Relazione
“Tower Defense”

Giacomo Arienti, Giacomo Boschi,
Davide Fiocchi, Pietro Pasini

7 giugno 2024

Indice

Capitolo 1

Analisi

Il team si propone di sviluppare un videogioco in stile *tower-defense*. Il genere prevede di difendere una base da orde di nemici costruendo delle difese (spesso rappresentate graficamente con delle torri da cui il nome). Nella versione semplificata che ci proponiamo di sviluppare, il gioco si configura come un arcade nel quale quindi l'unico obiettivo è sopravvivere al maggior numero possibile di ondate.

Figura 1.1: Un tipico gioco tower-defense

1.1 Requisiti

Requisiti funzionali

- Con l'avvio della partita dovrà iniziare una sequenza infinita di ondate di nemici di difficoltà crescente che punteranno ad attraversare la mappa, muovendosi su un percorso predefinito.
- Il giocatore inizierà la battaglia con un certo numero di vite che diminuirà ogni qual volta un nemico raggiunga il proprio obiettivo, terminando la partita con un'eventuale sconfitta se queste dovessero arrivare a zero.
- Il gioco dovrà consentire all'utente di edificare torri difensive di vari tipi in posizioni scelte strategicamente tra quelle disponibili, nel momento in cui lo ritiene opportuno.
- All'abbattimento di un nemico il giocatore otterrà potere d'acquisto che potrà spendere per la costruzione e il potenziamento delle difese. Sarà anche possibile smantellare le suddette per riottenere parte del loro valore.

Requisiti non funzionali

- Dovrà essere possibile selezionare la risoluzione desiderata della finestra di gioco.
- Il software potrà essere giocato con un frame rate e prestazioni accettabili su pc di fascia media.
- Il gioco dovrà essere compatibile sui seguenti sistemi operativi: Windows, Mac Os, Unix (e derivati).

1.2 Analisi e modello del dominio

Abbiamo individuato quattro macro-sezioni del modello, in modo da suddividerlo equamente tra i membri del gruppo.

Statistiche di gioco

Questa categoria prevede l'entità che contiene e modifica opportunamente i valori del numero di vite, delle monete spendibili, dello stato del gioco (in pausa, sta giocando, game over) e della wave corrente (che determina anche il punteggio ottenuto dal giocatore).

Mappa e posizioni

La mappa di gioco sarà suddivisa in celle che potranno essere di due categorie, quelle descriventi il percorso su cui si muoveranno i nemici e quelle edificabili. Sarà presente un'entità dedicata alla generazione e memorizzazione della mappa, ossia l'insieme delle celle. Questa dovrà inoltre occuparsi del posizionamento delle difese e comunicare ai nemici la posizione in cui muoversi.

Nemici e ondate

Ogni nemico è caratterizzato da un tipo il quale determina la velocità con cui percorre mappa, la massima quantità di danni che può ricevere prima di essere sconfitto (punti vita massimi) ed eventualmente l'aspetto. Nemici dello stesso tipo sono indistinguibili se non per posizione occupata e punti vita rimanenti. Al di fuori di velocità e punti vita massimi, diversi nemici non realizzano comportamenti diversi: tutti percorrono senza deviazioni il sentiero che conduce alla base la quale, se raggiunta, verrà danneggiata (diminuzione dei punti vita del giocatore); se invece un nemico viene sconfitto prima di raggiungere la fine del sentiero, rende disponibile al giocatore un potere di acquisto commisurato al suo "power level" (prodotto tra punti vita massimi e velocità, numero direttamente proporzionale ai danni su quantità di tempo che le difese devono erogare per abbatterlo). La generazione dei nemici è strutturata in "ondate" o "wave". Le ondate sono sequenze ordinate di tipi di nemici che devono essere generati uno in seguito all'altro sulla posizione di partenza della mappa.

Difese

Sulle celle edificabili descritte precedentemente sarà possibile costruire le difese del gioco. Le difese sono le entità responsabili di proteggere il punto di fine della mappa, danneggeranno i nemici tramite una logica di attacco, infliggendo danno in base alle loro statistiche (danno, velocità di attacco, range). Per costruire una torre sarà necessaria una certa quantità di valuta. Si potrà inoltre vendere una difesa costruita per recuperare una parte di tesoro spesa e liberare una cella edificabile della mappa.

Per il corretto funzionamento del gioco è necessario che queste quattro componenti siano coordinate e comunichino tra di loro: la *Mappa* deve poter accedere alle monete a disposizione del giocatore (*Statistiche di gioco*) e alle

Difese disponibili per permettere di costruire sulla cella selezionata, le *Difese* necessitano di conoscere le posizioni e la vita dei *Nemici* per poter scegliere quali attaccare, i quali a loro volta devono chiedere alla *Mappa* il percorso da seguire. Ogni volta che un *Nemico* raggiunge la fine deve comunicare la perdita di una vita alle *Statistiche di gioco* le quali a loro volta saranno responsabili di far partire l'ondata successiva una volta che tutti i *Nemici* saranno stati eliminati.

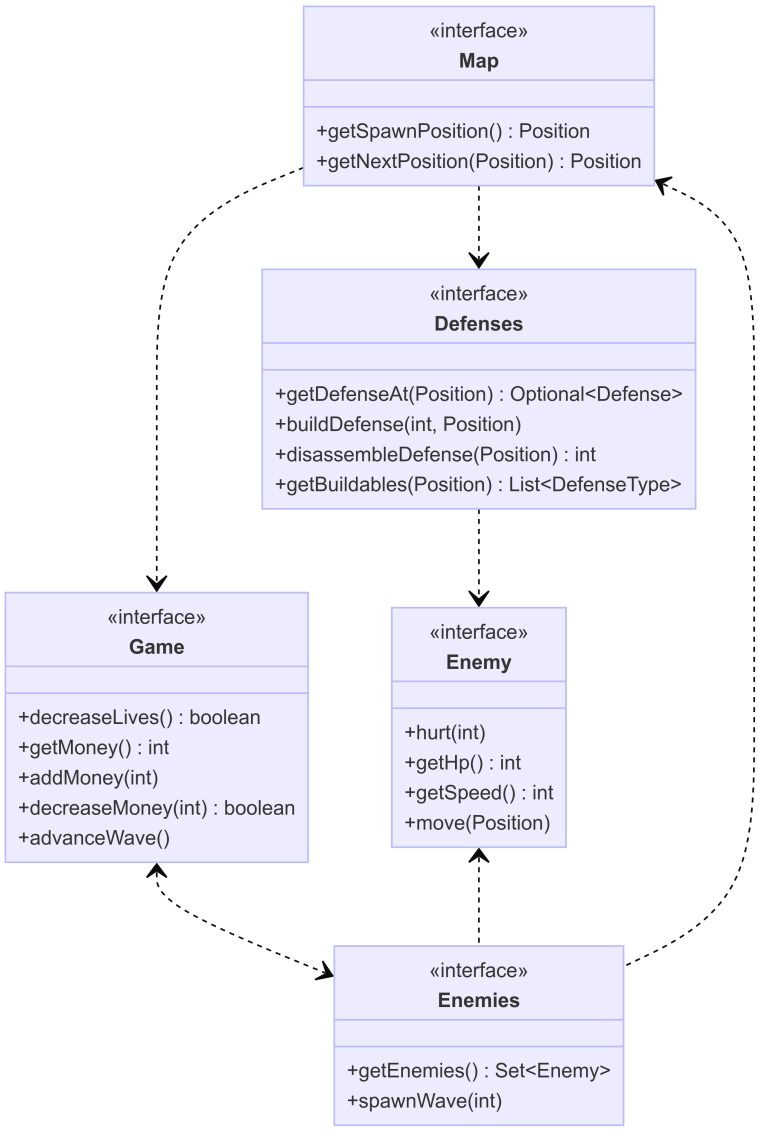

Figura 1.2: Il dominio del gioco

Capitolo 2

Design

2.1 Architettura

L'architettura dell'applicazione segue il pattern MVC. Il Model, una volta inizializzato dal Controller, affida in modo trasparente all'esterno la gestione della logica ad alcuni Manager i quali gestiscono vari aspetti del gioco, come i nemici, le difese, la mappa e le statistiche della partita. Il Controller funge da intermediario tra Model e View, limitando l'accesso esterno al Model e esponendo unicamente oggetti per il trasferimento di informazioni. Questo significa che gli altri componenti dell'applicazione, come la View, possono accedere solo a una rappresentazione non modificabile dei dati nel Model, garantendo un maggiore controllo sull'integrità dei dati. Per questa specifica implementazione grafica, la View delega il rendering alla Window. Pertanto, l'implementazione della View, così come è stata definita, è facilmente intercambiabile poiché non dipende dalle classi sottostanti.

