemico

- fireAt() Dato un nemico, prepara il proiettile per lo sparo. Il tipo di proiettile dipenderà dal tipo di torre

CircularRadiusTower

Questa classe definisce i metodi di tutte le torri che sparano in un raggio circolare e non usano proiettili (es. Torre che spara fiamme). A differenza di shooterTower non viene scelto un target da colpire ma qualunque nemico passa nel raggio al momento dello sparo subisce un danno.

Sono definiti i seguenti metodi:

- isColliding(radius: WayPoint, enemy: Enemy): Prende in ingresso il raggio (espresso tramite WayPoint) della torre (considerando il range che può coprire con lo sparo) e uno specifico nemico. Ritorna true se il nemico è in collisione false altrimenti.

- attack(): Dato la lista di nemici presenti sul ring di gioco, infligge danno a coloro che sono in collisione al momento dello sparo. Per capire se il nemico è in collisione o meno viene chiamato il metodo isColliding(radius: WayPoint, enemy: Enemy).

Queste due classi vengono poi estese dalle varie torri, che attraverso un override andranno a ridefinire alcuni parametri come la velocità di sparo, il range che la torre può coprire ect.

Sono stati implementati tre torri possibili:

- BaseTower: Estende shooterTower. E' la torre di base, che spara semplici proiettili e infliggono un danno ridotto.

- CannonTower: Estende shooterTower. Una torre cannone che spara proiettili di grosse dimensioni che infliggono un danno elevato.

- FlameTower: Estende CircularRadiusTower. Una torre che spara fiamme.

Per istanziare queste torri è definito una factory in TowerType che attraverso il metodo apply permette i generare una determinata tipologia di torre. Questo apply prende in ingresso un Enum che permette di capire la tipologia di torre da generare.

5.2.3 Tower Controller

Per la gestione delle torri è stato creato un tower controller, dalla quale ogni torre può essere derivato. Quando si decide di creare una torre, viene istanziato un tower controller che prende come parametro:

- il tipo di torre che si vuole gestire (Ottenibile chiamando la factory delle torri)

- Un oggetto Player che specifica owner della torre

- La posizione in cui è stato buildato la torre
- un istanza di GameController per rimanere aggiornato sullo stato del gioco, in particolare di capire lo stato dei nemici sulla mappa.

Tutte le informazioni inerenti a una torre possono essere reperiti chiamando l'istanza di controller associato alla torre creata. Inoltre ogni istanza di tower controller gestisce al suo interno i propri proiettili.

5.2.4 Implementazione proiettili

I proiettili vengono istanziati dalle shooter tower quando quest'ultimi decidono di sparare a un nemico. I proiettili possono essere di diversi tipi, ma la differenza sta nella dimensione del proiettile, dalla velocità che può assumere, dal colore e dal danno che può infliggere. Hanno però una logica comune che implica l'esistenza di un'unica implementazione effettuata tramite la classe Projectile. Ogni tipologia di proiettile quindi estende la classe Projectile modificando tramite override i valori, specificando i nuovi valori che caratterizzano quello specifico proiettile.

Class Projectile

Questa classe dunque definisce la logica di un proiettile. Per essere istanziato necessita di sapere:

- la posizione del target
- la posizione di origine (Ovvero da dove è stato sparato)
- la torre che ha sparato il proiettile
- il nemico da colpire e il controllore di torri.

Implementa al suo interno le seguenti funzionalità:

- CalculateDirection(): Permette di calcolare la direzione del proiettile utilizzando la posizione del nemico e la posizione di origine del proiettile stesso
- isColliding(pos: WayPoint): Prende una posizione, e verifica se il proiettile è il collisione con il nemico. In caso affermativo ritorna true, altrimenti false
- update(delta: Double): Permette di aggiornare dinamicamente la posizione del proiettile. Tramite il metodo isColliding(pos: WayPoint), controlla se quest'ultimo è il collisione con il nemico. In caso affermativo viene inflitto un danno al nemico

5.2.5 GameController

Il gameController è il main controller che gestisce lo stato del gioco. Si occupa d'inizializzare le entità principali necessari al funzionamento del gioco, come:

- Player
- GridController
- Wave e WaveScheduler
- Liste di torri e nemici che permettono di capire quali nemici e torri sono presenti sulla mappa.

