WRECK THE TOWER

Relazione del progetto per il corso di programmazione ad oggetti

Caguana Barrios Simone Andres 0000825544

A.A. 2018/2019

Indice

Capitolo 1

Analisi

1.1 Requisiti

Wreck The Tower è un videogioco della tipologia Tower Defense. Ci sono due giocatori: l'utente e il computer, ciascuno con una torre da difendere

Requisiti funzionali

- Ciascun giocatore dovrà difendere la propria torre dall'arrivo di unità nemiche della torre avversaria. Se nel braccio di ferro tra le unità una parte prevale sull'altra e distrugge la torre nemica, tale giocatore vince.
- Sono previsti 4 livelli unici, con difficoltà variabile in cui il computer userà sempre combinazioni di unità differenti. Inoltre il giocatore nel menu principale può selezionare il livello che vuole oppure può nel caso vincesse il livello, continuare per quello successivo o ripeterlo nel caso perdesse.
- Personalizzazione delle unità: il giocatore nel menu principale troverà una funzionalità che gli permetterà di personalizzare le proprie unità con le armi e le armature che preferisce.

Requisiti non funzionali

- Wreck The Tower deve essere portabile sui sistemi operativi Windows e Linux.

1.2 Analisi e modello del dominio

Gli elementi principali di cui ogni giocatore è disposto sono:

- una torre
- un insieme di "stampi di unità", i quali rappresentano l'equipaggiamento di un tipo di unità che il giocatore poi potrà impiegare in battaglia
- l'insieme di unità attualmente impiegate in battaglia.

Schema: Figura 1.1
Il giocatore per creare un'unità da uno dei suoi stampi dovrà pagare con dell'oro relativo al suo costo. Il nemico invece per creare le unità si basa su una strategia specifica per ogni livello che crea un insieme di stampi da usare direttamente (quindi senza l'utilizzo dell'oro).
Schema: Figura 1.2

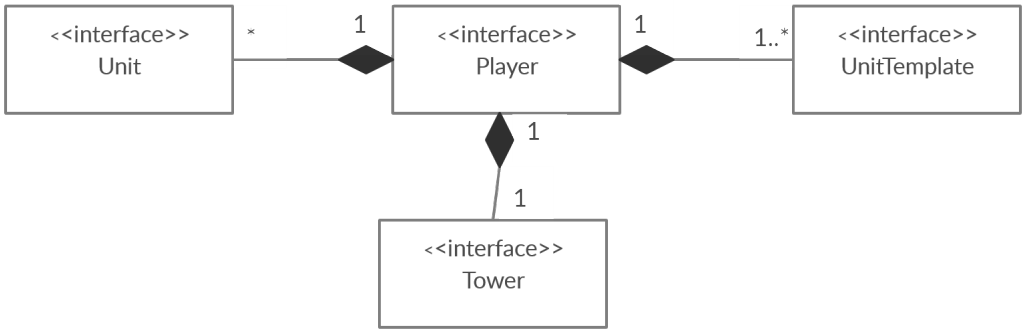

Figura 1.1: Schema UML dell'analisi del giocatore, con rappresentate le entità principali ed i rapporti fra loro

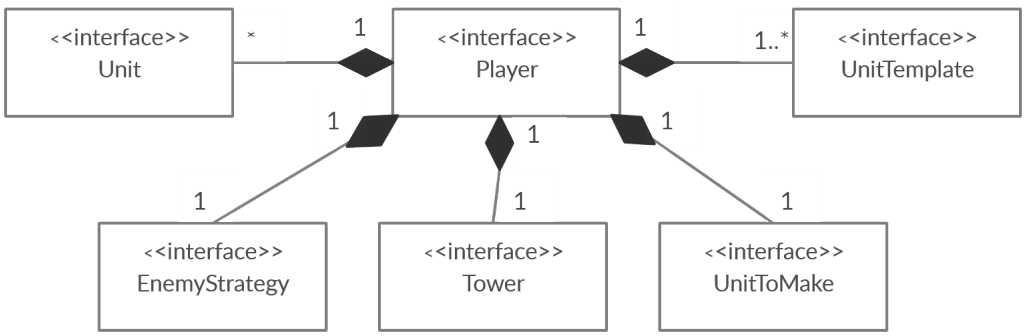

Figura 1.2: Schema UML dell'analisi del nemico, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Per la realizzazione di Wreck The Tower ho usato il pattern architetturale MVC (Model-View-Controller)