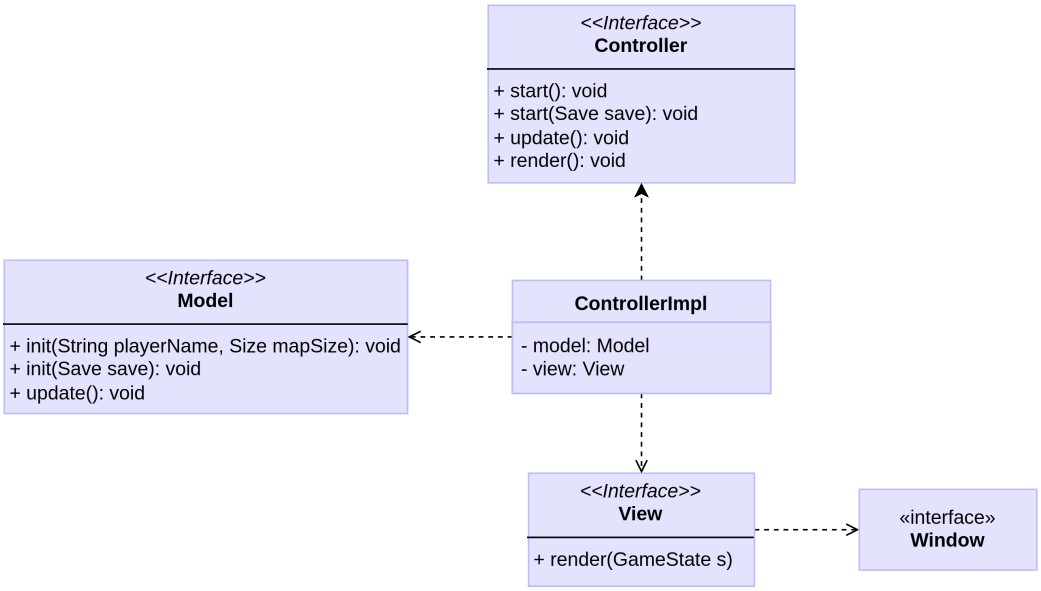

Figura 2.1: Architettura MVC

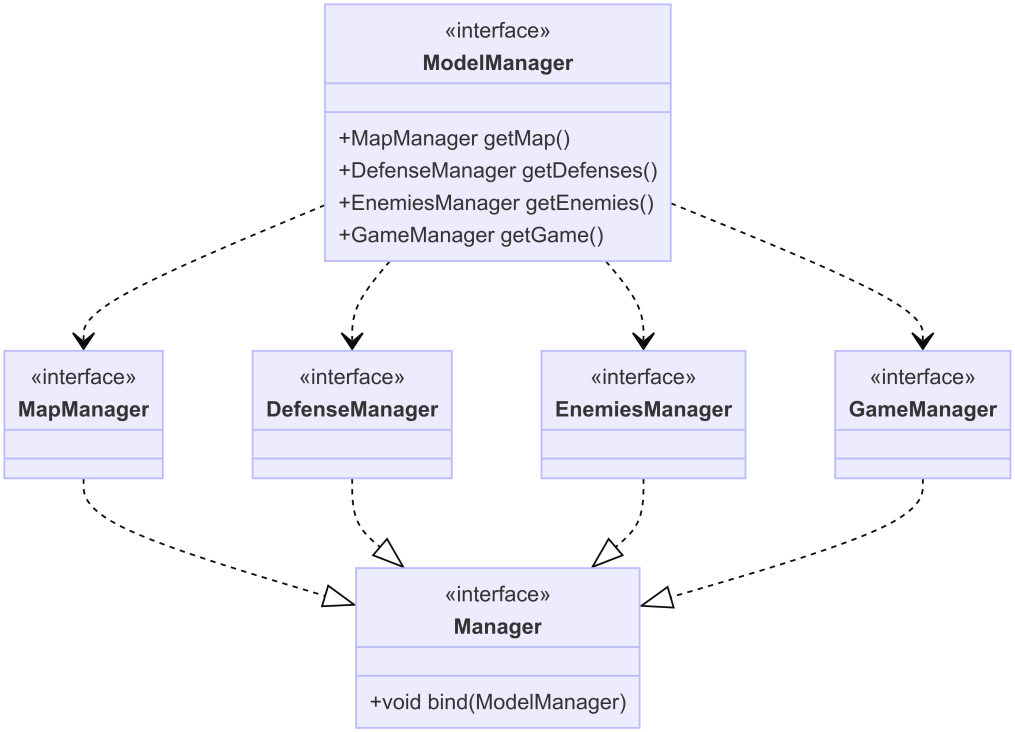

Figura 2.2: Architettura dei Manager

2.2 Design dettagliato

Giacomo Arienti

Gestione del Loop di Gioco

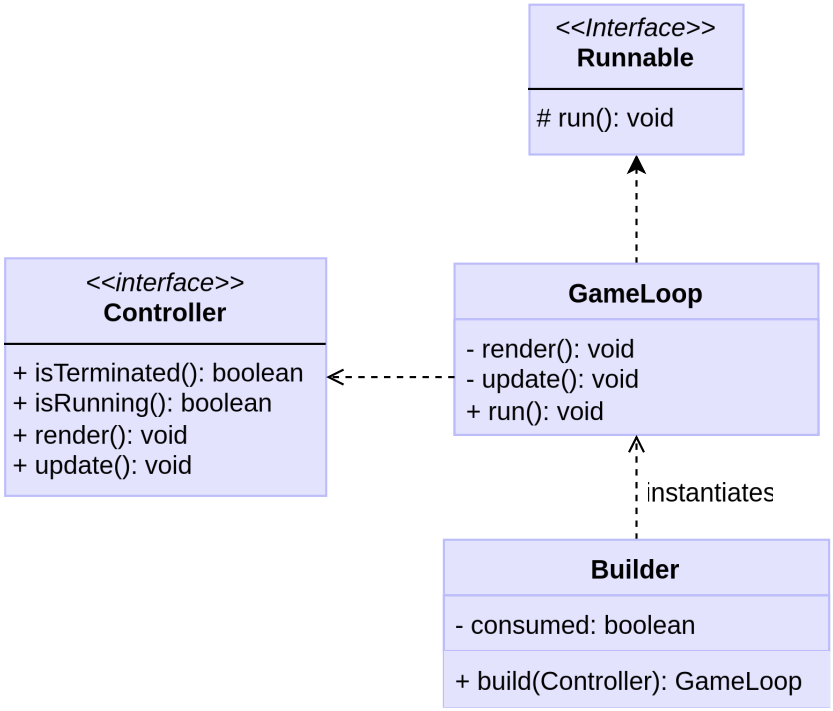

Figura 2.3: UML GameLoop

Problema È necessario implementare un loop di gioco che gestisca efficacemente l'avanzamento del gioco e il rendering, garantendo buone prestazioni (∼60 FPS) con un basso utilizzo di risorse. Inoltre, tutte le operazioni devono essere eseguite in perfetta sincronia tra loro.

Soluzione Per assicurare l'esecuzione delle operazioni a intervalli di tempo precisi, ho creato una classe GameLoop specifica, che opera su un thread parallelo a quello principale per garantire la corretta tempistica delle istruzioni. Per evitare la creazione di loop di gioco multipli, ho adottato il *Builder Pattern*, in modo che, una volta utilizzato il builder, non sia possibile creare ulteriori istanze di GameLoop (e quindi ulteriori thread). In pratica, il gameloop, a seconda del tempo trascorso dall'ultimo aggiornamento/rendering e dello stato del gioco (ad esempio, se il gioco è in pausa non vengono eseguite

operazioni), invoca il Controller il quale delega l'aggiornamento al Modello e il rendering alla View.

Aggiornamento delle entità interessate alle Statistiche di Gioco

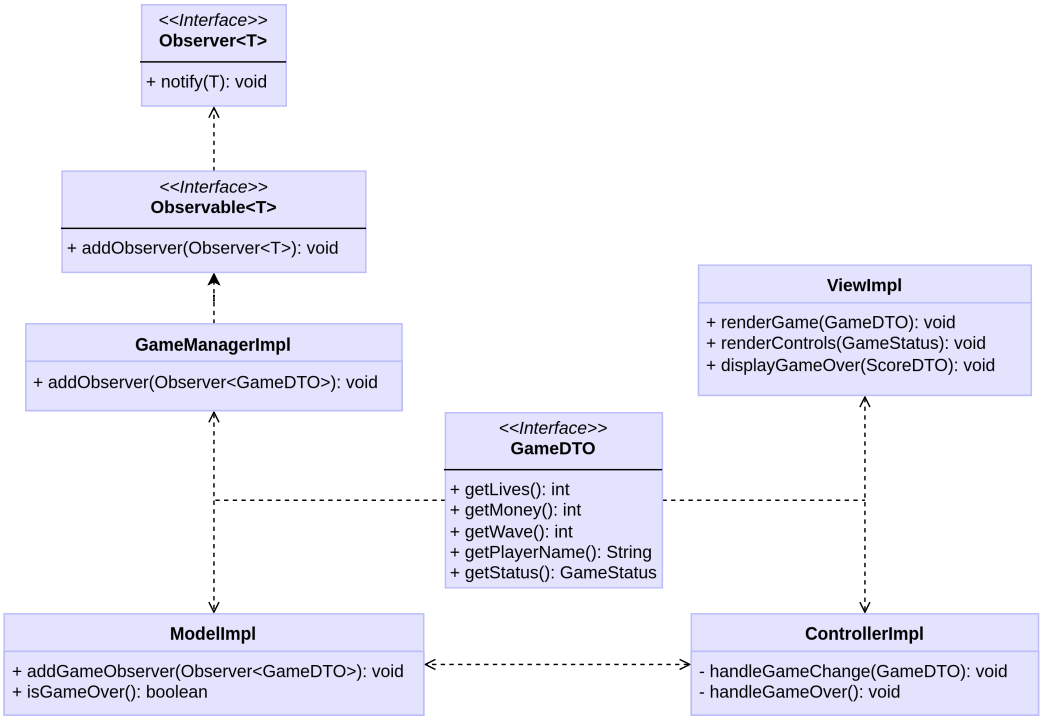

Figura 2.4: UML GameManager

Problema In un videogioco, è cruciale notificare le altre componenti del modello e della grafica quando le statistiche di gioco cambiano, come la diminuzione delle vite o l'aumento delle monete. Per fare ciò è necessario gestire efficacemente queste notifiche per mantenere sincronizzate tutte le parti del sistema.

