Towers and Stuff

Gabos Norbert, Filippo Di Pietro, Leonardo Randacio

22 giugno 2022

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il software Towers and Stuff (TAS) è un videogioco di tipo tower defence (TD). Il gioco consiste in un campo in cui è presente un percorso attraversato da dei nemici, i quali aumentano di numero nel tempo, passando da un'ondata ad un altra. Se i nemici raggiungono la fine del percorso, il giocatore perde parte della sua vita terminando la partita quando essa arriva a zero. Il giocatore per difendersi può piazzare delle torri sul campo da gioco, le quali alcune possono eseguire degli upgrade in danno dopo un po 'che sono state piazzate, per uccidere i nemici prima che raggiungano la fine del percorso. L'obiettivo è sopravvivere per il maggior numero di ondate possibile.

Requisiti funzionali

- Il giocatore potrà scegliere di giocare una nuova partita con i livelli di default, oppure crearne di suoi personalizzati, (modalità sandbox)
- I nemici generati seguiranno il percorso specifico del livello, con l'obiettivo di arrivare in fondo danneggiando il giocatore
- La difficoltà del gioco aumenterà man mano che si avanza con i round, generando sempre nemici più e in un intervallo minore
- Il giocatore per difendersi sceglierà di piazzare delle torrette che uccideranno i nemici, che non dovranno sovrapporsi nel percorso dei nemici, e sovrapporsi dalle altre torri già piazzate
- I nemici uccisi aumenteranno la quantità di denaro del giocatore permettendogli di acquisire nuove difese, e incrementa il punteggio
- Il gioco termina quando la vita del giocatore scende sotto lo zero
- Le torri saranno aggiornabili, in danno

Requisiti non funzionali

- Il gioco dev'essere fluido anche con molti elementi sullo schermo

- L’interfaccia utente deve essere intuitiva anche per un utente che non ha mai visto il gioco

1.2 Analisi e modello del dominio

Ci sarà un menù principale da cui si possono accede alla modalità sandbox in cui si potranno creare dei livelli nuovi o selezionare un livello per iniziare una partita.

Il mondo di gioco sarà composto da una mappa sulla quale verranno posizionate le varie entità che potranno interagire tra di loro. Queste ultime sono suddivise in **nemici** e **torri**.

I nemici sono entità semplici con le quali l’utente non potrà interagire e verranno generati automaticamente da una funzione matematica. Il loro obiettivo è quello di completare il percorso danneggiando il giocatore. Ogni nemico ha delle caratteristiche uniche, come: vita, velocità, danno, soldi, ecc.

Le torri invece sono entità più articolate, con differenti tipi di attacchi, che saranno generate dall’utente. Ogni torre ha le proprie caratteristiche che la rendono unica. Il loro obiettivo è quello di ingaggiare i nemici e sparargli finché non muoiono.

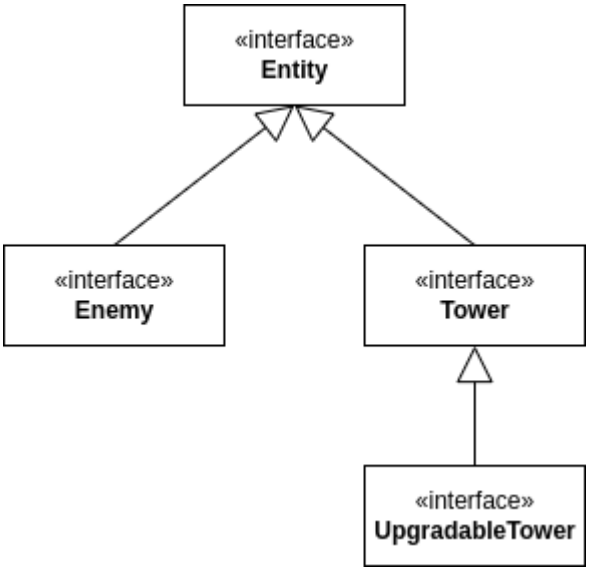


fig. 1.1 Gerarchia delle entità presenti nel progetto.

Capitolo 2

Design

2.1. Architettura

L’architettura del gioco segue il pattern architetturale MVC. Abbiamo scelto questo per separare in modo netto la logica dalle entità. Il model gestisce la logica di ogni sua entità da cui è composto, mentre la view si occupa dell'interazione con l’utente e di gestire le varie parti grafiche , mentre il controller si occupa di manipolare il model, e di aggiornare la view.

La classe che fa partire il gioco è chiamata **TowersAndStuff**, non fa altro che richiamare il metodo main della classe **MainController**. Quest’ultima si occupa di costruire la view, il model e il controller, sia del menù che del gioco.

Ad ogni interfaccia grafica del gioco, chiamate scene, corrisponde un **SceneController** permettendo al programma di poter intercambiare tra di loro le varie scene, rendendo il gioco estremamente customizzabile.

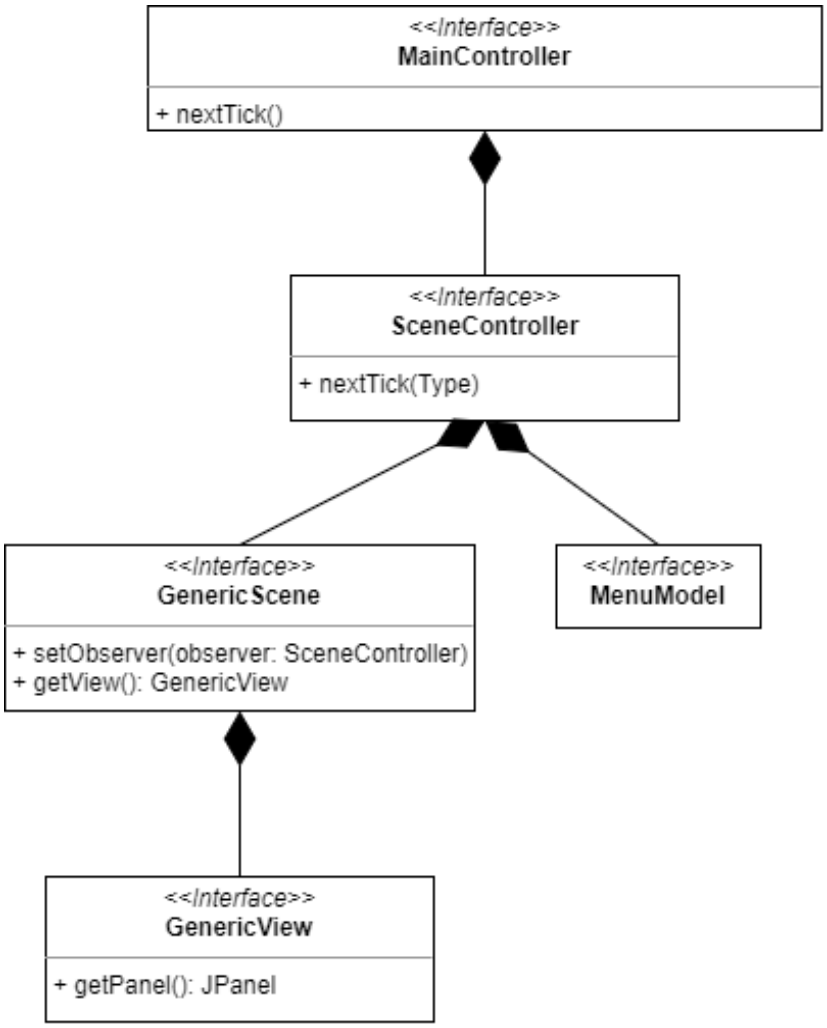


Figura 2.1.1 Rappresentazione UML della gestione del menu

Per quanto riguarda il **GameController**, esso si occupa di gestire i controller delle varie Entity, come la **EnemyLogic** e la **TowerController** Inoltre gestisce anche la generazione delle varie parti grafiche tramite la **GameView**.
In particolare:

