Relazione
“Time2Survive”

Andrea Cecchini
Nicolò D’Addabbo
Luca Oskari Fiumanò
Leonardo David Matteo Magnani

10 febbraio 2023

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il progetto si pone l'obiettivo di creare un videogioco survival shoot 'em up ispirato a "The Binding of Isaac" [1]. Con survival shoot 'em up si intende un genere di videogioco nella quale l'obiettivo è sopravvivere il più a lungo possibile a ondate di nemici, generandone di nuove una volta eliminata la precedente.

Requisiti funzionali

Requisiti Funzionali Obbligatori

- Il gioco dovrà occuparsi di gestire la partita dell'utente all'interno della mappa di gioco di Time2Survive, rendendo più difficile la sopravvivenza del giocatore attraverso ondate di nemici di difficoltà crescente. All'interno della mappa, il giocatore potrà:
  - Sfruttare le proprie abilità di videogiocatore per sopravvivere alle numerose ondate dei nemici che si susseguiranno.
  - Sfidare ondate che verranno generate al completamento della precedente. Le ondate terminano quando tutti i nemici sono stati eliminati.
  - I nemici si potranno eliminare attraverso l'utilizzo di proiettili generati dal personaggio.
  - La partita sarà conclusa alla morte virtuale del giocatore.

[1] `https://store.steampowered.com/app/113200/The_Binding_of_Isaac/`

- Il giocatore si troverà all'interno di una stanza in cui verranno generate le ondate di nemici.
- La difficoltà delle ondate verrà manipolata attraverso tre parametri:
  - Numero: numero dei nemici a schermo.
  - Tipo: i nemici avranno tipi diversi con caratteristiche uniche che renderanno l'esperienza di gioco più variabile.
  - Boss: dopo un certo numero di round verrà generata un ondata speciale formata da un Boss molto più potente rispetto ai soliti nemici.
- I nemici sono delle entità attive del gioco che cercano di eliminare il giocatore. Vengono controllati da una semplice AI (Artificial Intelligence). Per intelligenza artificiale si intende un software che conferisce a ogni nemico una strategia di movimento e di combattimento.
- Il giocatore si potrà muovere nell'ambiente di gioco costituito da una singola stanza.
- Il giocatore dovrà essere in grado di capire lo stato della partita in ogni momento della suddetta visualizzando la propria vita. Il giocatore sarà inoltre in grado di visualizzare le proprie capacità essendo in grado di visualizzare il numero di ondate alle quali è sopravvissuto.

Requisiti Funzionali Opzionali

- Per aiutare il giocatore, saranno disponibili diversi potenziamenti che gli permetteranno di aumentare le proprie abilità.

Requisiti non funzionali

- Il programma deve mantenere un profilo di prestazione adeguato e costante nonostante il variare dello stato del mondo di gioco alla fine di garantire un'esperienza di gioco fluida e gradevole.
- Per rendere il videogioco il più godibile possibile la grafica dovrà essere elaborata con uno stile grafico al passo coi tempi.

1.2 Analisi e modello del dominio

T2S mette a disposizione del giocatore la possibilità di giocare una nuova partita, che gestirà lo stato e il mondo di gioco.
Il mondo di T2S è l'arena dove le diverse entità interagiscono fra di loro.
Al giocatore sarà data la possibilità di controllare una di queste entità che rappresenterà il giocatore all'interno della partita.
Durante lo svolgimento della partita verranno generate delle ondate di entità controllate da un'intelligenza artificiale con il compito di attaccare e uccidere il giocatore. Per combattere questi nemici, il giocatore avrà la possibilità di sparare dei proiettili e di ottenere diversi tipi di power up, durante round prestabiliti e continui, che renderanno il più semplice sconfiggere tutti i nemici che gli si porranno avanti. Una volta sconfitta un'ondata viene aggiornato lo stato della partita incrementando il numero di round alla quale si è sopravvissuti e generandone una nuova di maggiore difficoltà. All'inizio della partita al giocatore verranno assegnati dei proiettili di default e un determinato numero di punti vita. Una volta subito un danno, a seconda della sua entità, verrà detratta una certa quantità dalla vita corrente, una volta persa tutta il gioco terminerà mostrando al giocatore per quanto è sopravvissuto.
Le difficoltà che il progetto si porta dietro sono:

- Un'implementazione di un intelligenza artificiale in grado di controllare correttamente i nemici. La prima versione del gioco fornirà ai nemici un Controller basato sulle coordinate del player nella mappa.
- Una grafica completa che verrà implementata inizialmente in maniera minimale.

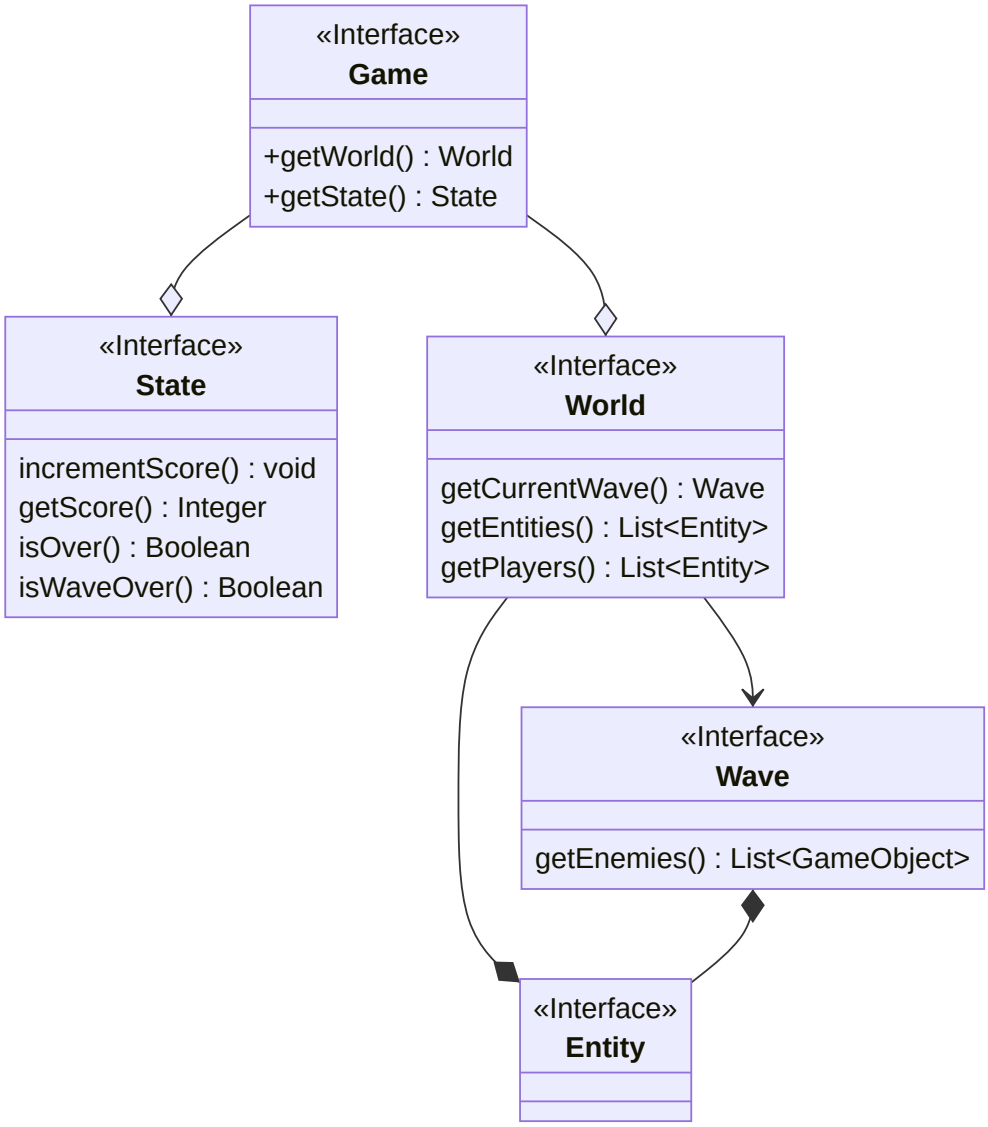

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

T2S è stato sviluppato adottando un pattern architetturale che mantenesse una simil struttura MVC, quindi con i suoi vantaggi, contaminandola però attraverso l'utilizzo del pattern Component.
Il pattern Component permette a una singola entità di estendersi su domini multipli senza accorparli in un unica classe.
Applicandolo al dominio dell'applicazione (Model), avremo che ogni elemento del gioco risulta essere un entità. Tale entità verrà adornata da dei componenti, ognuno dei quali appartiene a un dominio particolare (Input, Fisica, Grafica, ...), che sarà responsabile dell'aggiornamento del dominio all'interno dell'Entity. Tuttavia, pur appartenendo a domini diversi, i componenti potrebbero aver bisogno di comunicare tra di loro. Sfruttando come "container" l'entità stesso è stato messo a disposizione ai componenti un semplice ma funzionale sistema di messaggistica. L'utilizzo di questi componenti porta ad importanti vantaggi:

- **Stroncamento di un albero di ereditarietà troppo complesso e lungo**: Per rappresentare entità con diversi comportamenti si utilizza la composizione di componenti che implementino quei comportamenti, senza nessun utilizzo dell'ereditarietà.
- **Eliminazione dei problemi legati all'ereditarietà multipla**
- **Riduzione al minimo delle ripetizioni di codice** dovuto alla creazione di componenti altamente riutilizzabili