Soluzione Per risolvere questa problematica, ho implementato l'interfaccia *Observable* sulla classe GameManager, sfruttando l'*Observer Pattern*. Questa soluzione consente ad altre componenti di "registrarsi" per ricevere gli aggiornamenti direttamente dal GameManager. Ciò permette ad altre parti del modello (ad es. la mappa), di accedere alle statistiche di gioco aggiornate in modo efficiente. Questo approccio consente di effettuare operazioni asincrone, come il rendering delle statistiche di gioco, evitando così di dover ridisegnare tali elementi ad ogni frame, a meno che non vi siano cambiamenti.

Inoltre, per prevenire la creazione di dipendenze superflue all'interno delle View rispetto al Modello ho adottato un approccio basato sui *DTO* (Data Transfer Object): questi rappresentano copie immutabili delle statistiche di gioco, consentendo un trasferimento sicuro e senza la necessità di passare il riferimento del modello stesso.

Riutilizzo di codice per gli elementi di grafica

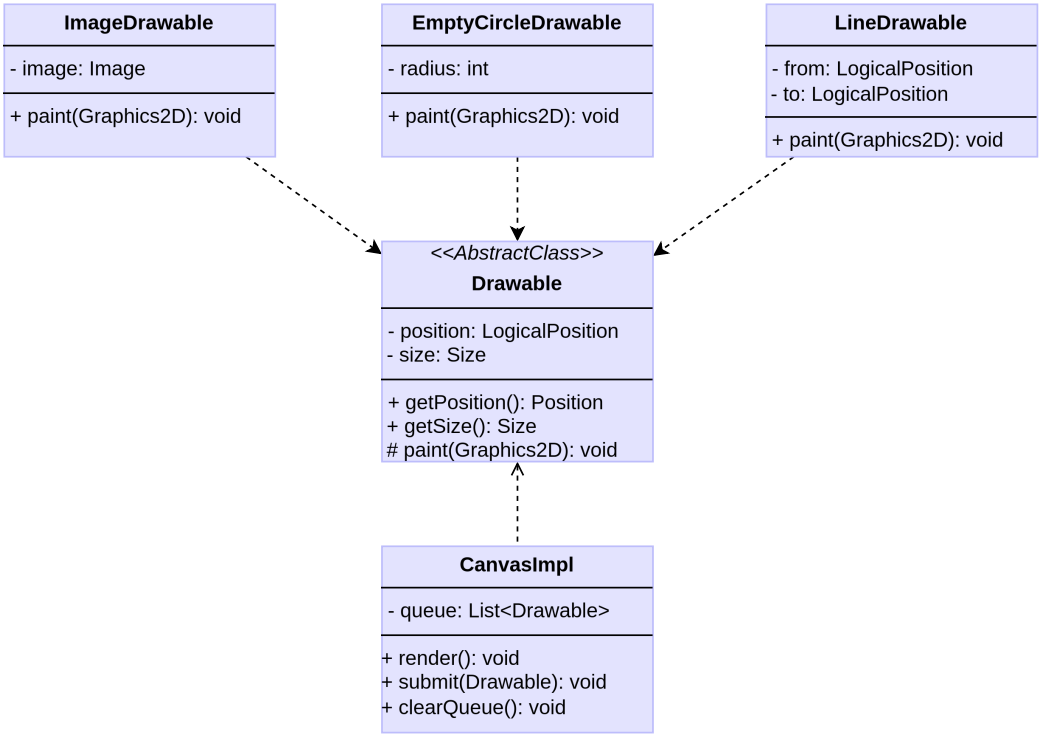

Figura 2.5: UML Drawable

Problema Durante lo sviluppo dell'aspetto grafico del gioco, è emersa la necessità di gestire diverse categorie di oggetti da visualizzare a schermo, con alcune caratteristiche comuni. Questa situazione ha reso indispensabile trovare una soluzione per riutilizzare efficacemente il codice condiviso.

Soluzione Per risolvere questa problematica, ho realizzato una *Abstract Class* Drawable che implementasse le funzionalità di base necessarie, come ad esempio la scalabilità dell'immagine o della figura in relazione alle dimensioni della mappa e del canvas. Le sottoclassi, invece, sono state responsabili di implementare il proprio metodo di rendering in base alle caratteristiche

specifiche dell'oggetto. Per garantire la sincronizzazione del rendering con i tempi stabiliti dal gameloop, ho introdotto una coda per contenere gli oggetti da renderizzare.

Giacomo Boschi

Generazione delle difese

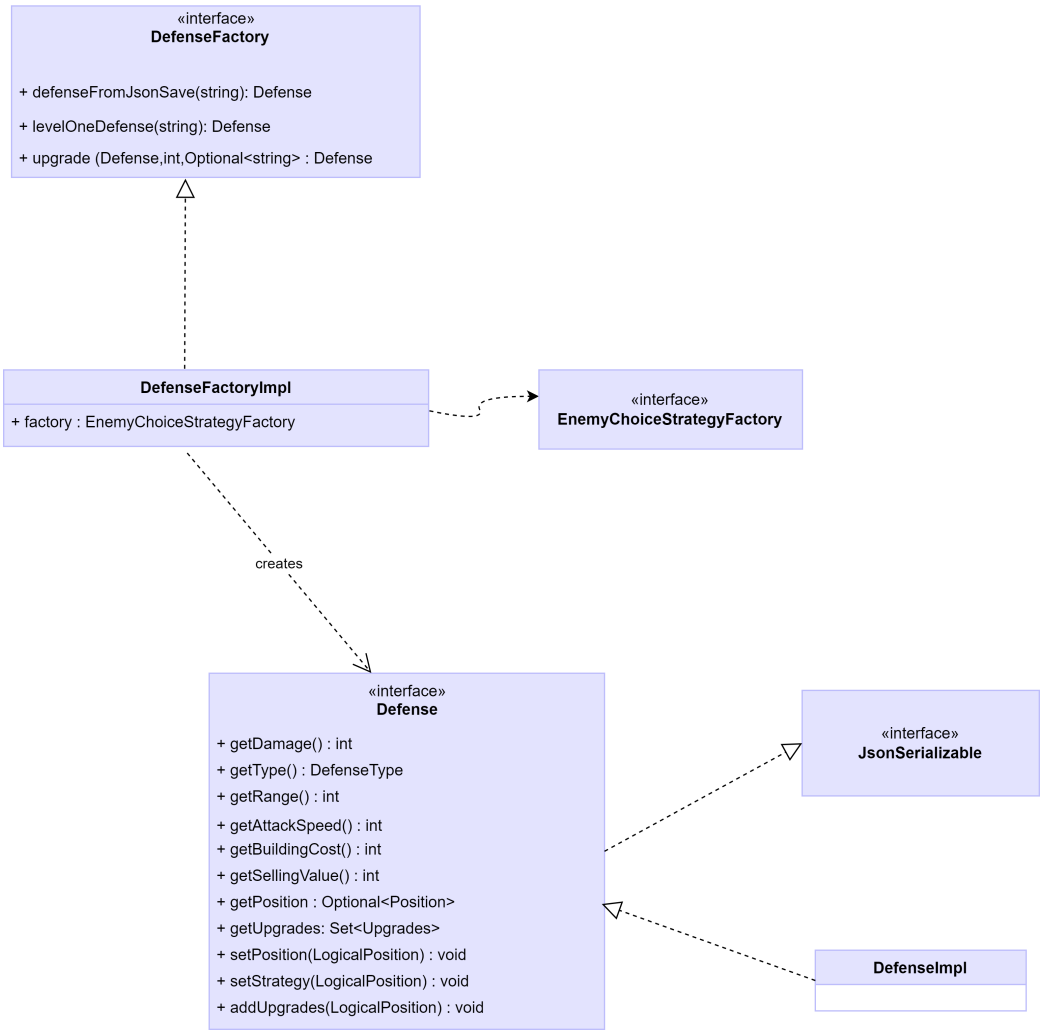

Figura 2.6: Le difese sono generate da una factory.

Problema Si vuole avere la possibilità di generare diversi tipi di difese senza necessità di avere codice estremamente simile in parti diversi dell'architettura. Utilizzare l'ereditarietà risulta dispersivo in quanto richiederebbe

un file nuovo ogni volta che si desidera generare torri lievemente differenti, siccome gli elementi cambiano da una struttura all'altra sono la strategia usata e le istanze delle statistiche.

Soluzione Si utilizza una Factory (DefenseFactory) che permette la creazione di una Defense a partire da un file json di statistiche (di salvataggio o pre-esistente) applicato ad una implementazione di base (DefenseImpl) uguale per tutti i tipi. Ne risulta che per gestire le differenze tra torri è sufficiente l'utilizzo di un enum chiamato DefenseType, il quale ha un suo getter all'interno della difesa stessa.

Problema I file json usati per generare le difese dal menù di costruzione non possono avere una posizione salvata in quanto sono template. Vi è una necessità di creare una difesa ed essere capaci si impostare la posizione dopo la creazione.

Soluzione Si implementa l'utilizzo di un setter per la posizione, e per assicurarsi il corretto controllo del valore si incapsula il getter in un Optional.