- **Model** racchiude i dati e la logica relativi al dominio dell'applicazione
- **View** gestisce l'interfaccia grafica e tutti i suoi elementi visivi e comunica i cambiamenti al controller
- **Controller** viene notificato dell'avvenire di eventi e comanda al model le modifiche e se necessario informa la view di applicare ulteriori cambiamenti

L'applicazione viene avviata da una classe Launcher che si occupa di istanziare la View e Controller.

La View (che sfrutta unicamente la libreria grafica JavaFX), è composta principalmente da:

- i file FXML generati con Scene Builder che rapppresentano lo scheletro dei menu e della schermata di gioco
- dalle classi che implementano Sprite
- dalla classe FrameUnitManager che contiene le liste di immagini che rappresenteranno le animazioni delle unità

Il Controller si occupa di gestire l'inizializzazione dei livelli, l'aggiornamento del livello massimo raggiunto e fa da tramite per le classi SceneController che lo usano per aggiornare i dati nel Model.

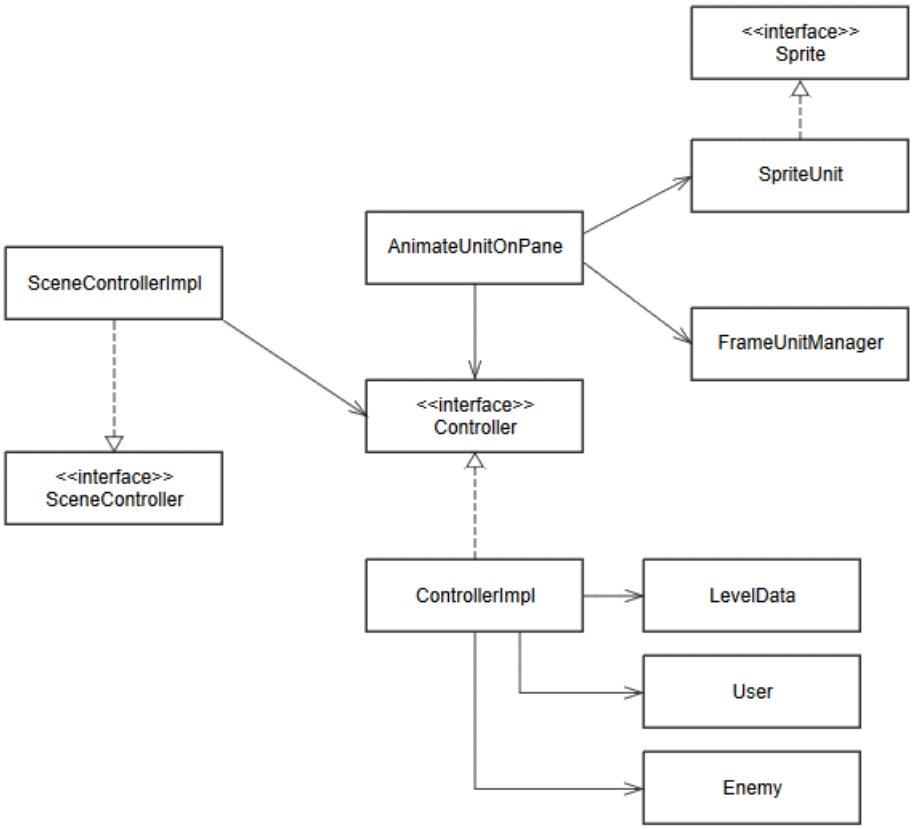

Figura 2.1: Schema UML dell'architettura MVC

2.2 Design dettagliato

2.2.1 Mosse del Nemico

La classe del Nemico sfrutta il design Pattern **Strategy** per eseguire le sue mosse durante una partita. L'interfaccia EnemyStrategy tramite il metodo enemyStrategyLoop va a pescare una serie di unità da creare, che non sono altro che dei riferimenti alla lista di stampi che il nemico ha in quel livello (ad esempio la mossa "normalAttack" se