- **EnemyLogic** gestisce tutti i nemici del gioco, permettendo di generare nemici nuovi all'inizio del round, di uccidere i nemici e di abilitarli
- **TowerController**, gestisce tutte le torri che vengono costruite dell'utente, di togliere i soldi quando l'utente le crea, inoltre si occupa poi ti disegnarle tutte

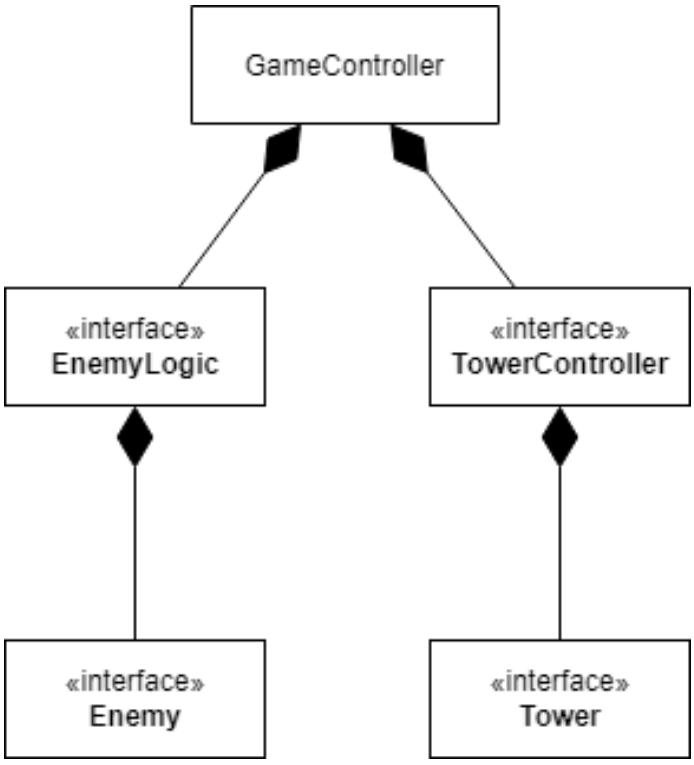


Figura 2.1.2 Rappresentazione UML dell'interazione tra i vari controller

Il model è completamente indipendente dal Controller e dalla View.

Per quanto riguarda la view, ogni scena ha la sua componente grafica **GenericScene**, che si occupa di modellare la GUI.

Il **MainController** si occupa di generare una **MainView**, che crea la finestra di gioco, dopo di che genera le varie scene grafiche in base alle scelte dell'utente.

Le scene a loro volta creano delle **GenericView** con le quali possono modificare la GUI.

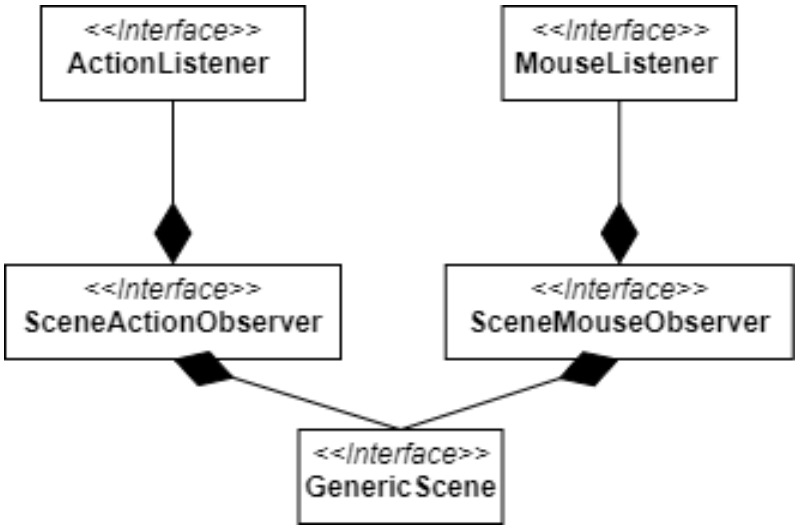


Figura 2.1.3 che mostra l'interazione tra il controller e la view.

Il model è completamente indipendente dal Controller e dalla View.

Per quanto riguarda la view, ogni scena ha la sua componente grafica **GenericScene**, che si occupa di modellare la GUI.

Il **MainController** si occupa di generare una **MainView**, che crea la finestra di gioco, dopo di che genera le varie scene grafiche in base alle scelte dell'utente.

Le scene a loro volta creano delle **GenericView** con le quali possono modificare la GUI.

L'unica interazione tra il model e la view è quando alla **GameView** vengono passate le Entity, questa utilizza la loro posizione per disegnarle sullo schermo.