Usando il GameController è possibile:

- Conoscere la lista di nemici presenti sulla mappa
- Conoscere la lista di torri piazzati sulla mappa
- Avere la lista di torri disponibili che si possono buildare
- Sapere quale torre è stata selezionata
- Capire se tutte le istanze sono state inizializzate, quindi se il gioco è partito

I metodi principali che ho implementato sono:

- onCellClicked(x: Double, y: Double): Questa funzione viene chiamata quando si vuole buildare una torre sulla mappa. Prende come parametro le posizioni X e Y (Valorizzati in base al punto di click sulla mappa) e controlla se sono rispettate determinate condizioni:
  - Se è stata selezionata una torre e il tile cliccato è buildable e il giocatore ha abbastanza soldi, allora prova a buildare la torre. La build viene fatto facendo il clone della torre selezionata con le nuove posizioni e rivalorizzando la variabile selectedTower. Inizialmente la variabile selectedTower viene valorizzato quando il giocatore prima di cliccare sulla mappa seleziona la torre che vuole piazzare usando il menu predisposto.
    - Se è già presente una torre in quella posizione, non può essere piazzato un altra torre.
  - Se il giocatore ha selezionato una torre ma al momento del click non ha abbastanza soldi, la build fallisce
  - Se non viene selezionata nessuna torre, la build fallisce.
- buildTower(tower: Tower): Permette di valorizzare il parametro selectedTower che tiene traccia della torre selezionata nei vari stati del gioco.

5.2.6 Update Manager

La classe UpdateManager si occupa di gestire il rendering dei componenti di gioco. Implementa una funzione run() che attraverso un animationTimer chiama la funziona update(delta: Double) ogni delta. La funziona update(delta: Double) si occupa di chiamare update di tutti i componenti quali: enemy, tower, grid.

La variabile delta corrisponde a ogni quanto tempo chiamare update.

5.2.7 WayPoint

La classe WayPoint permette di gestire le posizioni 2D. Offre diverse funzionalità che permettono di eseguire operazioni molto velocemente sulle posizioni.

5.2.8 Test

Ho implementato i test relativi alle classi di GameController e Tower

5.3 Vlad Mattiussi

5.3.1 Enemy model

Nel mio primo sprint ho realizzato l'entità di base del modello caratterizzante dei nemici. Ho realizzato l'interfaccia e la relativa classe implementativa in "Enemy". Ho implementato i nemici declinandoli in tre diversi tipi, ognuno con diverse caratteristiche di punti vita, velocità e danno. Se i suoi punti vita vengono portati a zero, esso deve essere eliminato e despawnato dalla mappa. Se riesce a raggiungere la fine del percorso, esso deve infliggere un certo ammontare di danno al giocatore ed, infine, despawnarsi. Sono presenti tre oggetti caratterizzanti i tre tipi di nemici, ognuno estende il trait "enemyType". Nel momento dell'inizializzazione del nemico, uno di questi tre oggetti viene passato al costruttore del nemico, in questo modo incapsulando e definendo il suo tipo. Il nemico è implementato come una classe privata "EnemyImpl" all'interno dell'oggetto "Enemy"; in questo modo la classe rimane "protetta" e per poter creare e accedere ai metodi di Enemy, bisogna utilizzare il metodo "apply" dell'oggetto, il quale restituirà un oggetto di tipo "Enemy".

5.3.2 Enemy move

Obiettivo del secondo sprint era modellare il comportamento legato al movimento: il nemico deve muoversi seguendo un certo percorso e con una certa velocità. Questo è stato implementato nel metodo "move": questo metodo viene chiamato ad ogni update, cioè ad ogni periodo di tempo delta; ad ogni chiamata l'entità avanza sul percorso di un certo spazio, in base alla sua velocità. Partendo dalla posizione di un certo tile, attraverso un algoritmo, viene cercata e identificata la prossima posizione corretta da seguire mantenendosi correttamente sul percorso; dopo l'identificazione il nemico avanza di un certo valore. Una volta raggiunto il tile identificato, l'algoritmo ricomincia cercando la

prossima posizione; questo processo continua fino a quando il nemico raggiunge l'ultima posizione sul percorso, nella quale si despawna e vengono inflitti danni al giocatore. Una volta identificato il prossimo tile, attraverso l'utilizzo di una variabile, viene imposto una lock: fino a quando il nemico non raggiungerà quel tile, la ricerca di tile si interromperà, impedendo di identificare possibili nuove posizioni; questo viene fatto per impedire la sopraggiunta di comportamenti imprevisti dell'algoritmo, come ad esempio cambiare direzione quando ancora non si è raggiunti il tile di destinazione.

Successivamente, per mostrare correttamente il nemico ed il suo movimento in modo fluido, il rendering è stato implementato attraverso la nostra entità incaricata del disegno, "DrawingManager"; ad ogni chiamata di update, il nemico viene renderizzato.