pescata potrebbe aggiungere alle unità da creare l'unitTemplate 1 e 2 della lista; mentre intenseAttack potrebbe aggiungere l'1 il 2 e il 4). Le unità pescate vengono aggiunte alla lista di unità da creare wrappata nella classe **"UnitToMake"** insieme a un double che ne rappresenta il **delay** (spiegato più avanti).
Le mosse sono scelte anche a seconda della vita rimasta alla torre (dal numero di unità perse ancora non è implementato).
Il metodo battleStrategy della classe Enemy viene richiamato dal controller

periodicamente con il delay dipendente da quanto questa mossa fosse forte (quest' ultimo viene inizializzato, ad ogni nuova mossa, proprio dalle classi che implementano la strategia).

L'utilizzo del pattern Strategy permette una notevole possiblità di personalizzare le mosse del nemico con svariati algoritmi e conseguentemente la facoltà di aumentare il numero di livelli dell'applicazione.

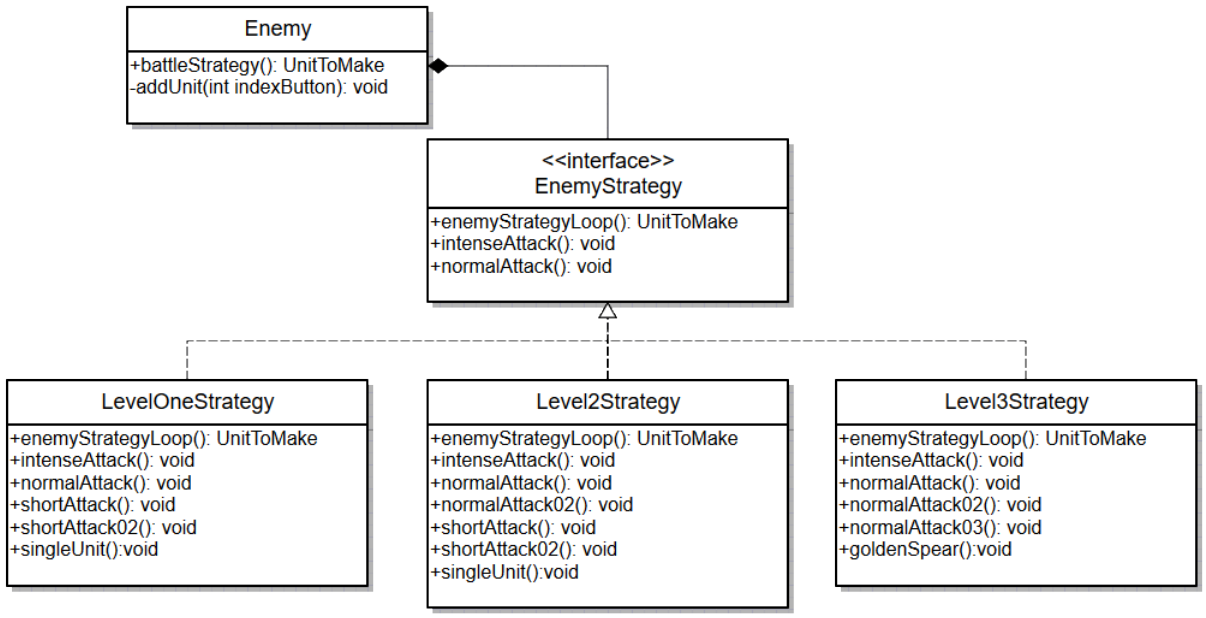

Figura 2.2: Schema UML del pattern Strategy per le mosse del nemico

2.2.2 Design Controller

Il controller principale è costituito dalla classe ControllerImpl che implementa l'interfaccia Controller. ControllerImpl possiede il metodo getLog **(implementazione del pattern singleton)**: ho deciso di utilizzare tale pattern perché la creazione di più istanze di oggetti di tipo controller avrebbe portato a problemi ossia il fatto che le diverse istanze avrebbero gestito il gioco parallelamente creando inconsistenza nei dati principali.