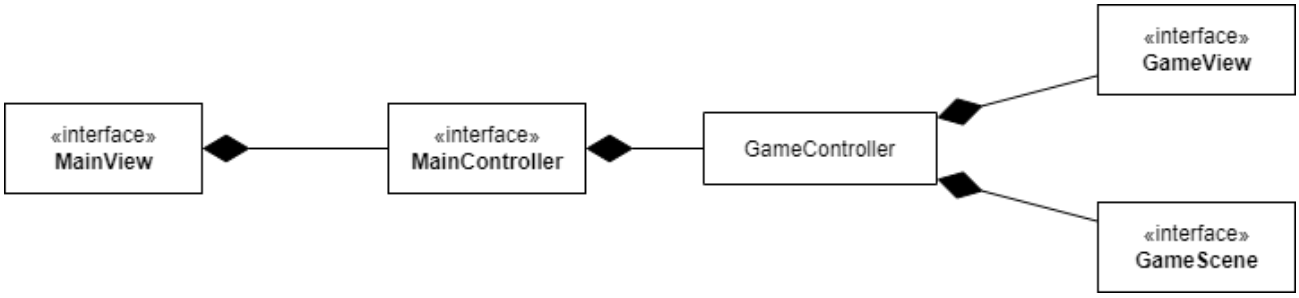


Figura 2.1.4 UML che mostra l'interazione tra i controller e le varie GUI

2.2. Design dettagliato

2.2.1. Gabos Norbert

- Generazione degli Enemy

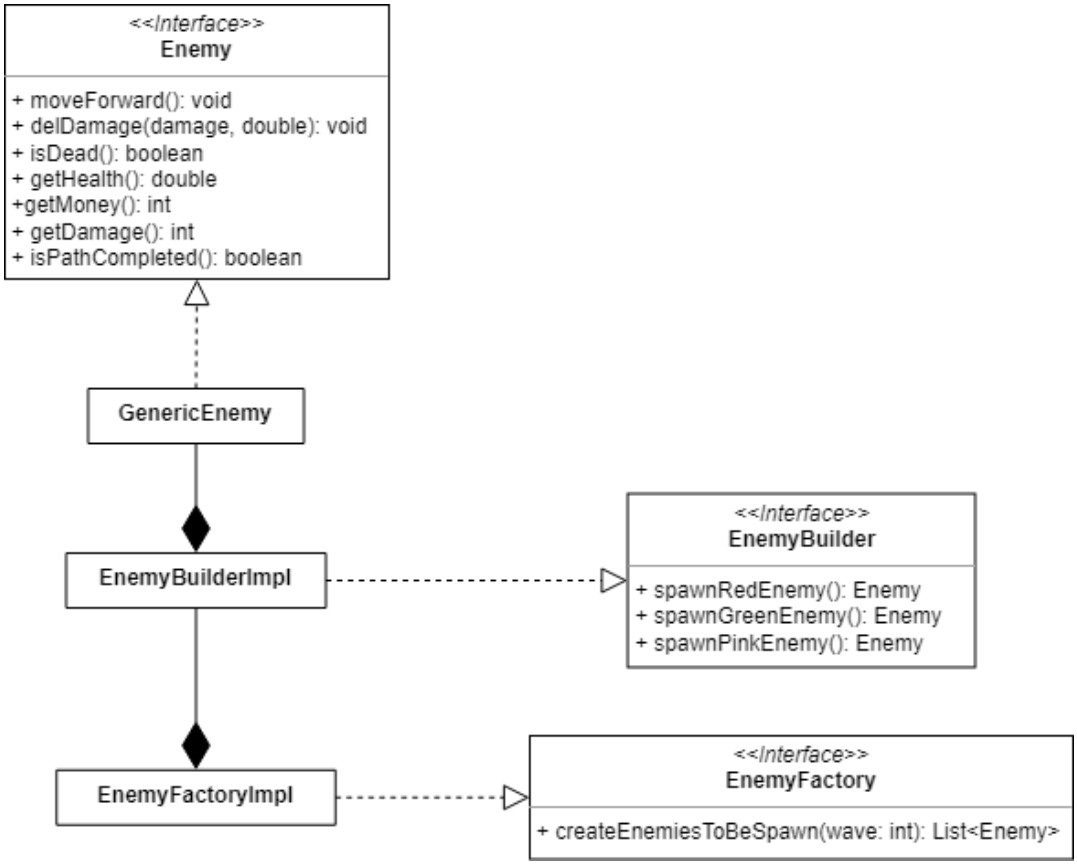


fig. 2.2.1.1 Rappresentazione UML della generazione degli Enemy

Problema: Bisogna creare più tipi di Enemy. Il mio obiettivo era quello di crearli utilizzando più codice possibile.

Soluzione: Inizialmente ho pensato di realizzare una classe astratta **AbstractEnemy**, dalla quale ereditano tutti i tipi di Enemy, l'unica cosa che cambia sono gli attributi delle varie classi. Questa soluzione l'ho subito scartata, perché risultava poco estendibile. Alla fine ho deciso di utilizzare il pattern builder, creando una classe **GenericEnemy**, la quale veniva istanziata dalla classe **EnemyBuilderImpl**.

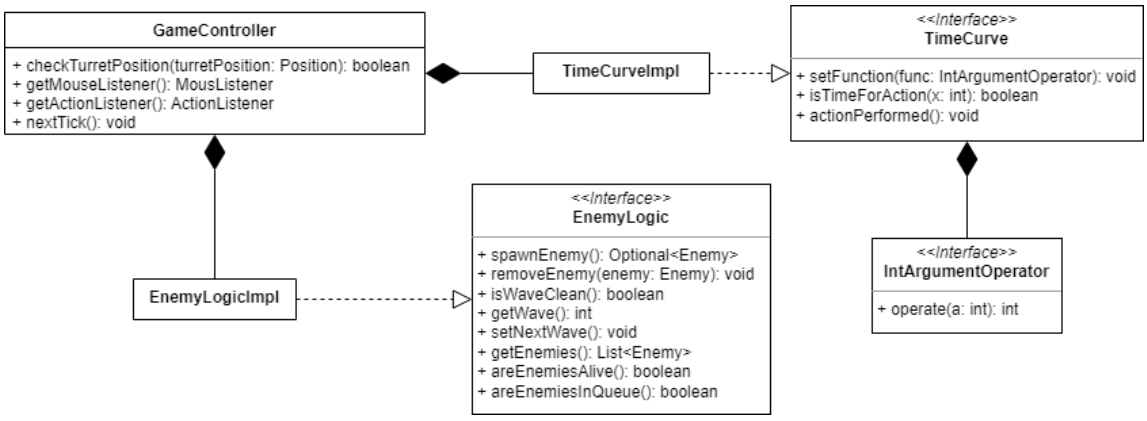


fig. 2.2.1.2 Rappresentazione UML della gestione degli Enemy

Problema: Gli Enemy hanno bisogno di essere generati in modo dinamico, ovvero aumentando di numero e diminuendo il tempo di generazione tra uno e l'altro man mano che si avanza con i round.

Soluzione: Per risolvere la prima parte ho deciso di creare una **EnemyLogic**, la quale utilizzando il pattern Simple Factory, che genera i nemici di cui ha bisogno round per round. Per la seconda parte ho creato la classe **TimeCurve**, che accetta in ingresso una funzione matematica (in questo caso una funzione esponenziale con esponente compreso tra 0 e 1) e che aspetta un tempo determinato da quest'ultima prima di generare il prossimo Enemy.

- Gestione degli Enemy

Problema: Dopo la generazione degli Enemy, le classi del controller e della view hanno bisogno di poter accedere alla loro lista.

Soluzione: Inizialmente ho pensato di utilizzare il pattern Proxy, per evitare che altre classi al di fuori della **EnemyLogic** possano modificare gli Enemy. Questa soluzione ho deciso di scartarla, perché sporca molto il codice obbligando la classe **EnemyLogic** di tenere in cache una copia della lista di un eventuale "ProxyEnemy". Alla fine ho optato di passare la lista degli Enemy così com'è.

- Gestione degli scambi di scena

fig. 2.2.1.3 Rappresentazione UML della gestione delle scene

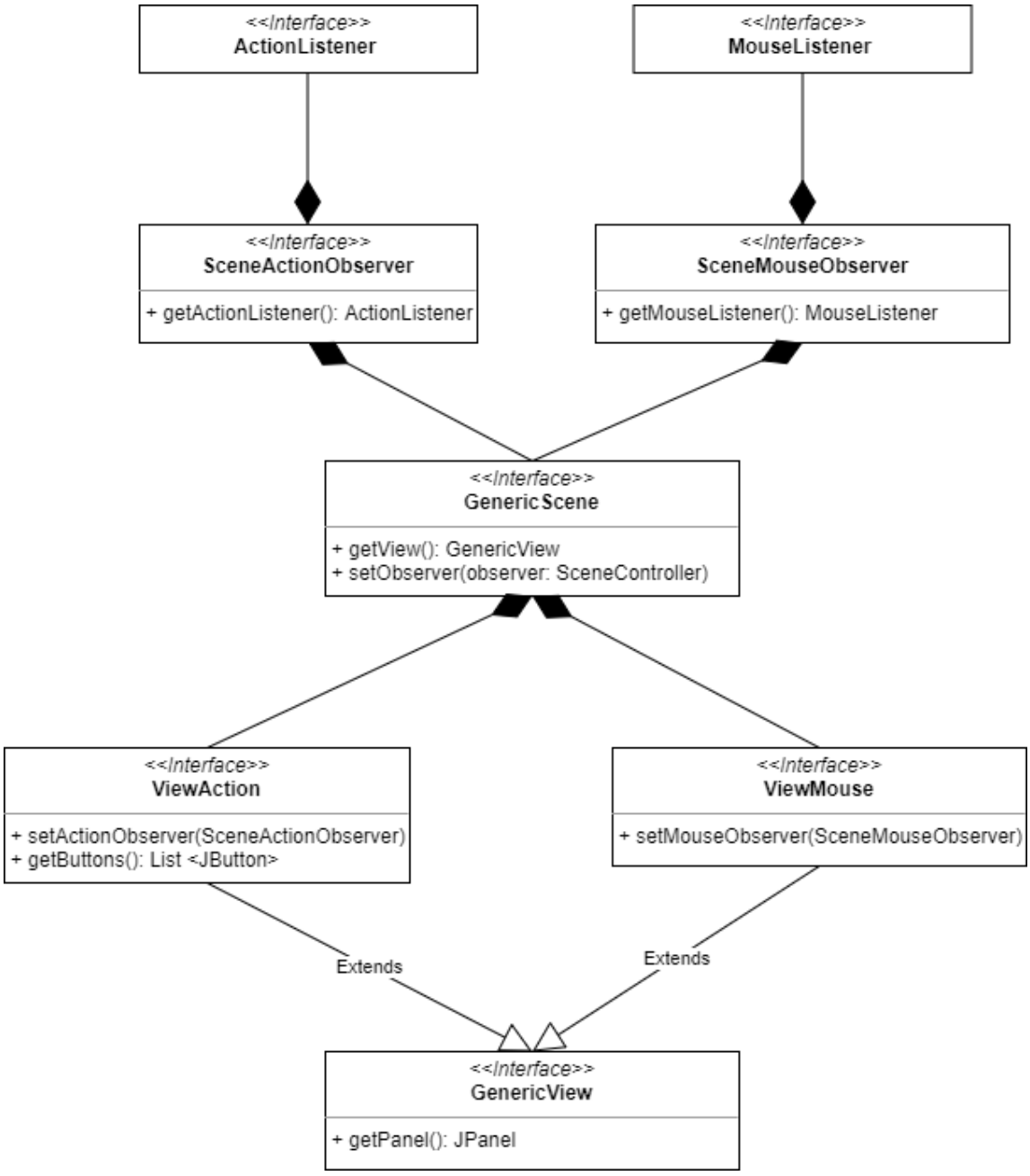
Problema: Il gioco essendo composto da più scene (per esempio quella del menù, oppure del gioco), ha bisogno di poterle scambiare tra di loro a piacimento.