Dal punto di vista del pattern architetturale Model-View-Controller, i rispettivi moduli sono rappresentati da:

- **Model**: Il ruolo del "Model" viene affidato all'insieme delle classi che sono contenute nella classe Game (inclusa). Difatti, per creare un gioco diverso da T2S basterebbe cambiare la parte di "Model" dell'architettura in questione, le parti da sostituire ricadono nell'insieme sopra citato.

- **Controller**: Questo ruolo viene affidato all'interfaccia GameEngine, la quale ha il compito di gestire l'aggiornamento di ogni Entity, dei suoi componenti e della sua renderizzazione sulla View.

- **View**: Il ruolo della view viene demandato a delle interfacce ben connesse tra di loro: GameScene e Graphics. L'interfaccia GameScene astrae il concetto di "Scena" di gioco
  L'interfaccia Graphics ha il compito di saper "disegnare" le diverse entità con la tecnologia grafica che si è scelto di utilizzare.

Dal diagramma delle classi, qui sotto fornito, è possibile notare un elemento di differenza rispetto a una classica architettura MVC: un componente di spicco, denominato GraphicsComponent, risulta avere dei riferimenti alla parte di view e di model, creando un collegamento che a prima vista potrebbe risultare problematico. La modularità di M.V.C. non viene però attaccata: volendo, come esempio, cambiare la tecnologia di sviluppo per quanto riguarda la grafica si modificherà esclusivamente la parte di View dell'applicazione, senza ripercussioni in nessun altro modulo del programma. L'ipotetico passaggio da JavaFX a Swing o ad altre tecnologie, dunque, non risulta essere problematico: si dovranno sostituire le implementazioni di JavaFX delle interfacce Graphics, GameScene con delle nuove implementazioni che utilizzino la nuova tecnologia.