E' delegato al controller l'aggiornamento delle unità scelte dall'utente nel menu "Choose Units" e del livello massimo raggiunto. Ci tengo a precisare che tali informazioni non vengono perse tra una partita e l'altra o dalla chiusura dell'applicazione in quanto vengo scritte su dei file nelle risorse, che

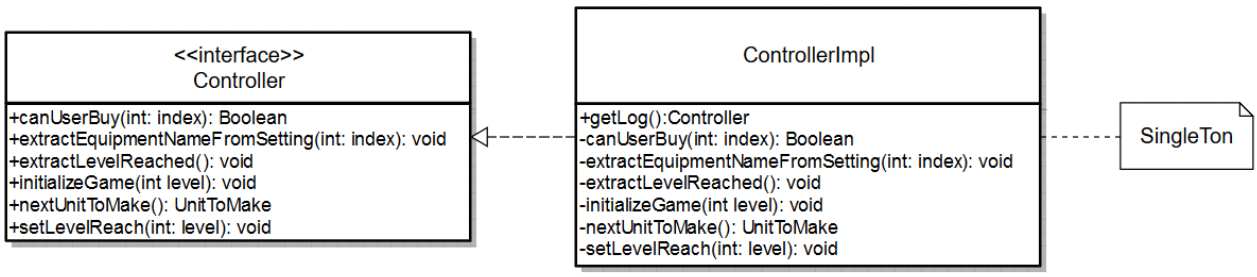

Figura 2.3: Schema UML del controller con implementazione del pattern Singleton

vengono utilizzati come "setting file".

SceneController:

SceneControllerImpl e tutte le sue sotto-classi implementano SceneController. Queste classi hanno un riferimento alla singola istanza del controller e possono usarla per aggiornare i dati nel model a fronte degli eventi di gioco.

- **MainMenuController:** menu principale che fa partire il gioco principale e/o la scelta delle unità
- **ChooseUnitsController:** menu di personalizzazione delle unità, si compone di un UnitChoiceMenu per ogni unità, una volta che l'utente ha finito, trascrive sul setting file le modifiche.
- UnitChoiceMenu: controller del container includente un menu a tendina nel quale selezionare l'equipaggiamento e una rappresentazione grafica della scelta tramite delle immagini
- **GameSceneController:** controller dello Stage principale di gioco includente dei container per mostrare la salute delle torri e l'oro, un container per la creazione di unità e un GameSpriteAnimation.
- GameSpriteAnimation: controller della Scene nella quale avviene il movimento e le interazioni tra le unità.