Soluzione: Per mantenere il codice più pulito ho deciso di adottare il pattern Strategy, creando un'interfaccia **SceneController** che dichiare il metodo nextTick. Così facendo, nel **MainController** c'è un main loop, che cicla il gioco all'infinito e per ogni frame chiama il nextTick della scena attualmente in uso. Questo metodo non fa altro che eseguire le istruzioni della scena per ogni frame.

2.2.2. Leonardo Randacio

Problema: la view deve comunicare con il controller quando l'utente fa scaturire degli eventi (cliccare lo schermo o un pulsante)

Soluzione: Ho applicato il pattern observer, rendendo il SceneController un observer la GenericView un observable. Per semplificare la possibile futura modifica del codice anche GenericScene ha un metodo setObserver, in modo che SceneController comunichi solo con la sua GenericScene, che all'interno può essere organizzata come preferisce. Ho diviso i diversi tipi di observer in SceneActionObserver (se si aspettano un evento scaturito da un pulsante) e SceneMouseObserver (se si aspettano un evento scaturito dal cliccare lo schermo)



UML dell'applicazione del pattern OBSERVER.

Problema: tutti i listener hanno bisogno di comunicare all'observer (SceneController) quando un evento è avvenuto

Soluzione: ho create una classe GenericListener che può essere estesa da tutti i listener

per evitare la molteplice riscrittura di codice.

Problema: i listener, come parte del controller, devono essere indipendenti dalla view.

Soluzione: ho creato un metodo in ViewAction, getButtons(), che restituisce la lista dei pulsanti della view, in modo che i listener possano ricevere la lista e restituire il comando giusto a seconda di quale pulsante della lista è stato cliccato.

2.2.3. Filippo Di Pietro

Nello sviluppo di tas mi sono occupato delle Torri e di loro attacchi, di come controllare tali torri una volta piazzate, e controllare che una volta scelta una posizione in cui si voglia costruire una torre, sia distante abbastanza dalle torri già costruite.

Modellazione dell'entità torre

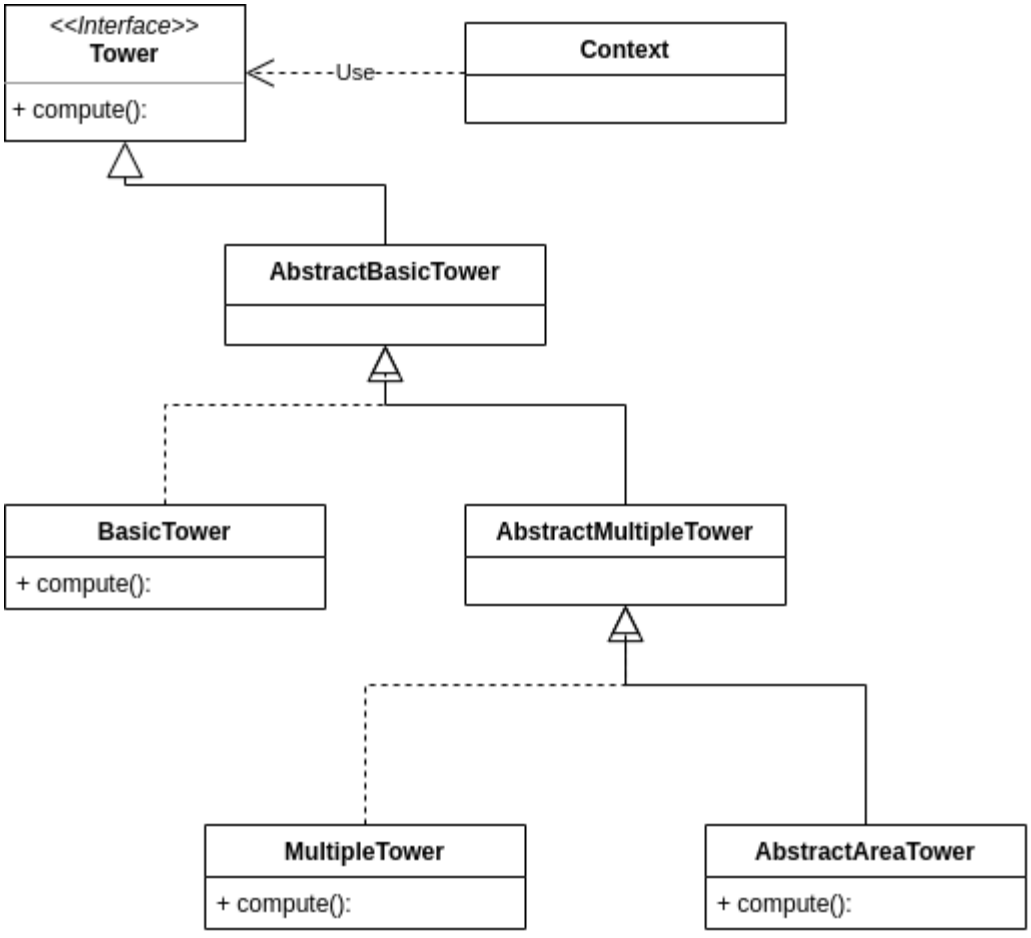


Fig. 2.2.3.1

PROBLEMA: Creare una gerarchia di classi/ interfacce in maniera tale da poter creare

varie torri di natura e comportamento diverse, cambiando anche le modalità di attacco e che siano facilmente estendibili, e facili da comprendere.

SOLUZIONE: Per la soluzione di questo problema ho optato per creare una interfaccia Tower che modella una qualsiasi torre presente sul gioco. In secondo luogo ero in dubbio se usare il pattern strutturale proxy, oppure l'uso del pattern Strategy, ho optato per quest' ultimo poiché non violava il Single Responsibility Principale e l' Open Closed Principle, e ho introdotto una ulteriore modifica a tale pattern: ho raggruppato caratteristiche comuni a molte torri, attraverso classi astratte, in modo tale che se due torri sono molto simili non devo riscrivere parti di codice similari, rispettando il DRY. Ho optato per le classi astratte invece di interfacce, poiché mi servivano funzioni in comune, come le funzioni protected (che non si possono fare con le interfacce), ma non visibile al client, e in alcuni casi servivano variabili in più (sempre non visibili al client) di conseguenza ho scelto le classi astratte.