5.3.3 Wave e wave scheduler

Nel secondo e terzo sprint ho implementato la parte creazione e gestione di ondata.Ogni ondata possiede una lista interna "enemyList" che viene inizializzata nel momento di creazione della wave: in questo momento viene chiamato il metodo "populate" che serve a creare e inserire in lista i vari nemici, in base al numero di ondata. Ho, inoltre, implementato "WaveScheduler", una classe per la gestione della programmazione di ondata, la quale deve spawnare un certo numero e un certo tipo di nemico in base al numero di ondata attuale. Attraverso la chiamata del metodo "Update_check" ad ogni delta, vengono controllate due condizioni:

- Se il nemico raggiunge la fine del percorso
- Se i punti vita del nemico raggiungono o superano lo zero

Quando un nemico soddisfa una di queste due condizioni, esso viene despawnato. Quando tutti i nemici dell'ondata attuale si despawnano, viene inizializzata la nuova wave. Ogni volta che viene inizializzata una nuova ondata, quella vecchia viene "distrutta" e sostituita da quest'ultima. Il contatore di wave si aggiorna ad ogni creazione di ondata, e in base a questo, il metodo "populate" presente in "WaveImpl" spawna un particolare numero e tipo di nemici. Si continua così fino a quando i punti vita del giocatore scendono a zero.

5.3.4 Game Controller

Durante il terzo sprint ho collaborato sulla parte di Controller con Hamado per quanto riguarda le interazioni fra torri e nemici, con il corretto comportamento di questi ultimi, descritto in precedenza.

Una volta cliccato sul bottone di start, viene fatto partire il game loop e il gioco inizia, con la possibilità di piazzare torri e danneggiare i nemici in arrivo. Per una migliore gestione del gioco, torri e nemici presenti sulla mappa vengono aggiornati costantemente in due apposite liste.

5.3.5 View

Durante l'ultimo sprint ho lavorato alla realizzazione del migliore rendering grafico dei nemici.

5.3.6 Test

Ho scritto i test per le classi di Enemy e Wave.

6 OPS

Il termine **Operations** indica tutte quelle soluzioni che permettono di semplificare e automatizzare alcuni workflow relativi alla gestione del progetto. L'utilizzo del **CI/CD** rende la fase di *deploy* il più automatizzata possibile, permettendo di superare problematiche che si possono verificare negli scenari in cui lo sviluppo software avviene in parallelo. In questo progetto come ambiente di **Continuos integration** è stato utilizzato **Github Actions**, questo consente di automatizzare alcuni aspetti fondamentali del progetto come:

- compilazione e generazione dell'eseguibile
- release di una versione stabile del progetto
- esecuzione dei test
- pubblicazione degli artefatti su un repository o su un server dedicato

6.1 Build automation

La build del progetto viene effettuata tramite un workflow. Tramite una matrice di build viene riutilizzato il codice per l'esecuzione su diversi sistemi operativi. I sistemi operativi utilizzati sono Windows, MacOs, Ubuntu, ognuno all'ultima versione disponibile. I trigger che fanno scattare l'esecuzione del workflow sono i seguenti:

- push: quando viene effettuato un push sul branch main o sul branch dev
- pull request: quando viene aperto un PR verso il Branch main o dev

Nel caso delle **Pull Request**, è stato introdotta una *branch protection rules* che permette il merging solo quando il worflow esegue correttamente la build e i test.

6.2 Dependabot

Per la protezione del proprio software è sempre opportuno mantenere le dipendenze aggiornate. Tuttavia verificarne manualmente lo stato può risultare molto arduo, sopratutto se il progetto è di grosse dimensioni con un numero di dipendenze molto elevato. Dependabot risolve questa problematica controllando e aggiornando automaticamente le dipendenze del progetto. Una delle funzionalità che offre dependabot è quello di effettuare anche aggiornamenti di sicurezza, aggiornando i pacchetti che presentano vulnerabilità note.

La configurazione di dependabot scelta prevede un controllo a cadenza giornaliera delle dipendenze del branch dev. Quando viene generata una PR da dependabot, se tutti i test passano e l'aggiornamento è una minor, la PR viene approvata e mergiata automaticamente da un workflow.

6.3 Automatic delivery e deployment

Per la gestione delle versioni del progetto è stata adottata la politica di *semantic versionig*. Per gestire il versionamento è stato utilizzato il plugin Glovo/gradle-versioning-plugin, per la creazione delle nuove versioni si utilizzano i tag di GIT.