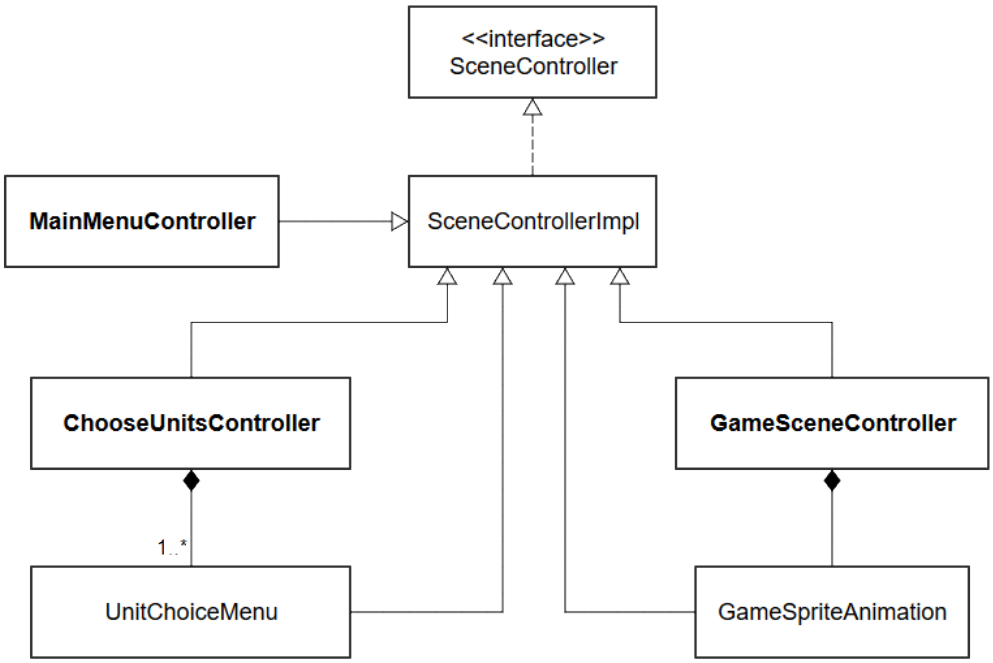

Figura 2.4: Schema UML dei controller delle Scene dell'applicazione (in grassetto quelle principali)

2.2.3 Frame manager

FrameUnitManager è una classe che contiene:

- Le liste di immagini che rappresentano i frame di animazione del corpo, dell'armatura e dell'arma dell'unità.
- I dati sulle hit-box del corpo e delle armi dell'unità, che essa avrà una volta "renderizzata" nel container del GameSpriteAnimation (permettono all'unità di capire quando attaccare, fermarsi o muoversi).

FrameUnitManager è utilizzata da **AnimateUnitOnPane** la quale viene istanziata per ogni unità e ne gestisce le interazioni e i movimenti, quindi usando opportunamente anche la lista di frame da mostrare.

Per gestire la grande varietà di frame a seconda della scelta di armi e armature nell'inizializzazione di FrameUnitManager ho utilizzato il pattern **Builder**. FrameUnitBuilder include totalmente l'algoritmo di costruzione dell'unità, la classe client non fa altro che chiamare il metodo build.

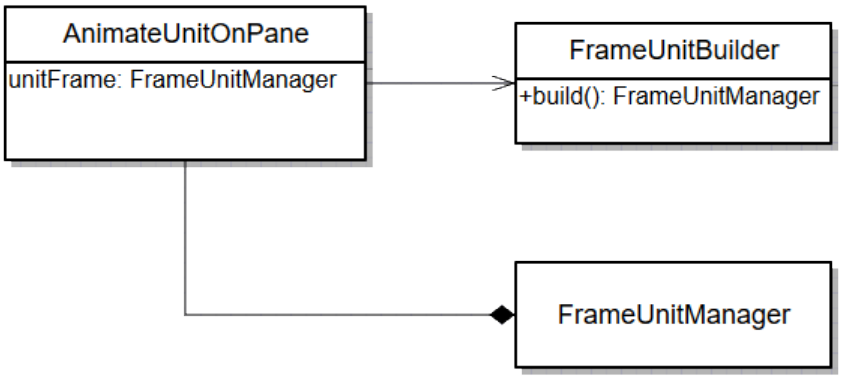

Figura 2.5: Schema UML del pattern Builder utilizzato

2.2.4 Sprite

Sprite è l'interfaccia comune a tutti gli elementi grafici interagibili nel gioco, SpriteImpl l'implementa e serve per rappresentare tutti gli elementi che non possono muoversi, SpriteUnit estende SpriteImpl e rappresenta le unità le quali possono essere animate. SpriteImpl e SpriteUnit sfruttano anch'essi il pattern **Builder** con una leggera modifica necessaria (Figura 2.6) :

Essendo che SpriteImpl sfrutta il sovracitato pattern, SpriteUnit che la estende dovrebbe poterlo usare anch'essa; Mentre si costruisce uno SpriteUnit con i metodi del costruttore ereditati si riscontra il problema che questi restituiscono uno SpriteImpl.Builder che non ha i metodi per i campi di SpriteUnit. Quindi tramite l'auto referenzialità data dalla classe astratta Init<T extends Init<T>> ogni Builder che la estende implementerà il metodo astratto ***self*** per restituire la corretta versione di se stesso (il giusto T che estende Init).