Attacco ad area

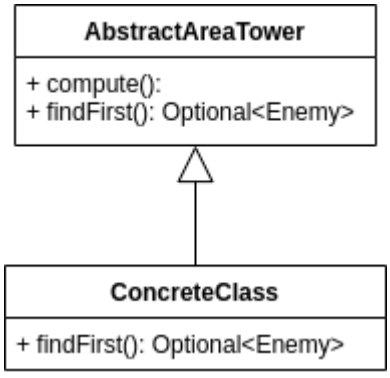


Fig 2.2.3.2

PROBLEMA: Tutte le torri ad attacco ad area hanno tutte lo stesso funzionamento, tranne per un piccolo particolare, quale nemico scegliere per primo, poi di conseguenza verranno attaccati tutti i nemici li attorno.

SOLUZIONE: La classe che modella una torre con attacco ad area, ha un funzionamento ben definito, come sopra citato, di conseguenza ero indeciso su che pattern usare, tra Strategy o Template Method, solo che in primis dovevo estendere una classe astratta (motivo citato sopra), invece di implementare una interfaccia, poi non mi piaceva il fatto di creare una classe concreta per fornire un metodo solo (findFirst), allora ho deciso di optare per un Template Method in cui viene definita la funzione mancante solo quando viene costruita tale torre.

Torri upgradabili

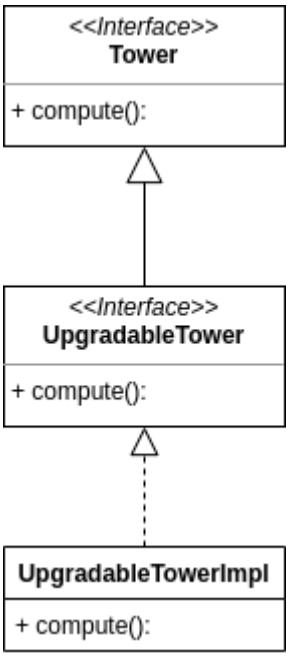


Fig. 2.2.3.3

PROBLEMA: Costruire torri aggiornabili in attacco, dopo un po che tale torre è stata costruita

SOLUZIONE: Per risolvere tale problematica avevo inizialmente pensato di costruire l'ereditarietà multipla, però poi dovevo creare molte interfacce, complicando di molto la soluzione, ad un problema a parer mio banale, per cui ho optato per il pattern

Decorator, che con la delegazione permetteva di risolvere il problema.

Costruzione

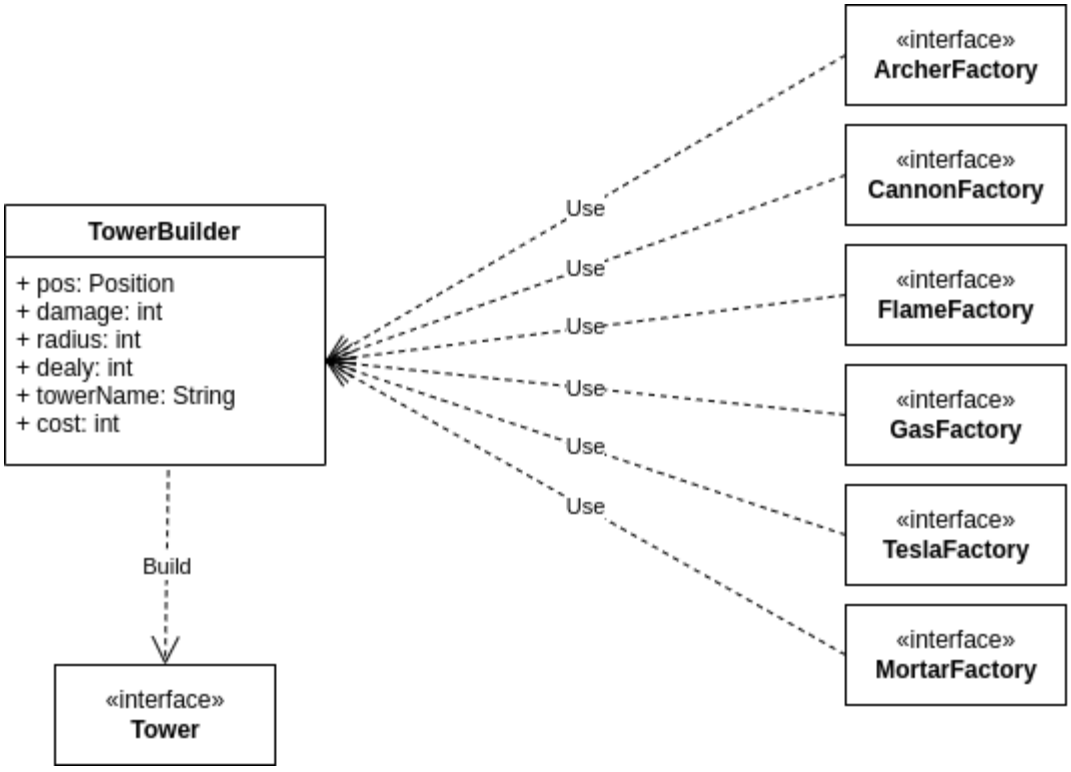


Fig. 2.2.3.4

PROBLEMA: Costruire le torri, siccome le torri possono avere attacchi diversi, ci sono in gioco numerosi parametri in più, rispetto magari ad una torre basilare, di conseguenza mi servirebbe un modo facile per costruire tali torri in maniera personalizzata. Ma non basta perché per far girare il gioco serve un set di torri minimo per farlo funzionare.

SOLUZIONE: Per la soluzione alla prima parte del problema ho deciso di usare il pattern Builder, siccome ci sono molteplici parametri in certi casi mentre in altri ce ne sono pochi in gioco. Il Builder controlla anche che i valori inseriti siano validi e che non ci sia una definizione di un parametro quando non serve. Mentre per le torri basilari che ha il gioco di default ho deciso di usare il pattern Factory, come interfacce e di default vengono restituite un'implementazione dell'interfaccia Tower, che poi ho sfruttato il Builder per questioni di chiarezza, semplicità e poi lo avevo sviluppato prima. Inoltre ho deciso che le informazioni basilari possono essere prese da un file json, così da renderle più facilmente modificabili, rendere le factory più chiare, e facilitare di un po l'aggiunta di torri.

Lista di nemici

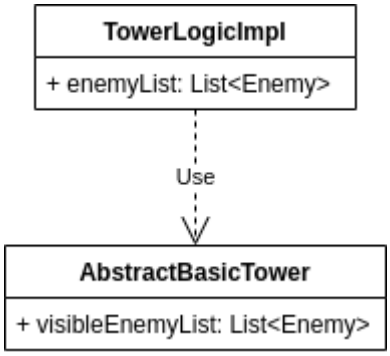


Fig. 2.2.3.5

PROBLEMA: Tutte le torri costruite sullo stesso campo devono poter accedere alla stessa lista di nemici

SOLUZIONE: Inizialmente avevo pensato ad una variabile globale in una classe statica, che poteva essere inizializzata solo una volta, poi mi sono reso conto che in caso dovessi istanziare più di due campi da gioco, non posso farlo perché la lista è una, poi in ogni caso è una brutta pratica l'uso di variabili globali. Di conseguenza ho preferito che tutte le torri tengano un puntatore all'oggetto lista contenente i nemici e quando si vuole visionare tale lista ne viene creata una copia.

TowerController

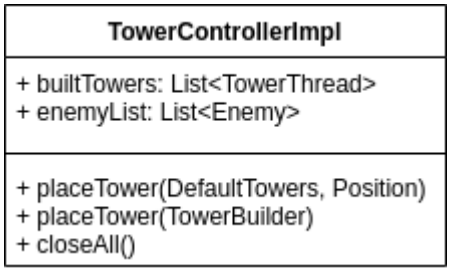


Fig. 2.2.3.6

PROBLEMA: Aggiungere le torri ad un campo, quando necessario cancellarle, e di disegnarle tutte a schermo ad ogni tick del gioco.