Gestione delle strategie

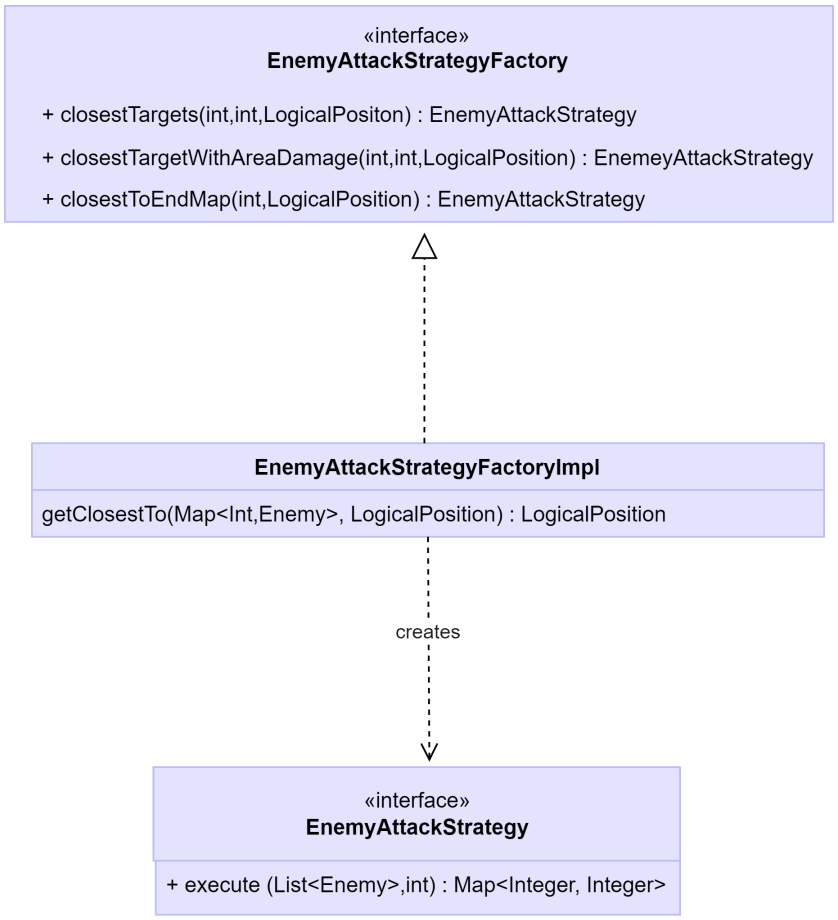

Figura 2.7: Una factory genera le strategies.

Problema Analogamente al problema precedente,si vuole poter generare le strategie di attacco in maniere flessibile e scalabile, senza avere la necessità di generare una classe nuova per ogni strategia. Si vuole inoltre non essere vincolati troppo alla difesa che si sta usando.

Soluzione La strategia viene incapsulata in una sua classe a parte chiamata EnemyChoiceStrategy, in modo da non essere vincolata alla difesa. Si utilizza nuovamente una Factory (EnemyChoiceStrategyFactory) per la generazione delle strategie passando ai vari metodi generatori dati più elementari come una range, posizione di riferimento e numero di bersagli. In questo modo la factory delle difese può scomporre in sottoproblemi la generazione delle torri,

utilizzando un istanza della EnemyChoiceStrategyFactory per le strategie e i metodi di DefenseImpl per la generazione di difese da file json.

Problema Poichè il funzionamento di una strategia è basato su interfacce funzionali, non è possibile salvare tale strategia dentro ad un file json.

Soluzione La DefenseFactory dispone di un metodo interno per associare la strategia in base al tipo di Difesa (ovvero il DefenseType).

Gestione della grafica

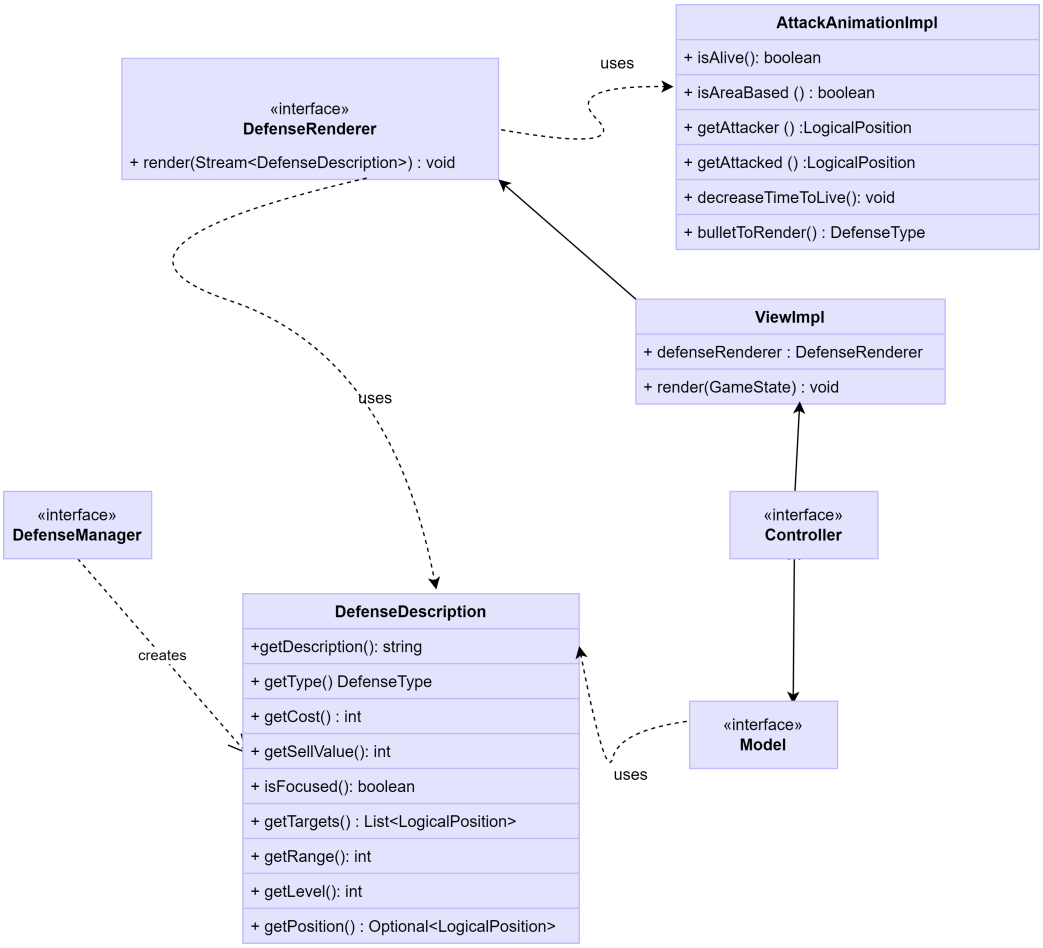

Figura 2.8: Funzionamento della view per difese.

Problema Vi è la necessità di renderizzare sia le difese nella loro posizione di mappa, sia la necessità di renderizzare un qualche tipo di immagine per far capire quando un torre sta attaccando un nemico.

Soluzione Si scompone nuovamente in sotto-problemi: utilizzando il dto relativo alle difese chiamta DefenseDescription, il DefenseManager è in grado di passare a controller le informazioni sulla difesa e quali bersagli sta attaccando. Quando tali informazioni vengono passate da controller a view, il DefenseRenderer si occupa di scomporre la visualizzazione delle difese in parti: il rendering delle difese e quello degli attacchi, che è delegato alla classe AttackAnimationImpl, la quale si genera sempre da DefenseDescription.

Davide Fiocchi

Gestione della morte dei nemici tramite Observer

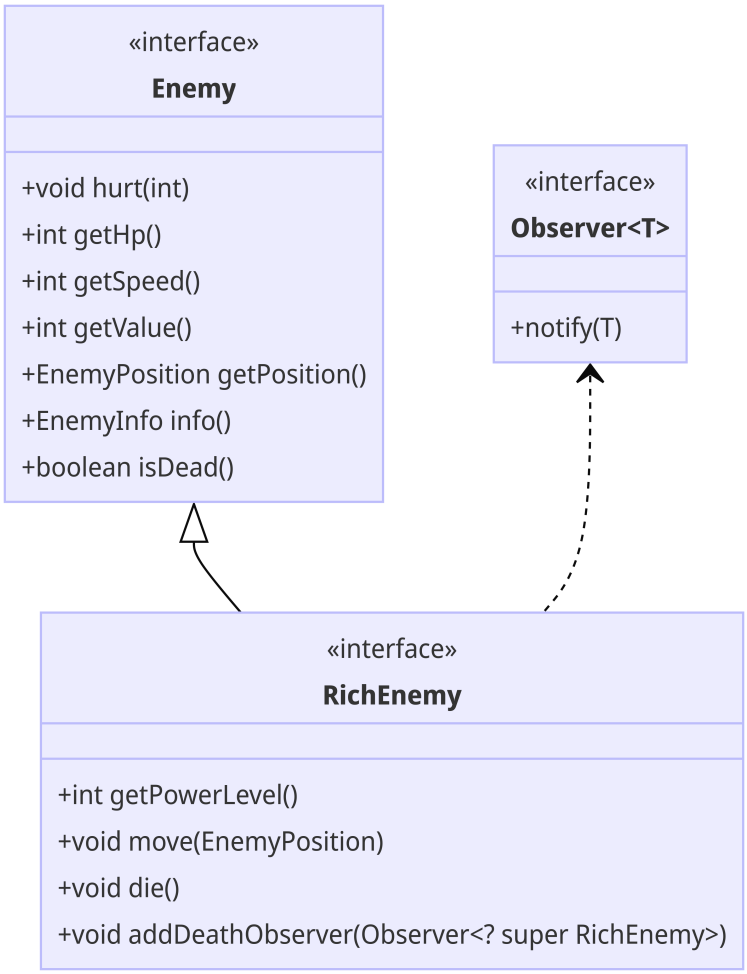

Figura 2.9: Relazione tra enemy e Observer

Problema Alla morte di ogni nemico è necessario svolgere diverse azioni, come aggiornare le strutture dati che lo contenevano o aggiungere il loro valore al potere di acquisto a disposizione del giocatore.