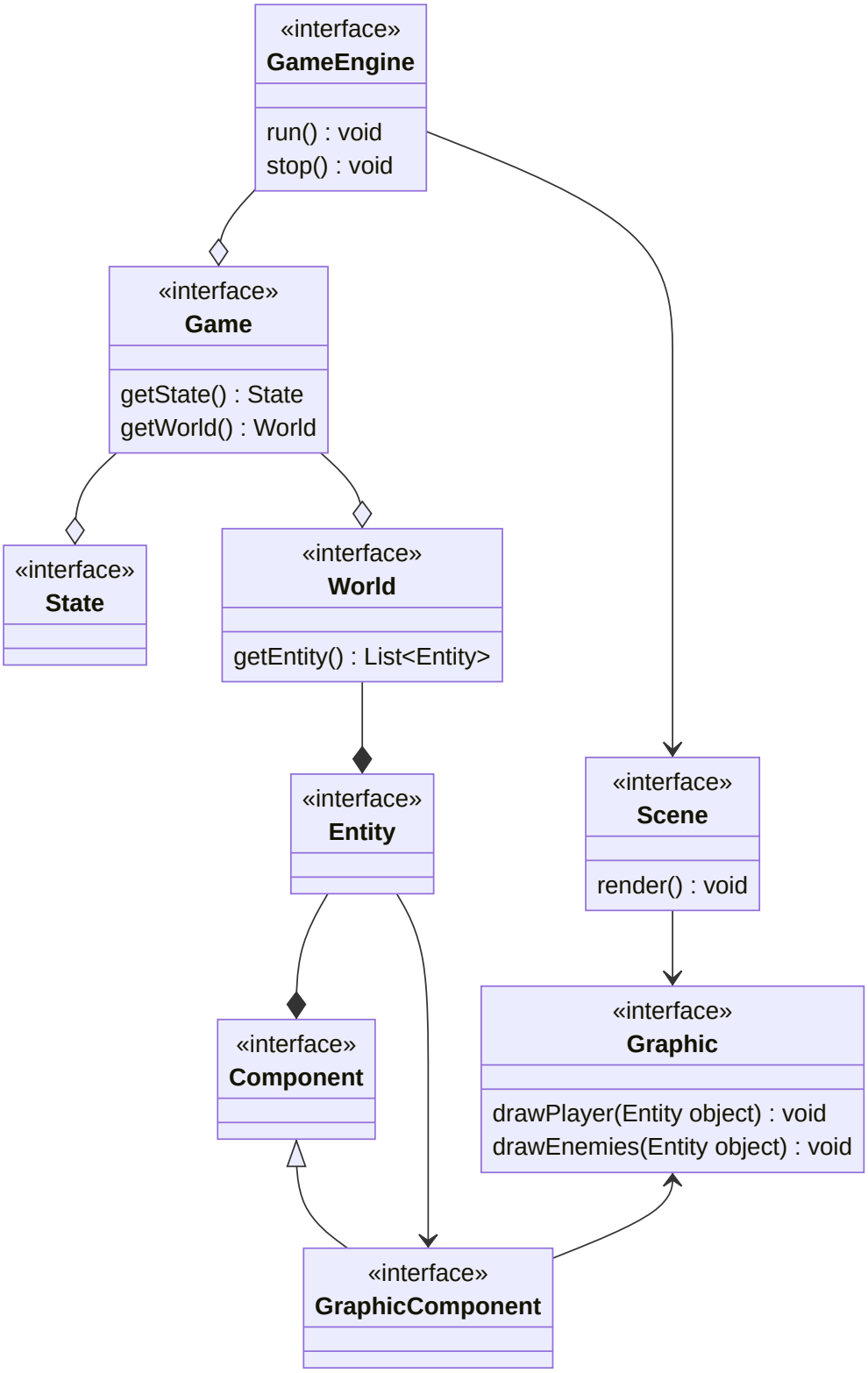

Figura 2.1: Schema UML architetturale di T2S.

2.2 Design dettagliato

Andrea Cecchini

Separare il dominio del GameLoop da quello del GameEngine

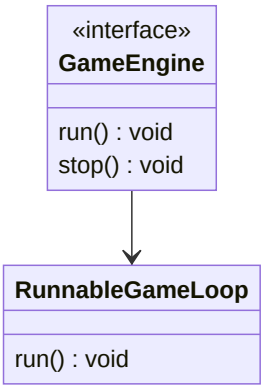

Figura 2.2: Rappresentazione UML del pattern Strategy per separare il dominio della classe GameLoop da quello del GameEngine.