SOLUZIONE: Ho creato una classe nel controller che tiene traccia di tutte le torri costruite, per ogni istanza di questa classe generata simula un campo di gioco, dove questa classe controlla tutte le torri costruite. Siccome prima di costruire le torri serve la lista di nemici (sopracitata), quando viene istanziato un oggetto della classe TowerController bisogna che questa lista sia tra i campi del costruttore della classe. Così che sia il TowerController che istanzi la torre. Alla fine del gioco viene chiamata la funzione fatta apposta per cancellare tutte le torri.

Thread

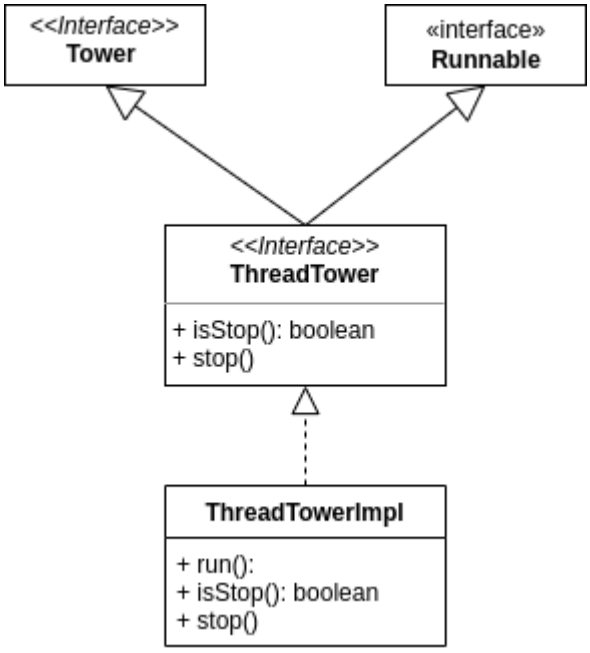


Fig. 2.2.3.7

PROBLEMA: Gestione di più torri, che in momenti diversi fanno cose diverse a seconda della loro implementazione interna

SOLUZIONE: All'inizio ho pensato che ogni tower dovesse avere a se un thread dedicato, poi mi sono accorto che poi l'interfaccia Tower non modellasse più a pieno solo il concetto di torre, e poi invalidava anche l'MVC, poiché se per assurdo si trovasse una maniera differente di controllare le torri senza usare i thread, l'interfaccia Tower estenderebbe Thread comunque, e sarebbe totalmente inutile. Di conseguenza mi sono accorto che potevo usare il pattern Decorator con delegazione che risolveva tutti i problemi della versione precedente.

PROBLEMA: Implementazione di una meccanica per colpire correttamente i nemici

SOLUZIONE: All'inizio ho pensato di usare dei proiettili per colpire dei nemici, poi mi sono reso conto che dietro ci stava troppa matematica, e mille altre complicazioni in più, di conseguenza ho deciso semplicemente che ogni torre ha le sue regole per stabilire se un nemico è da colpire o meno, e in caso rispettasse queste regole gli viene scalato della vita.

PROBLEMA: Controllare che in fase di costruzione delle di una torre non ci siano torri vicine

SOLUZIONE: Inizialmente ho pensato di dover controllare che le aree spaziate dalle torri non si sovrapponessero, poi mi sono reso conto che era troppo complicato, di conseguenza ho optato per controllare che il centro di una torre sia distante quanto la somma di metà diagonale di una e metà diagonale dell'altra, (poiché se hanno dimensioni diverse non hanno la stessa diagonale)

Capitolo 3

Sviluppo

3.1. Testing automatizzato

I testing sono stati fatti tramite la JUnit su Eclipse.

Elenchiamo le classi usate per il testing:

- **EnemiesBuilderImplTest** : testa la corretta creazione dei vari Enemy presenti nel gioco
- **EnemiesFactoryImplTest** : testa che i nemici vengano effettivamente spawnati
- **GameModelImplTest** : testa la corretta gestione dei soldi e della vita del giocatore
- **GenericEnemyTest** : testa il corretto funzionamento dei vari Enemy
- **MenuModelImplTest** : testa il corretto funzionamento della classe MenuModelImpl che si occupa di implementare il model dei menu.
- **AttackTest**: Vengono testati tutti i tipi di attacco definiti nel package **model.tower**, non vengono testate tutte le torri dentro il package factory, poiché essendo una composizione di quelle presenti nel **model.tower.** In particolare non vengono testate le tesla e i mortai, e quindi la loro

implementazione della findFirst(), poiché la loro implementazione è una funzione già testata, o composizione di funzioni già testate.
- **BuildTest:** Viene testato che il builder non accetti configurazioni, non valide di torri
- **FakeEnemy:** classe di appoggio per astrarsi dall'implementazione dei nemici, per muoverli più facilmente durante l'attacco delle torri e vedere se effettivamente sono troppo lontani
- **TowerController:** vengono testate tutte le funzionalità del controller delle torri

La View e l'interazione con l'utente è stata testata manualmente. L'applicazione è stata provata sia su Linux, che su Windows. Inoltre abbiamo utilizzato SpotBugs con il relativo plugin per un'analisi del codice in cerca di possibili bug e incorrettezze nella programmazione.

3.2. Metodologia di lavoro

La parte di analisi e modello del dominio è stata svolta assieme, così come l'architettura è stata creata in gruppo. Una volta creata la struttura del progetto, ognuno di noi si è cimentato nell'implementazione della propria parte in autonomia. Per fare ciò abbiamo utilizzato il **DVCS GIT** lavorando su **branch** dedicate per ogni membro, eventualmente effettuando delle **merge** sul branch main per poter testare le interazioni tra le diverse porzioni di codice. Avvicinandosi al completamento del progetto abbiamo lavorato sempre più a stretto contatto per migliorare la qualità delle interazioni tra le parti di ciascuno.

3.2.1. Gabos Norbert

- All'interno del package **tas.model.enemy** mi sono occupato dell'interfaccia Enemy e della sua implementazione, non che del builder e della factory
- All'interno del package **tas.controller** mi sono occupato con la collaborazione di Filippo Di Pietro e Leonardo Randacio del GameController e del package **enemy**, che si occupa della gestione degli Enemy presenti sul campo.
- All'interno del package **tas.view** mi sono occupato della creazione dello SquarePanel, dell'AdaptiveLabel e ho collaborato con Filippo nell'implementazione dell'ImageLoader.
- All'interno del package **tas.utils** mi sono occupato dell'implementazione delle classi JsonUtils, Position e TimeCurve

3.2.2. Leonardo Randacio

Mi sono occupato delle view e dei controller di tutte le classi che gestiscono i vari menu

del gioco, ad eccezione del gioco stesso, ovvero : SandboxMode, LevelsFull, SelectLevel, EndGame, MainMenu, Settings.

Ho gestito l'inventario presente nella partita, ovvero la classe InventoryView e i metodi che la gestiscono in GameController.

Nella SandboxMode faccio uso delle classi SquarePanel e JsonUtils fatte da Gabos Norbert.

Nel GameController faccio uso delle classi DefaultTowersInfo, TowerBuilder e TowerControllerImpl fatte da Filippo Di Pietro.