Quando viene creato un tag tramite git, viene eseguito un workflow per la pubblicazione della release. Tramite il tag creato, il workflow crea la release con riferimento al tag stesso e allegando l'eseguibile. Oltre alla release viene anche eseguito il deploy e inviato l'eseguibile su un server linux.

Dato che in questo progetto la creazione del tag corrisponde ad un rilascio, è stato sviluppato anche un workflow che al rilascio, genera una PR per preparare la repository alla versione major successiva. Tuttavia, la PR viene effettuata solo se la release in questione è una major. In caso di release minor, non viene effettuato l'incremento alla major.

6.3.1 Pubblicazione dell'eseguibile sul server linux

La pubblicazione dell'eseguibile consiste sostanzialmente nel buildare il progetto per la matrice di SO impostata e inviare dunque ogni eseguibile sul server. Per ogni versione viene creato un cartella apposita dove verranno caricati gli eseguibili.

6.4 Licensing

La licenza adottata è **MIT License**. Permette a chiunque ottenga un copia del software e dei file di documentazione associati, di trattare quest'ultimo senza restrizioni, senza limitazione nell'utilizzo, copiare, modificare, unire, pubblicare, distribuire, concedere o vendere copie del software. Tuttavia specifica che il software viene fornito cosi com'è, senza alcun tipo di garanzia, espressa o implicita. In nessun caso quindi gli autori o titolari saranno responsabili per alcun reclamo, danno o responsabilità.

6.5 Quality Assurance

6.5.1 Testing

Parti del codice sono state testate utilizzando le librerie *scatatest* e *Junit* seguendo la metodolia TDD (Test Driven Domain). Il progetto è stato sviluppato principalmente su windows, ma i test (localmente) tramite un container docker potevano essere lanciati anche su unix partendo dall'immagine docker **gradle:7.3-jdk17-alpine** . Inoltre ad ogni push su github vengono eseguiti i test e la build usando windows, linux e macOs. Lo sviluppo è stato fatto usando java 17. Non è garantito dunque il supporto per versioni precedenti.

6.5.2 Covarage

La copertura dei test è stata verificata tramite il plugin Gradle Scoverage. La percentuale di coverage ammonta al 38.47% influenzato dal fatto che per le

funzionalità di interfaccia non è stato eseguito nessun test.

Figure 6: Statement coverage

All packages	38.47%
Controller	18.77%
Controller.Tower	62.26%
Controller.Wave	72.34%
Model	19.23%
Model.Enemy	55.43%
Model.Grid	66.67%
Model.Grid.Tiles	47.22%
Model.Projectile	76.32%
Model.Tower	82.11%
Utility	40.62%
Utility.Cache	0.00%
Utility.Configuration	70.63%
Utility.Logger	100.00%
View.EventHandlers	0.00%
View.ViewController	0.00%
View.ViewModel	0.00%

Figure 7: Package coverage

6.5.3 Code Style

Durante la compilazione del codice Scala, per mantenere un codice il più pulito possibile è stato inserito una jvm option per il i task di tipo *ScalaCompile* in modo da rilevare tutti i warning. Questo implica quindi che tutti i warning vengono interpretati come errori di compilazione. La JVM options usata è **Werror**.

Per la formattazione del codice non è stato utilizzato nessun plugin esterno, ma è stato utilizzato la formattazione già predisposta da *IntelliJ*. Per automatiz-

zazione della formattazione sono stati abilitati i flag come da guida in reformat on save.

7 Retrospective

7.1 Project setup sprint

Activity	Notes
Branch protection rules	
Project setup	Gradle project setup github action workflow for build and test
Setup dependabot auto-merge for minor patch	
Setup action for release on tag	
Dockerfile for run test locally	
Added deploy github action file	Github action to deploy the jar on aws server
Create Dev branch	

7.2 Sprint v1.0.0

Issue	What has been done?	Type
Create tower	Tower Entity Creation and sub entities.	Feature
Create Enemy	Create enemy entity with their all sub entities	Feature
Create Map Model	Create model of grid and grids sub entity (Tile, type o Tiles, Grid)	Feature
Fix and refactor project structure #28	Fix View Event Handlers. Creation of a controller for each entity and a generic controller that includes all sub-controllersGeneric Refactoring	Feature
Enemy movements	Implement methods to permit to enemy move around the map, starting from the spawn tile to the end tile.	Feature

7.3 Sprint v2.0.0

Issue	What has been done?	Type
Wave scheduler	A wave scheduler to manage enemy spawns	Feature
Create the GameController	A class that handle a game, it must contain a grid, a player and a wave	Feature