2.2.5 Sprite Animation

Il controller dei movimenti degli sprite su schermo e incluso l'attaccare e l'infliggere danno è gestito da AnimateUnitOnPane una superclasse astratta. Un oggetto del tipo AnimateUnitOnPane controlla una singola unità che implementa il pattern **Template Method**.

- Il metodo di template è executeSpriteStrategy dal quale partono tutti i metodi "start".

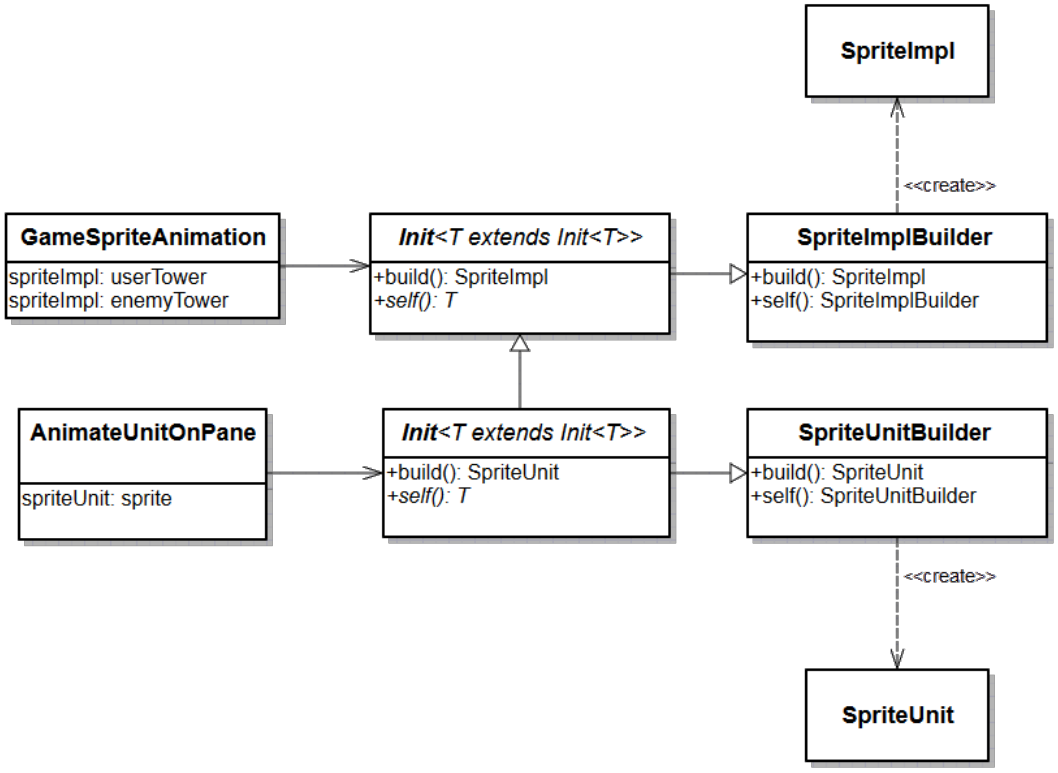

Figura 2.6: Schema UML del pattern Builder adattato alla condizione di eridetarietà di Sprite

- Tali metodi definiscono la strategia (l'algoritmo) di controllo d'animazione di diversi contesti (movimento, morte, attacco).
- AnimateMelee e AnimateRanged definiscono il passo degli algoritmi per quanto riguarda l'attacco in quanto una specializzazione rappresenta il combattimento ravvicinato e uno quello a distanza.
- Gli argomenti annotati con "many args" in Figura 2.7 , sono principalmente i parametri fondamentali di Model passati da GameSpriteAnimation con i quali si definiscono gli inneschi di certe animazioni. Da questi metodi nasce l'interazione fondamentale tra gli Sprite (View) e i dati delle Unità e del giocatore (Model).