Soluzione Ho deciso di adottare il pattern Observer per notificare ogni entità interessata alla morte dei nemici: ognuno dei suddetti espone il metodo addDeathObserver per aggiungere un Observer alla lista di quelli che

verranno notificati alla sua morte, comportandosi sostanzialmente da Observable. Questo approccio permette di mantenere un basso accoppiamento tra i componenti del sistema. Nuovi osservatori possono essere aggiunti senza modificare la logica interna del nemico, rendendo il sistema più liberamente estensibile. Ogni volta che un nemico muore, notifica gli osservatori registrati chiamando il loro metodo notify e passandogli un riferimento a sé stesso in modo che possano richiedere le informazioni necessarie a reagire all'evento (per esempio il valore). Questo semplifica molto l'aggiunta di nuove azioni da eseguire alla morte del nemico. Altre soluzioni seppur alleggeriscano il momento della creazione del nemico (non si deve inizializzare una struttura dati per contenere gli osservatori) sono più onerose in un secondo momento: ci basti pensare che è facile siano in vita anche decine di nemici contemporaneamente, su cui a ogni ciclo bisognerebbe effettuare controlli al fine di aggiornare il resto del modello. Invece, il dominio prevede che a ogni ciclo venga generato al massimo un nemico e quindi le operazioni di aggiunta degli osservatori hanno un impatto molto più distribuito nel tempo.

Generazione dei nemici mediante Factory Method

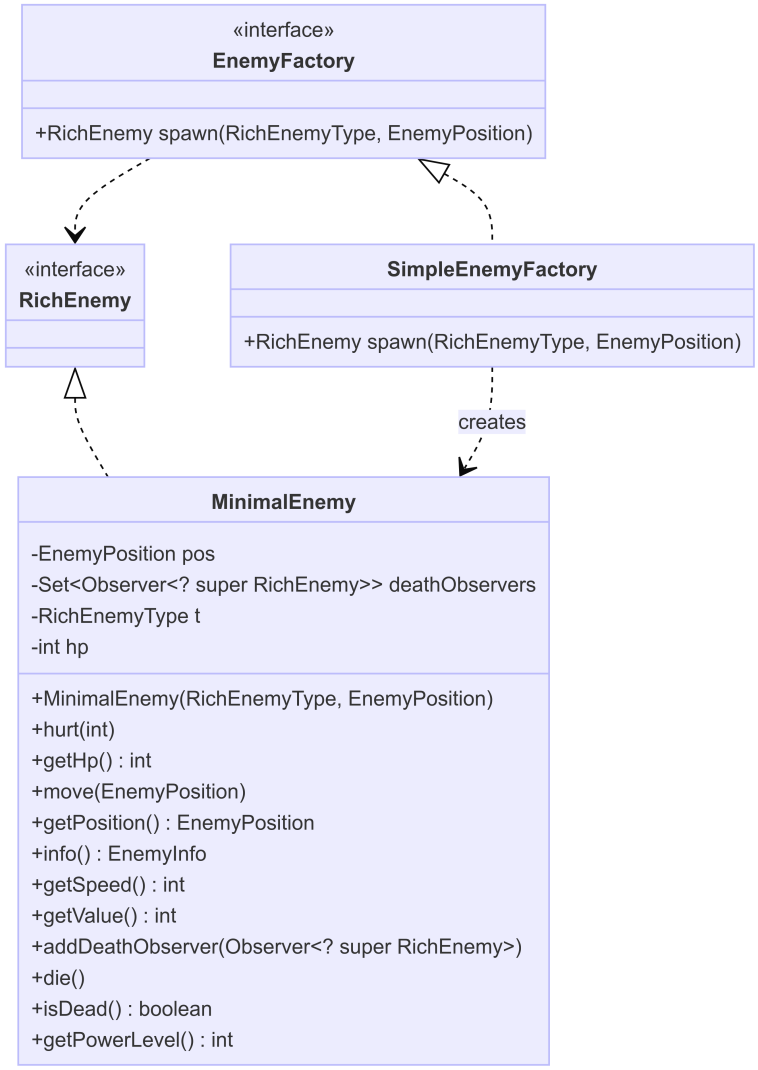

Figura 2.10: Le classi che realizzano il pattern FactoryMethod

Problema La creazione di diverse tipologie di nemici potrebbe diventare complessa e disordinata se non gestita in maniera dedicata. Inoltre, ogni tipo di nemico nonostante abbia diversi parametri che lo contraddistinguono dovrebbe avere lo stesso comportamento generale e creare sottoclassi diverse per ognuno risulterebbe dispersivo.

Soluzione L'utilizzo del pattern Factory Method per la creazione dei nemici risolve efficacemente questi problemi. Creando una classe dedicata, come SimpleEnemyFactory, si ottiene un'organizzazione più chiara e modulare del codice. Questa implementa l'interfaccia EnemyFactory, che espone un metodo spawn per creare un nuovo nemico di un certo tipo in una posizione specifica. All'interno di SimpleEnemyFactory, è definita una classe interna privata chiamata MinimalEnemy, che implementa RichEnemy ed i cui attributi sono popolati sulla base del tipo e della posizione dati come parametri alla Factory. In questo modo aggiungere nuovi tipi di nemici è semplice e richiede solo di aggiungere nuovi oggetti di tipo EnemyType.

Generazione delle ondate tramite Strategy

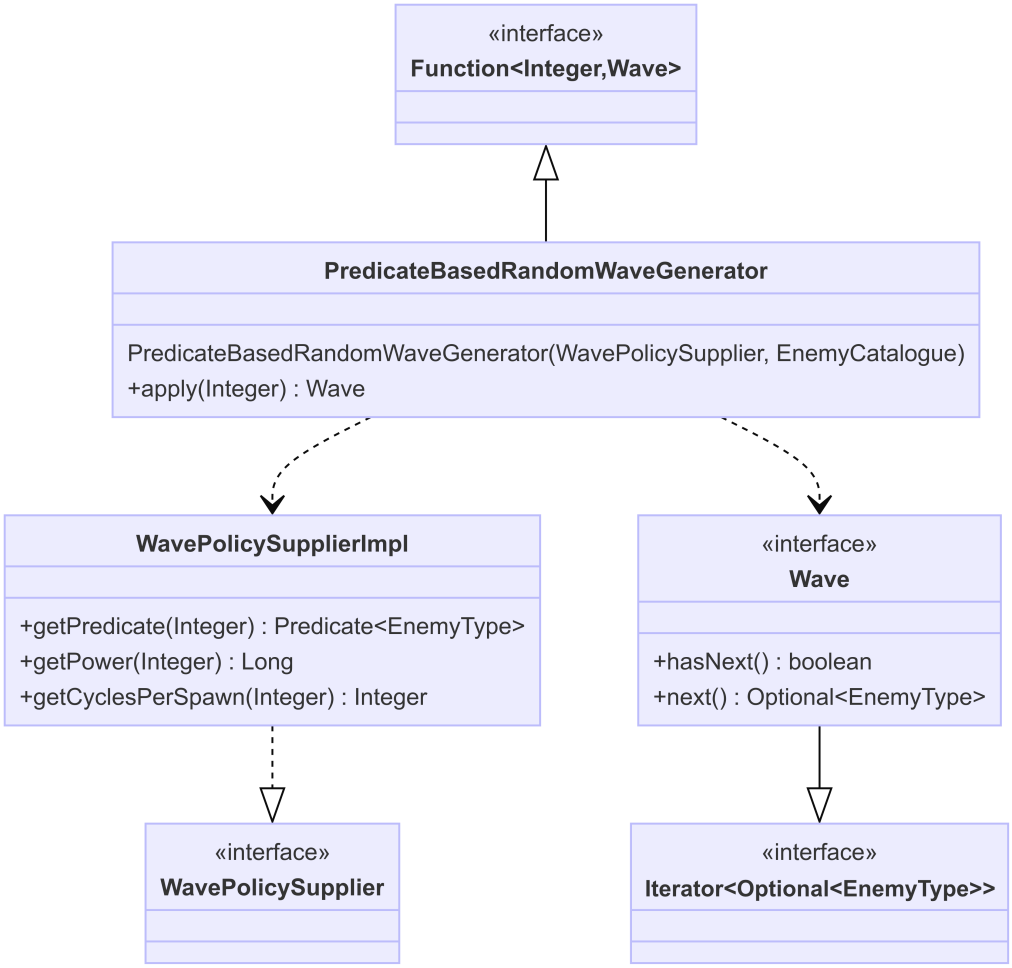

Figura 2.11: Classi che collaborano alla generazione delle ondate

Problema Nel contesto di un gioco tower defense, la generazione delle ondate di nemici è una parte cruciale che deve essere ben bilanciata per rendere il gioco divertente. Le ondate devono variare in termini di tipi di nemici, numero di nemici e distribuzione temporale, il che può diventare complesso da gestire in maniera flessibile. Gestire direttamente all'interno di una singola classe tutti questi aspetti di generazione delle ondate può portare a un codice rigido, difficile da mantenere e aggiornare, specialmente se si vuole modificare o estendere il comportamento delle ondate in futuro.