3.2.3. Filippo Di Pietro

Mi sono occupato dello sviluppo delle torri e quindi dei seguenti package

- Tutto il package **tas.model.tower**, contenente tutte le classi che possono descrivere torri con comportamenti e attacchi diversi.
- Tutto il package **tas.controller.tower.factory**, package destinato alle torri di default presenti nel gioco.
- Tutto il package **tas.controller.tower**, package destinato al controllo di più torri nello stesso campo.
- Tutto il package **tas.controller.tower.builder**, destinato a creare torri con tipi di attacco definiti in una enumerazione.
- Piccole parti per collegare il **GameController** con la **TowerController**.
- Ho collaborato con Norbert nell'implementazione dell'**ImageLoader**.

3.3. Note di sviluppo

3.3.1. Gabos Norbert

- Utilizzo della libreria JSON per la creazione degli Enemy.
- Utilizzo degli Optional per la gestione dei null.
- Utilizzo di reflection per eseguire dei test

3.3.2. Leonardo Randacio

- Utilizzo della libreria JSON per il salvataggio dei livelli.
- Algoritmo per la distanza punto-segmento per controllare se la posizione delle torri sia legale.

3.3.3. Filippo Di Pietro

- Utilizzo della libraria JSON per salvare informazioni basilari sulle Torri
- Uso di lambda expression
- Uso di Opzionali
- Uso di stream

Script presi da internet:

- Per il ridimensionamento delle immagini abbiamo preso uno script da Stack Overflow, per rendere il gioco più fluido
- Per il corretto ciclo del mainLoop abbiamo preso spunto da questa risposta
- Funzione trovata online che usa il Shoelace Theorem per trovare la distanza punto-retta.

Capitolo 4

Commenti finali

4.1. Autovalutazione e lavori futuri

4.1.1. Gabos Norbert

Sono abbastanza soddisfatto del nostro progetto, in particolare temevo che avendo tante istanze delle Entity, come torri e nemici, il gioco rallentasse di molto, ma alla fine riusciamo ad ottenere una media di 60 frame anche su computer poco performanti. Per quanto riguarda le parti che non mi soddisfano, la EnemyFactoryImpl, che utilizza un metodo diverso per ottenere il numero di Enemy da far spawnare ogni round. Ciò viene raggiunto mediante delle funzioni lineari, che dato il numero del round restituiscono il numero desiderato. Questa soluzione risulta poco estensibile, in quanto se si desidera aggiungere un nuovo Enemy, bisognerebbe scrivere un metodo apposito solo per esso. Una soluzione sarebbe quella di creare un nuovo parametro dentro al file json che tiene

traccia di ogni caratteristica degli Enemy, che gestisca lo “spawn rate” di ognuno e di creare una classe che traduca la funzione matematica da formato stringa in una funzione comprensibile alla factory. Un cosa che mi sarebbe piaciuto tanto implementare sarebbe di gestire tutti gli Enemy come thread in modo da rendere il gioco molto più fluido.

4.1.2. Leonardo Randacio

Sono contento di come ho portato avanti il progetto, essendo il mio primo progetto di queste proporzioni. Mi sono cimentato sia nella progettazione di parti totalmente indipendenti da quelle dei miei colleghi, ovvero la gestione dei diversi menù, ma anche nella progettazione di parti del progetto più intersecate con codice scritto dai miei colleghi, ad esempio la gestione dell’inventario per le torri. Ho apprezzato molto la collaborazione tra i componenti del gruppo, permettendoci di completare le conoscenze e le idee di ciascuno in modo da rendere il progetto finale un lavoro effettivamente svolto in gruppo e non tre progetti a sé stanti semplicemente uniti assieme una volta ultimati. Mi piacerebbe in futuro poter implementare meglio le impostazioni del gioco, dato che ora il menù impostazioni è di fatto vuoto. Si potrebbe aggiungere la possibilità allo user di modificare le statistiche di inizio partita (vita, soldi) o la difficoltà del gioco (ad esempio aumentando la vita dei nemici o numero di nemici). Un altro aggiornamento che si potrebbe fare sarebbe permettere al giocatore di cancellare un singolo livello in caso i livelli salvati fossero già il numero massimo.

4.1.3. Filippo Di Pietro

Sono complessivamente contento di come ho portato a termine la mia parte di codice, mi sembra facile da leggere, facile da comprendere, e mi pare che rispetti tutti i principi di qualità di programmi object-oriented, mi piace come ho strutturato il model, sono convinto che sia facilmente estensibile, sia all’interno del package che al di fuori, le due cose che non mi convincono sono: i costruttori pubblici delle classi che modellano le torri, che sono fondamentali per il builder che si trova in un altro package, l'altra sono le factory, non mi convincono fatte così, e non saprei come migliorarle. Tutto sommato non mi sembra di aver programmato male. In futuro mi piacerebbe estendere un po il package delle torri, aggiungendone una che ad esempio aumenta di danno ogni volta che attacca un target, finché non muore.

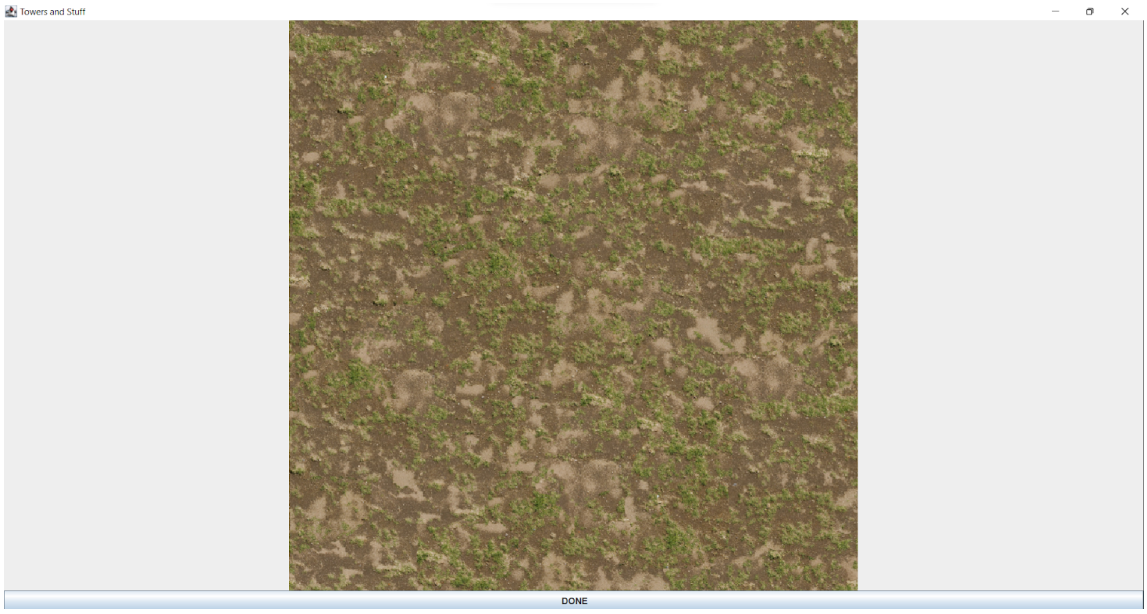
Appendice A

Guida utente

Al lancio dell’ applicazione, ci si troverà nel Main Menu.