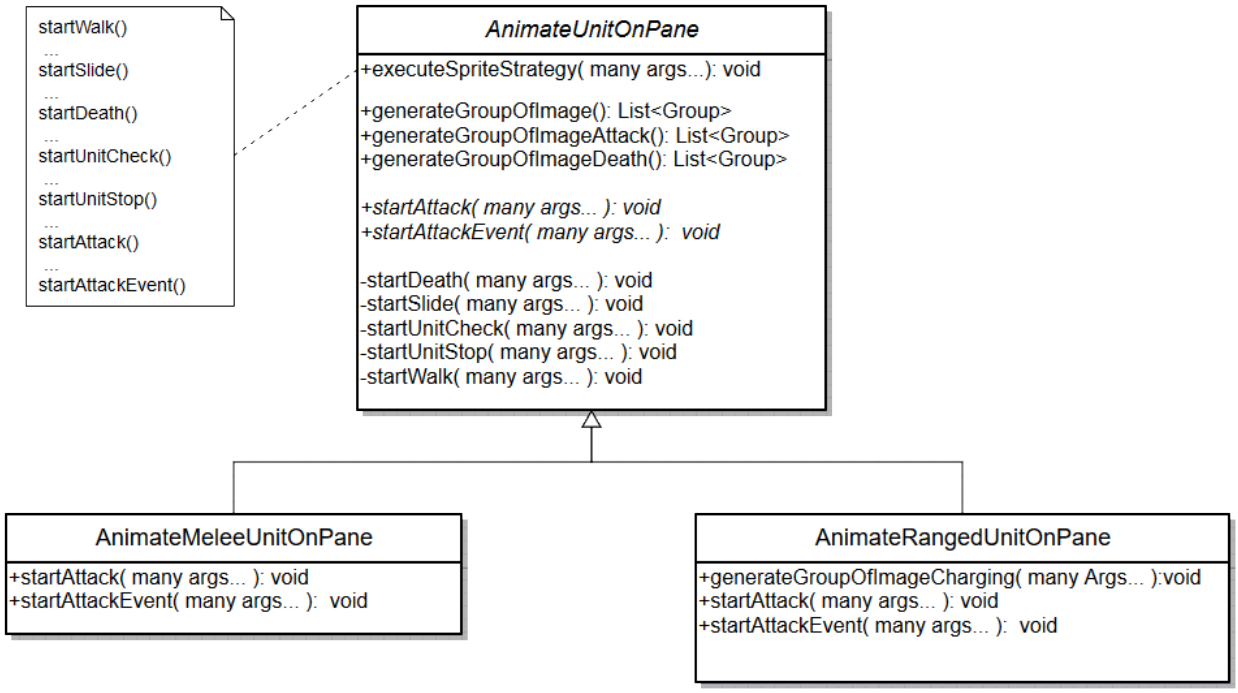

Figura 2.7: Schema UML del pattern Template Method usato per AnimateUnitOnPane

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per quanto riguarda il testing automatizzato non ne ho fatto grande utilizzo in quanto gli algoritmi più lunghi e complicati sono quelli del controller dell'animazione, invece le classi di Model come: Tower, Unit e Player contengono quasi solo dati semplici con getter e setter.

- Ho fatto un piccolo metodo di test nella classe WeaponTest per controllare che dato un tipo di arma, il suo nome restituito come Stringa sia corretto
- Una classe LevelOneStrategyTest che si assicura che a fronte di una determinata mossa della strategia il delay sia corretto

Per quanto riguarda il controllo dell'animazione e delle interazioni fra unità, ho testato l'applicazione manualmente:

- Il menù di scelta dell'unità salva correttamente le scelte del giocatore
- La schermata principale di gioco non genera alcun errore nel creare le unità, nel loro movimento, attacco e morte.
- Una volta completato una partita ferma tutte le animazioni, aggiorna il livello massimo se necessario e presenta dei pulsanti per continuare o tornare al menu principale correttamente.