Issue	What has been done?	Type
Index 613 out of bounds for length 15	Index 613 out of bounds for length 15 when function isTileBuildable(x: Int, y: Int): Boolean = gridController.isTileBuildable(x, y) is call.	Bug
Create a Wave (of enemies)	This class handle the generation of enemies and must contain the wave information	enhancement
Create the Player	This class must contain the information about the player	Feature

7.4 Sprint v3.0.0

Issue	What has been done?	Type
Wave and enemies update	Now enemies despawn correctly (by death or touching end tile). When all enemies of wave despawn, a new wave starts.	
Projectile entity		
Implement tower shoot	implement logic to identifier enemy in range and shoot enemy with projectile.	
Custom maps	Permit user to upload map grid to the game and play with that.	
Added first part of path chooser and fixed path selection	Implemented the file chooser and relative controls. For the moment the map is selected by game difficulty, but i think it will changed in another selection named “Map Selection” with map names, we use the difficulty is only for enemy spawn.	

7.5 Sprint v4.0.0

Issue	What has been done?	Labels
Restart game	Implement the restart game button in the GameView	Feature

Issue	What has been done?	Labels
Handle player health and money when enemy is passed	Added data binding from Object Player to label in the game view in class UpdateManager	Bug
Grid controller mustn't expose the grid	At the moment the grid controller expose the grid from the grid model. Fix all usages of this method with the appropriate one.	enhancement
Fix the DrawingManager	In this moment the object DrawingManager need an istance of the GraphicContext (from canvas). I'm removing this dependency because it is an unmaintainable solution.	enhancement, Upgrade
Fix tower and projectile parameter		enhancement, Upgrade
Fix file path reader		enhancement, Upgrade
Tower > Adjust all default parameters		enhancement
Player health money label	I added the data binding from the Object Player to the label in the game view in the class UpdateManager	Feature
fixed the grid model dependencies	Fixed the grid model dependencies. Removed from grid controller the metod that retrieve the grid from the grid model. Added two methods, one to get the number of rows of the grid and another one for the columns	enhancement

8 Conclusioni

Portare a termine il progetto non è stato semplice, nonostante fosse molto contenuto è stato difficile ottenere un prodotto finale che fosse vicino agli standard prefissati, in realtà siamo consapevoli di alcune mancanze, come la conoscenza approfondita e perfetta del linguaggio Scala. Nonostante fosse un'applicazione abbastanza ridotta sono servite svariate accortezze per poter lavorare in gruppo senza avere conflitti. Inoltre non è stato facile individuare quali sono i compiti da svolgere e con quali priorità. In fine, considerando che due membri del gruppo fossero studenti lavoratori, è stato difficile organizzarsi per gestire le riunioni che Scrum richiede, questo è stato abbastanza problematico anche durante l'implementazione effetiva dell'applicazione.

Nonostante tutto, alla fine del quarto sprint ci siamo resi conto che tutta la parte di progettazione e l'utilizzo di uno sviluppo basato su metodologie "agile" ci ha permesso di raggiungere i nostri obiettivi. La progettazione iniziale purtroppo era abbastanza generica e mancava di alcuni dettagli, però è stata molto utile e ci ha guidato durante la fase di sviluppo evitandoci di commettere errori concettuali. In fine è tornato molto utile fare i *daily scrum* e gli *weekly scrum* durante gli sprint settimanali, dove si decidevano le priorità. Le dimensioni del progetto fortunatamente erano state pensate bene per essere svolte da tre persone e ogni membro ha lavorato in maniera efficiente alla propria parte.

Fortunatamente non ci sono stati grossi intoppi con le librerie esterne, le difficoltà riscontrate sono state più nell'aver deciso di utilizzare come tecnologia di *build automation* tool Gradle al posto di SBT, ma questa scelta è stata dettata dal fatto che fosse più noto e versatile rispetto ad SBT. Riteniamo di aver avuto un'esperienza che giudichiamo più che positiva con il linguaggio Scala, esso permette una transizione ottima da un approccio agli oggetti ad uno funzionale.

Complessivamente ci possiamo ritenere moderatamente soddisfatti dei risultati raggiunti. Si possono aggiungere in maniera molto semplice nuove funzionalità ed apportare ulteriori migliorie al codice. Si potrebbero aggiungere nuovi tile, enemy e tower o aggiungere nuove interfacce per gestire nuovi menù in maniera modulare grazie al fatto che il codice sia stato scritto in maniera tale da permettere questa operazione.