Soluzione Per risolvere il problema della generazione flessibile e modulare delle ondate, è stato adottato il pattern Strategy in modo innestato. Questo pattern permette di incapsulare le diverse strategie di generazione delle ondate in classi separate, che possono essere facilmente scambiate o modificate senza toccare il codice che utilizza queste strategie. Le classi concrete che implementano, l'interfaccia WavePolicySupplier agiscono come strategia per la scelta dei contenuti delle ondate definendo i metodi necessari per ottenerne le politiche di generazione, come i tipi di nemici ammessi, la potenza massima (somma dei "power level" dei nemici) e la frequenza degli spawn. Senza alcuna pretesa di aver realizzato un gioco ben bilanciato o "avvincente", ne abbiamo in questo modo predisposto le basi: anche se non è stato possibile farlo nel monte ore, sarebbe facile realizzare diverse "difficoltà" aggiungendo altre implementazioni di WavePolicySupplier. La stessa PredicateBasedRandomWaveGenerator si configura come strategia di generazione delle ondate, e può essere sostituita da qualsiasi altra Function da Integer a Wave, per esempio per un'eventuale futura implementazione di ondate con lunghezza infinita.

Pietro Pasini

Unica mappa costituita da celle diverse

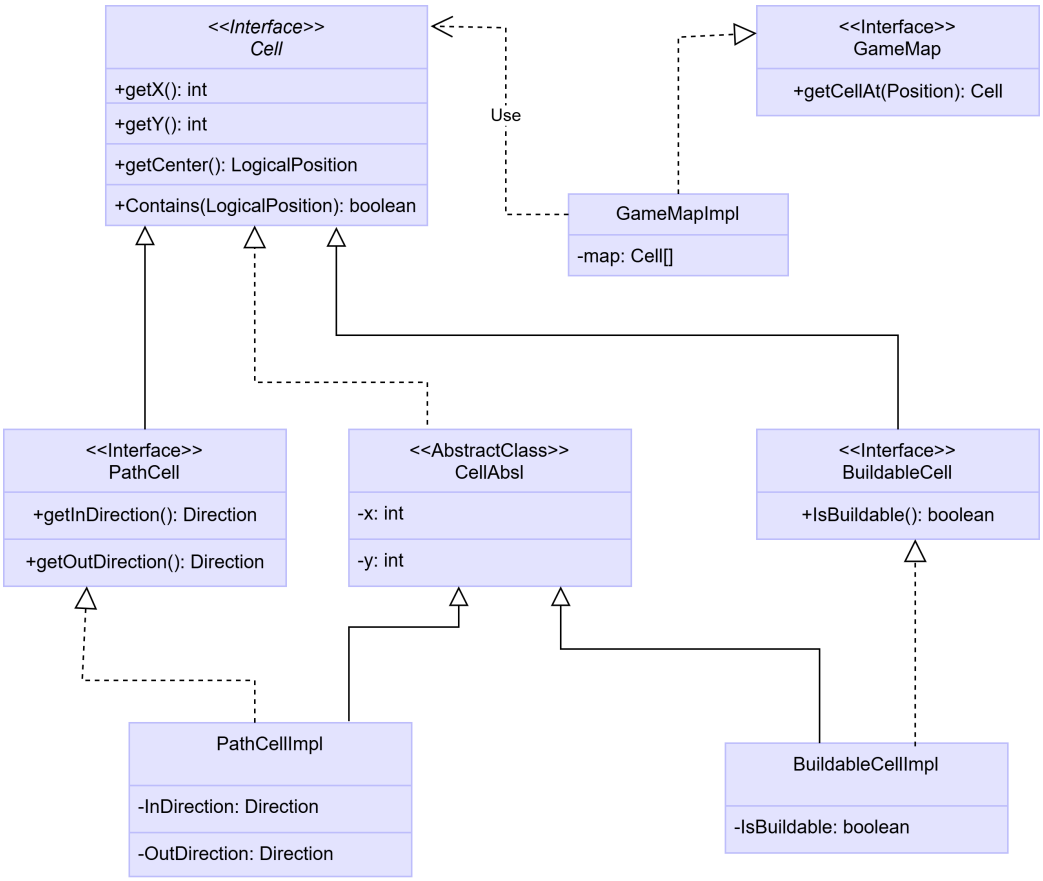

Figura 2.12: UML Cell

Problema La mappa di gioco è costituita da una griglia di celle di dimensioni uguali, ciascuna con delle proprie coordinate univoche e diversi metodi relative a queste. Tuttavia è necessario che quelle che costituiscono il percorso abbiano a tale scopo dei metodi specifici, a differenza delle restanti che presenteranno invece la possibilità o meno di costruire difese su di esse. Dunque ogni cella istanziata deve obbligatoriamente ed esclusivamente appartenere ad una delle due categorie.

Soluzione Seguendo il pattern Composite, ho adottato l'architettura descritta nell'UML figura 2.6. Utilizzando la classe CellAbs, i metodi comuni a

tutte le celle sono implementati una sola volta, ugualmente per tutte, mentre le due specializzazioni che la estendono devono implementare solamente i metodi caratteristici della propria interfaccia. Essendo questa una classe astratta non è possibile istanziare una cella che non appartenga ad una delle due categorie implementate. In questo modo la classe GameMapImpl può contenere tutte le celle in un unico vettore di Cell.

Posizione logica

Problema Per gestire il movimento dei nemici è emersa la necessità di suddividere le celle in unità molto più piccole. Inizialmente ho pensato di suddividere le celle del modello in pixel che dovevano corrispondere alla rappresentazione grafica, ma ho capito che questa soluzione non era corretta in quanto non adattabile a diversi tipi di view, e dipendente dalle dimensioni della finestra di gioco.

Soluzione Abbiamo creato la classe LogicalPosition, un'estensione di Position, così che fosse un tipo distinguibile dalla posizione delle celle, a cui ho aggiunto dei metodi per ricavare la cella di appartenenza. Inoltre questa classe contiene il campo statico SCALINGFACTOR che determina il rapporto con la dimensione delle celle. Per la rappresentazione su schermo sarà la view ad occuparsi della conversione in posizione grafica, ad esempio in pixel, sapendo il numero di celle e lo SCALINGFACTOR. Con questa soluzione le posizioni nel modello sono indipendenti dalle quelle nella view, si può cambiare a piacimento la precisione degli spostamenti nel modello o la dimensione della finestra della GUI.

Gestione posizionamento delle difese

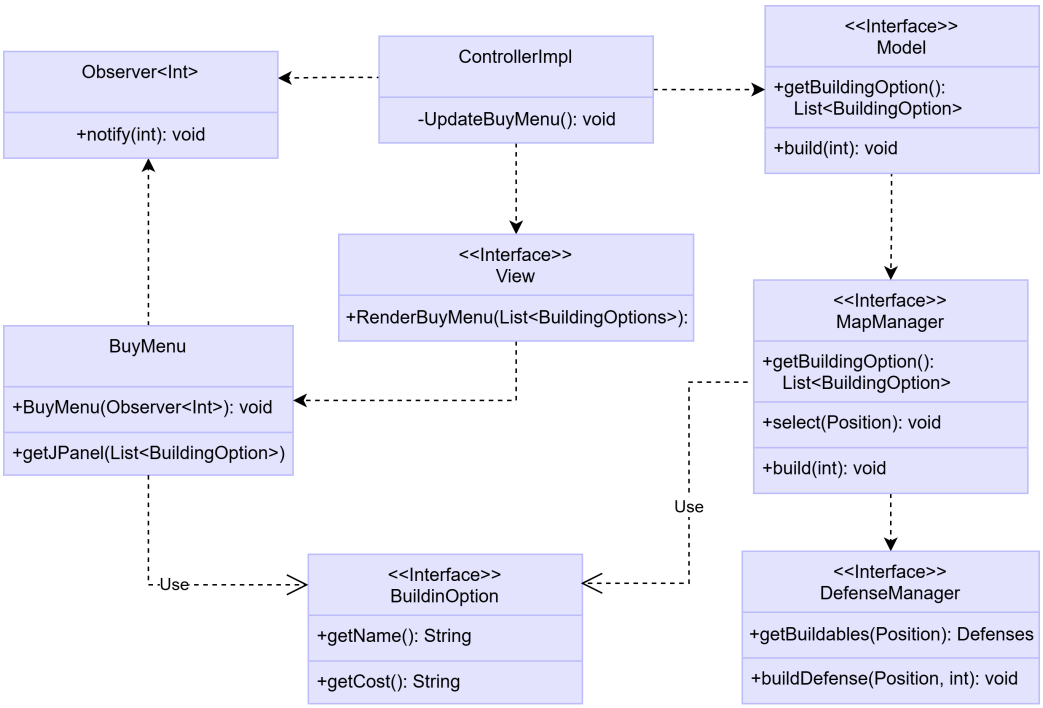

Figura 2.13: UML

Problema Comparsa asincrona di un menù con bottoni per permettere all'utente di scegliere la torre da costruire tra le possibili opzioni relative alla cella selezionata.