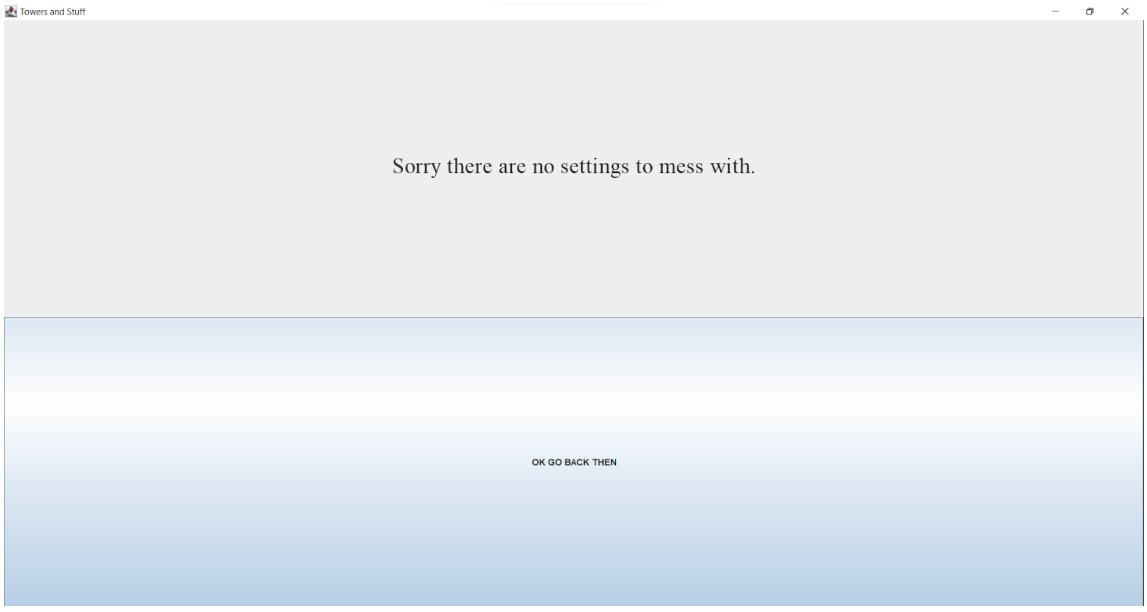
Da qui si può entrare in diversi menu cliccando il pulsante corrispondente:

SANDBOX MODE



Qui si può creare un nuovo livello da aggiungere ai livelli giocabili. Si clicca sul campo da gioco almeno due volte per far comparire il percorso bianco dei nemici. Ogni click seleziona un nuovo nodo per il percorso. Un livello per essere valido deve contenere almeno due nodi. Una volta finito di tracciare il percorso dei nemici si potrà salvare il livello cliccando sul pulsante DONE

SETTINGS



In questa pagina si possono modificare le impostazioni del gioco. Al momento non ci

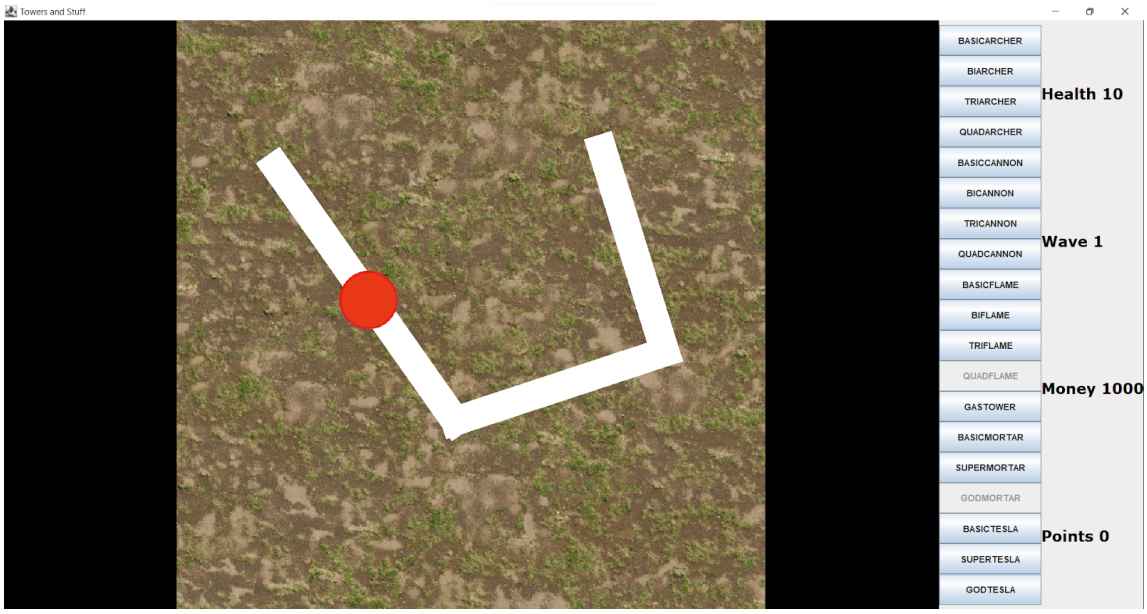
sono impostazione da modificare, dunque si può solo cliccare il pulsante OK GO BACK THEN per tornare al menu principale

EXIT

Cliccando questo pulsante si chiude l'applicazione

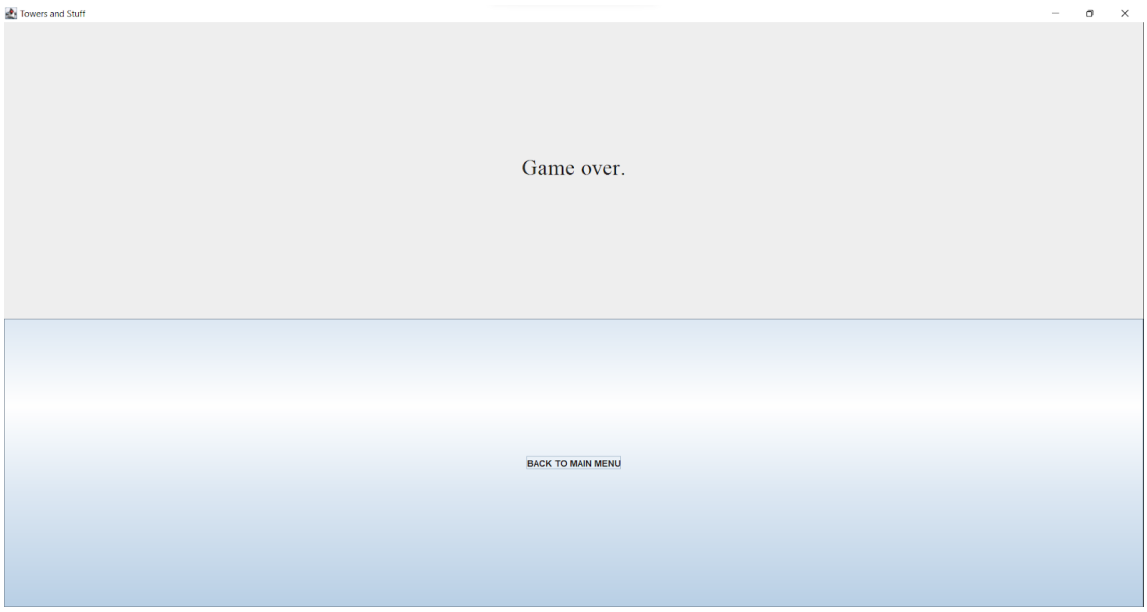
NEW GAME

Qui si può selezionare il livello che si desidera giocare. Una volta selezionato il livello del gioco si entra nella partita con il livello selezionato



I pallini colorati sono i nemici che tentano di raggiungere la fine del percorso. Sulla sinistra del campo da gioco è presente l'inventario. L'inventario contiene dei pulsanti per selezionare la torre e delle statistiche sulla partita (vita = Heath, ondata = Wave, soldi = Money, punti = Points). Se si hanno abbastanza soldi i pulsanti delle torri che si possono posizionare saranno attivati. Una volta cliccato sul pulsante di una torre che si può posizionare, il pulsante diventerà **rosso** per indicare che è stata selezionata quella torre. Ora si può cliccare sul campo da gioco per posizionare la torre in quel luogo. Non si possono posizionare più torri una sopra l'altra, troppo vicine al bordo o sopra al percorso dei nemici.

Quando si perdono tutti i punti di vita la partita finisce:


Towers and Stuff
Game over.
BACK TO MAIN MENU