3.2 Metodologia di lavoro

Al termine della fase di implementazione posso sottolineare che il lavoro è stato piuttosto lineare visto lo sviluppo individuale. Non ho riusato nulla

di ciò che i miei ex compagni di progetto hanno prodotto, in quanto, nel repository non vi era quasi nulla se non un paio di abbozzi di classi senza metodi e il restante lavoro che sarebbe dovuto essere caricato non è mai stato pervenuto. Mi sono avvalso del DVCS Git in modo da poter sempre consultare la catena di sviluppo e perchè faciliterà l'eventualità di versioni future dell'applicazione.

3.3 Note di sviluppo

Le feature avanzate che ho usato per lo sviluppo dell'applicazione sono:

- **Lambda** Nelle classi del package dello scenecontroller soprattutto per definire i Keyframe e aggiungerli a una Timeline in un'unica operazione. Nell'iterazione delle collezioni di dati con forEach.
- **Optional** Usati nel FrameUnitManger per le liste di immagini dell'arma e dell'armatura, in quanto è contemplata la possibilità di un'unità disarmata, nuda o entrambe.
- **Javafx** Ho scelto tale libreria per la creazione e gestione delle scene dell'applicazione
- **FXML** Usata per separare facilmente l'aspetto delle scene dalla loro logica. Per creare più facilmente questi file mi sono anche avvalso dello SceneBuilder.

Per apprendere bene il funzionamento di Javafx mi sono basato sulle slide del corso e sui progetti passati, tra cui, per lo più (Lego Minotauros). Per l'implementazione della classe sprite mi son informato cercando su Stack Overflow e ho adattato la soluzione al mio caso dove non ho una sola immagine ma anche immagini dell'equipaggiamento interscambiabili. Per implementare il pattern Builder con eredità ho consultato: `https://community.oracle.com/blogs/emcmanus/2010/10/24/using-builder-pattern-subclasses`

Ritengo molto interessante l'algoritmo che rappresenta le mosse del nemico nelle classi che estendono EnemyStrategy, in quanto può essere facilmente arricchito di nuove unità e nuove combinazioni di esse.
Inoltre ci tengo a sottolineare che gli algoritmi sfruttati per la gestione delle unità nella schermata di gioco, sfruttano il package animation di Javafx che grazie ai KeyFrame(classe che rappresenta un elemento di una Timeline) permettono di programmare la cadenza di certi eventi di gioco relativamente a dei parametri esterni, in particolar modo nella mia implementazione: ogni

arma ha la sua velocità d'attacco, ogni unità la sua velocità di movimento e ogni mossa del nemico ha un cooldown la cui applicazione viene implementata da un algoritmo ricorsivo che permette di creare una nuova timeline con un KeyFrame il cui cooldown è relativo all'attuale mossa.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Alla fine del progetto mi ritengo molto soddisfatto del risultato, mi sono messo alla prova con un numero notevole di idee e anche se riconosco che il lavoro svolto è ben maggiore di quello normalmente stabilito per una sola persona, ho voluto andare fino in fondo perchè ritengo di essermi arricchito di esperienza e aver fatto luce su alcuni meccanismi fondamentali della gestione degli elementi grafici di un videogioco. Penso che già nel prossimo futuro vorrò ampliare i contenuti e migliorarne la struttura.

Riconosco che nello strutturare la parte grafica ho abusato dei "numeri magici".
Ho dovuto forzare il meccanismo del MVC, cioè che nelle classi di animazione la distinzione tra model e view sia sfumata, ma essendo che la logica degli sprite è proprio quella di usare elementi di view (Rectangle2D in javafx) per definire le collisioni (quando uno sprite tocca un altro $->$ metodo intersect) ho pensato fosse immensamente più chiaro, semplice ed esplicativo, che gestire la stessa cosa tramite il model con altri thread utilizzando degli interi per rappresentare i vertici dell'hitbox nello schermo e gestirne con il controller la collisione.

Penso che il progetto come videogioco sia potenzialmente molto divertente e personalizzabile grazie alla meccanica di composizione delle unità con diversi equipaggiamenti e la facilità di aggiunta di armi, armature e livelli.

Bibliografia

[1] Design patterns di Gamma, Helm, Johnson, Vlissides