Soluzione La cella selezionata è memorizzata nel MapManager il quale richiede le torri edificabili in quel punto al DefenseManager e le converte nel DTO BuildingOption per poterle esporre fuori dal modello. Il Controller aggiorna il menù ogni volta che si modifica la cella selezionata, che viene costruita una torre oppure che cambia il GameState (ad esempio le monete diventano abbastanza per abilitare un pulsante), chiamando RenderBuyMenu sulla View e richiedendo le opzioni al Model. Per permettere che il segnale proveniente dai pulsanti della GUI possa raggiungere il Controller ho utilizzato un Observer che viene conferito al BuyMenu con l'inizializzazione, e attiva, attraverso il Controller, la chiamata Build sul Model.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per la verifica del corretto funzionamento dell'applicazione sono stati creati test automatizzati usando JUnit. I test realizzati mirano a garantire il corretto funzionamento del Model e delle classi di utility.

- **TestSkipIterator:** Testing automatizzato del corretto funzionamento della classe SkipIterator, si verifica che l'iteratore salti il numero specificato di elementi tra un Optional pieno e l'altro e che si comporti correttamente quando raggiunge la fine della sequenza.
- **TestFileUtils:** Testing automatizzato delle funzionalità di utility relative ai file (creazione cartelle, scrittura e lettura da file, lettura stream, lettura resource).
- **TestImageLoader:** Testing automatizzato del corretto funzionamento della classe ImageLoader, verificando la gestione di argomenti non validi, il caricamento di immagini inesistenti e il corretto scaling di immagini di diversi aspect rateo in base alle dimensioni e ai fattori di scala specificati.
- **TestDefenseFactoryImpl:** Testing automatizzato dei metodi di creazione delle difese e del salvataggio in formato json per la classe DefenseFactoryImpl. Si utilizzano i getters per testare la correttezza della generazione, che avviene tramite alcuni file json di testing.
- **TestDefenseImpl:** Testing automatizzato delle funzioni di DefenseImpl (getters, loading e salvataggio in formato json, setters). Vengono usati alcuni file json di test e si verifica la correttezza dei valori restituiti dai getters.

- **TestDefenseManagerImpl:** Piccolo insieme di test molto basilari che si assicurino la corretta esecuzione della classe DefenseManagerImpl senza il throw di eccezioni. Ci si assicura principalmente la correttezza del numero di difese costruite.

- **TestEnemyChoiceStrategyFactoryImpl:** Testing automatizzato dei metodi di EnemyChoiceStrategyFactoryImpl, si verifica il corretto funzionamento fornendo un campione prova di nemici da attaccare, e ci si assicura che la mappatura del danno corrisponda a dei risultati prestabiliti.

- **TestEnemiesImpl:** Testing automatizzato del comportamento di EnemiesImpl, si verifica la corretta inizializzazione, la gestione degli aggiornamenti, la creazione e le dinamiche delle ondate di nemici nonché l'aggiornamento della collezione di nemici in seguito alla loro morte e la gestione coerente dello stato dell'ondata.

- **TestEnemiesManagerImpl:** Testing automatizzato del binding di EnemiesManagerImpl. I test assicurano che i metodi di EnemiesManagerImpl lancino eccezioni se chiamati prima del bind e che lo stesso non possa essere effettuato più di una volta.

- **TestEnemyCatalogueFactory:** Testing automatizzato della classe EnemyCatalogueFactory, verifica la capacità di caricare configurazioni da file ben formattati e di lanciare eccezioni per file mal formattati. I test verificano anche che il catalogo contenga tutti i tipi di nemici e che il metodo di selezione basato su predicati funzioni correttamente.

- **TestEnemyCollectionImpl:** Testing automatizzato della classe EnemyCollectionImpl. Si verifica il corretto comportamento durante le varie fasi della vita di un nemico. Un osservatore dummy viene aggiunto alla collezione degli observer della morte dei nemici e si verifica il corretto funzionamento del sistema.

- **TestPredicateBasedRandomWaveGenerator:** Testing della generazione automatica di ondate di nemici. Verifica che le ondate generate siano coerenti alle politiche fornite.

- **TestSimpleEnemyFactory:** Testing automatizzato della Factory di nemici. Verifica che i nemici creati rispettino correttamente le specifiche iniziali ed espongano il comportamento atteso.

- **TestWavePolicySupplierImpl:** Testing del caricamento corretto della configurazione delle politiche delle ondate da file. Verifica che il caricamento avvenga correttamente da file ben formattati e che un'eccezione sia lanciata in caso contrario. Include test specifici che verificano la corretta estrazione delle informazioni da un file di configurazione scritto appositamente.
- **TestGameManagerImpl:** Testing automatizzato delle funzionalità di modello relative al GameManager (aumento e decremento monete, diminuzione vite, inizio wave, test di serializzazione e deserializzazione a json).
- **TestCell:** Testing automatizzato dei metodi della classe astratta per le celle.
- **TestMap:** Testing automatizzato della generazione e serializzabilità della mappa, e del metodo che fornisce la posizione in cui muoversi ai nemici.
- **TestSavesImpl:** Testing automatizzato delle funzionalità di caricamento dei salvataggi e scrittura su disco di un nuovo salvataggio.
- **TestScoreboardImpl:** Testing automatizzato delle funzionalità di caricamento della Scoreboard e salvataggio degli Score, insieme alla verifica del corretto formato del file scoreboard.
- **TestCollisionBoxImpl:** Testing automatizzato della funzionalità di collisione tra oggetti.
- **TestPositionImpl:** Testing automatizzato delle funzionalità relative alle operazioni eseguibili sull'oggetto posizione (aggiunta, sottrazione, distanza euclidea, serializzazione).
- **TestSizeImpl:** Testing automatizzato della classe Size.
- **TestVector2DImpl:** Testing automatizzato delle funzionalità relative alle operazioni eseguibili sull'oggetto vettore (aggiunta, sottrazione, moltiplicazione, prodotto scalare, direzione).

3.2 Note di sviluppo

Giacomo Arienti

Uso di Stream

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/model/saves/SaveImpl.java#L136-L143

Uso di Optional

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/model/game/GameManagerImpl.java#L339-L376

Uso di generici

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/commons/api/Copyable.java#L7-L13

Uso di Wildcard

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/view/graphics/Canvas.java#L29

Uso di libreria di terze parti (Google Guava)

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/commons/dtos/game/GameDTOImpl.java#L172

Uso di libreria di terze parti (Json)

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/commons/dtos/game/GameDTOImpl.java#L127-L133

Giacomo Boschi

Uso di Stream

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/model/defenses/EnemyChoiceStrategyFactoryImpl.java#L64-L71

Uso di Optional

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/model/defenses/Defense.java#L61

Uso di interfacce funzionali e lambda expressions

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/model/defenses/EnemyChoiceStrategyFactoryImpl.java#L27-L56

Uso di WildCard

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/model/defenses/DefenseManagerImpl.java#L182

Uso di librerie di terze parti (Json)

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/model/defenses/DefenseImpl.java#L223-L252

Davide Fiocchi

Uso di generici

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/5169831fe5cebf4c8f0348abf25717ee21e142dc/src/main/java/it/unibo/towerdefense/model/enemies/SimpleEnemyFactory.java#L138

Uso di stream

Utilizzati pervasivamente. Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/view/enemies/EnemyRendererImpl.java#L37

Uso di librerie di terze parti (org.imgscalr)

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/commons/utils/images/ImageLoader.java#L49

Uso di librerie di terze parti (Json)

Usati anche in un altra classe in modo molto simile: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/model/enemies/WavePolicySupplierImpl.java#L71

Uso di librerie di terze parti (Apache Commons)

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/5169831fe5cebf4c8f0348abf25717ee21e142dc/src/main/java/it/unibo/towerdefense/model/enemies/WavePolicySupplierImpl.java#L88

Uso di lambda expressions

Utilizzate pervasivamente, un esempio: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/model/enemies/EnemiesManagerImpl.java#L47

Pietro Pasini

Uso di Stream

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/model/map/ReversedPathFactory.java#L40C5-L71C6

Uso di librerie di terze parti (org.imgscalr)

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/view/map/MapRendererImpl.java#L61C5-L65C6

Uso di librerie di terze parti (Json)

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/model/map/GameMapImpl.java#L135C5-L181C6

Uso di Stream e generici

Permalink: https://github.com/giacomoarienti/OOP23-TD/blob/fca3f6f9c546a3b659c3b1ee3956c12b80f076d8/src/main/java/it/unibo/towerdefense/model/map/GameMapImpl.java#L190C4-L195C6

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Giacomo Arienti

Avendo principalmente contribuito alla parte "architetturale" del progetto, vista la mia parte di Model non molto ampia, mi sono trovato a dover gestire problematiche relative alla interoperabilità delle varie componenti del progetto tra di loro. Mi sono reso conto di aver avuto difficoltà nel dover pensare a come queste parti dovessero comunicare tra di loro senza aver a disposizione la loro implementazione essendo abituato ad un approccio di tipo *code-first*. Inoltre è stata la mia prima esperienza di sviluppo di un applicativo desktop avendo sempre lavorato in ambito web, è stato quindi necessario trovare una nuova metodologia d'approccio non avendo a disposizione un framework sul quale sviluppare il progetto. Oltre alle critiche, nel complesso sono orgoglioso del risultato ottenuto, ho apprezzato l'opportunità di lavorare con un team che ha favorito un ambiente stimolante e produttivo, e di aver potuto mettere in pratica le conoscenze acquisite durante il corso.

Giacomo Boschi

Lavorando sullo sviluppo delle difese ho avuto come principale sfida la scomposizione in sotto-problemi riguardo a come operano le torri. Per quanto ritengo che l'attività svolta sia soddisfacente, specialmente le parti relative alle strategie di attacco, ritengo che ci sia spazio per migliorare le altre parti, con un po più di tempo a disposizione si può ridurre l'uso di classi statiche e fornire delle vere animazioni di attacco al gioco. Ritengo comunque di aver svolto un ottimo lavoro, e che il sistema messo in atto permetta l'aggiunta delle features nominate precedentemente abbastanza semplice.

Davide Fiocchi

Lavorare a questo progetto è stata per me un'esperienza molto formativa. Da quello che ho potuto vedere, sono convinto che ognuno di noi (ma, ovviamente, parlo soprattutto per me stesso) abbia lavorato al meglio delle possibilità che la nostra inesistente esperienza ci ha concesso, sfidandosi costantemente a fare meglio e cercando di non scendere mai a compromessi. Questo non vuol dire che le difficoltà non ci siano state né che tornando indietro non cambierei nulla, anzi: nonostante siamo partiti con un progetto che ritenevamo ben pensato, alla prova dei fatti si è rivelato insoddisfacente e questo ha portato a dover rivedere diverse volte parti del codice già scritte. Queste difficoltà sono sicuramente state dettate dalla nostra scarsa esperienza sia con lavori di queste dimensioni sia con la diversa realtà del lavoro collettivo, ma ritengo che siamo stati in grado di adattarci rapidamente e in modo efficace al configurarsi di necessità che non avevamo previsto. Per quello che mi riguarda strettamente, non ho trovato particolari difficoltà nel riadattare il codice già scritto anche a fronte di cambiamenti della sovrastruttura che lo conteneva, e confido che questo possa essere segno di una buona strutturazione del codice da parte mia. Un aspetto (dovuto allo sviluppo incrementale descritto sopra) di cui mi pento è forse l'aver ripetuto più volte codice con comportamento molto simile (seppur non identico) che avrebbe potuto essere estratto e centralizzato. Anche correndo il rischio di essere accecato dall'"affetto" verso un progetto in cui ho (abbiamo) investito tanto impegno, mi ritengo tuttavia soddisfatto del lavoro compiuto e tutto sommato anche del risultato ottenuto.

Pietro Pasini

Questo progetto è stata la mia prima esperienza di programmazione di queste dimensioni. Rivedendo il risultato finale penso che molte delle parti che ho realizzato potevano essere progettate meglio dal punto di vista dell'architettura, nonostante abbia più volte modificato drasticamente la struttura della mia parte di modello. Questi tentativi sono risultati stressanti e dispendiosi di tempo, dunque mi pongo come obiettivo per un progetto futuro quello riflettere molto attentamente sulla scelta iniziale della struttura e cercare il più possibile di non cambiarla durante l'implementazione. Sono comunque contento di aver contribuito al progetto e ritengo che mi sia stato molto utile per consolidare e applicare i contenuti del corso, in particolare ho imparato molto seguendo anche lo sviluppo del codice dei miei colleghi che ho cercato di imitare in alcune strategie da loro adottate.

Appendice A

Guida utente

Launcher

All'avvio dell'applicazione si aprirà una piccola finestra in cui selezionare la risoluzione in cui verrà aperto il gioco. Verrà inoltre chiesto il nome giocatore, in modo da poter caricare successivamente i rispettivi salvataggi.

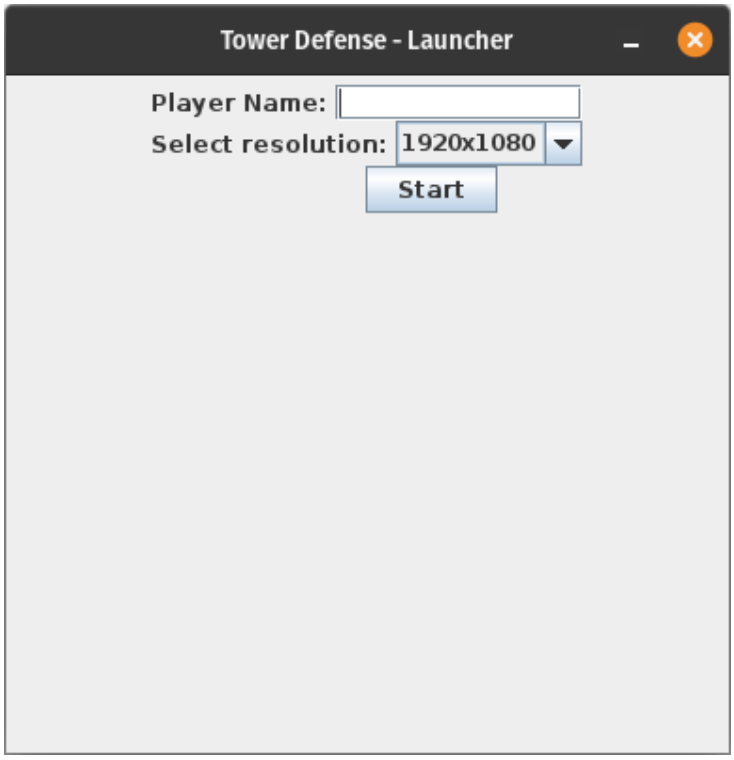

Figura A.1: Game Launcher

Menù

Avviando il gioco vero e proprio si visualizzerà un menù.
Premendo il tasto **Play** si inizierà una nuova partita su una mappa generata, che partirà immediatamente dalla wave 1 di nemici.
Scegliendo invece il pulsante **Load Game** si visualizzerà una finestra a scorrimento con l'elenco di tutti i salvataggi precedenti di quel giocatore, che potrà riprendere dall'inizio della wave in cui ha abbandonato. Infine sarà presente l'opzione **Scoreboard** per visualizzare i migliori dieci punteggi mai raggiunti e i giocatori in loro possesso.

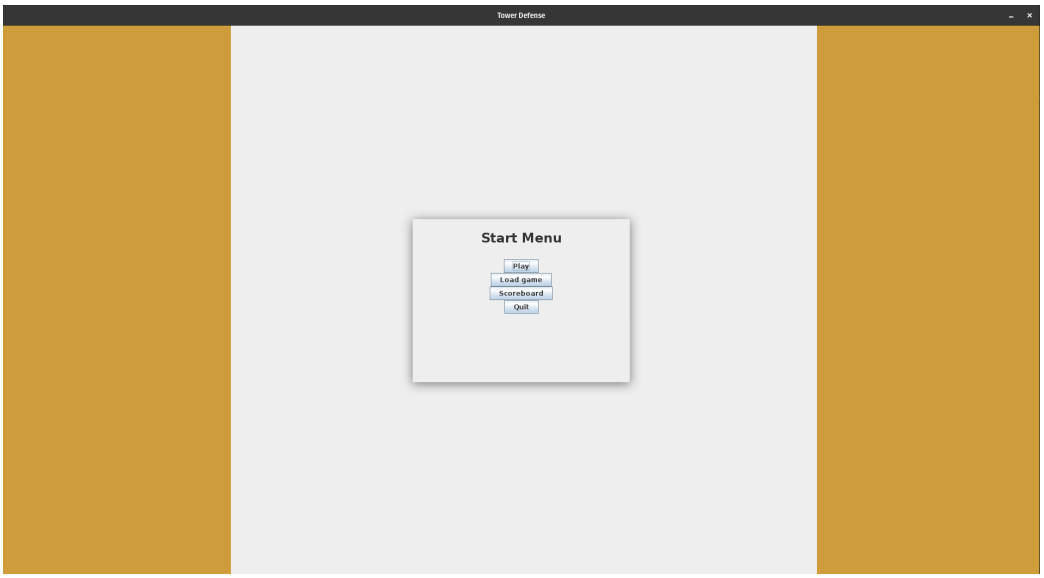

Figura A.2: Menu

Game

Iniziando una nuova partita bisognerà rapidamente costruire delle torri per eliminare i nemici. Per costruire una difesa bisogna selezionare una cella cliccandoci sopra (attenzione in presenza di alberi non sarà possibile selezionare). Con la selezione comparirà un menù in alto a destra con la lista delle possibili azioni, i pulsanti si abiliteranno solo se si avrà sufficiente denaro. Inoltre tenendo il cursore qualche secondo sul pulsante si potranno vedere le statistiche della torre che si andrà a costruire. Selezionando una cella con una torre costruita si vedrà un cerchio che rappresenta il raggio d'azione della torre (attenzione: leggere la descrizione per la ThunderInvoker), mentre nel menù ci saranno i potenziamenti per quella torre e il tasto per venderla.

Figura A.3: Game

Appendice B

Esercitazioni di laboratorio

giacomo.arienti2@studio.unibo.it

- Laboratorio 07: `https://virtuale.unibo.it/mod/forum/discuss.php?d=147598#p209320`
- Laboratorio 08: `https://virtuale.unibo.it/mod/forum/discuss.php?d=148025#p209770`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=149231#p211270`
- Laboratorio 10: `https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212762`
- Laboratorio 11: `https://virtuale.unibo.it/mod/forum/discuss.php?d=151542#p213960`