Problema Disacoppiare la logica della classe **GameLoop** dal logica della classe **GameEngine**, al fine di garantire il **Single Responsability Principle**.

Soluzione La classe GameEngine utilizza il pattern **Strategy** al fine di delegare alla classe **RunnableGameLoop** la gestione del dominio del GameLoop. Quindi, la classe GameEngine avrà la responsabilità di gestire solamente ciò che le compete, delegando il resto.

Decorazione della classe GameLoop

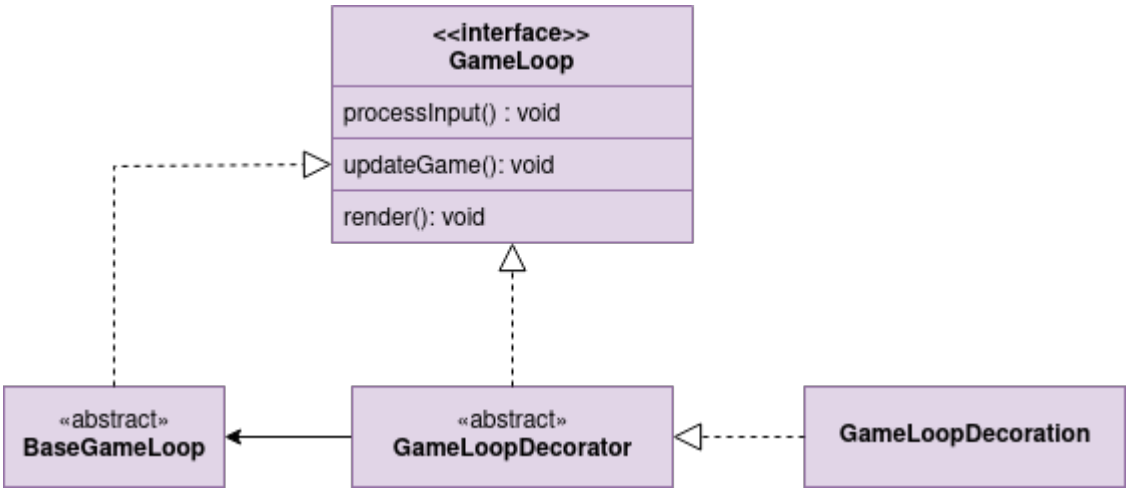

Figura 2.3: Rappresentazione UML del Decorator pattern per il GameLoop

Problema La classe **GameLoop** risulta essere prona alla confusione e alla non manutibilità. Questo è dovuto alla natura stessa del GameLoop, nel quale al suo interno si sviluppano diversi domini e logiche quali:

- Processione dell'input.
- Aggiornamento dello stato di gioco
- Gestione della renderizzazione.

Inoltre, ognuno di questi domini potrebbe raccogliere delle particolarità che possono differire dalla implementazione basica contenuta in **BaseGameLoop**. Aggiungere nuove feature risulta quindi essere un lavoro poco gradevole e prono alla aggiunti di possibili errori.

Soluzione Utilizzare il pattern **Decorator**, nel quale il BaseGameLoop viene decorato con delle possibili implementazioni di decorazione, permette di risolvere tutti i problemi sopra presentati:
In particolare il codice non risulta più essere accumulato in un punto specifico della classe BaseGameLoop e l'aggiunta di nuove feature, riguardante i domini sopra citati, risulta essere estremamente facile, difatti basta aggiungere una nuova decorazione al game loop già sviluppato.
Il codice risulta essere estremamente manutenibile, questo dovuto alla separazione di codice che il pattern Decorator offre.

Implementare diversi tipi di basi per il game loop

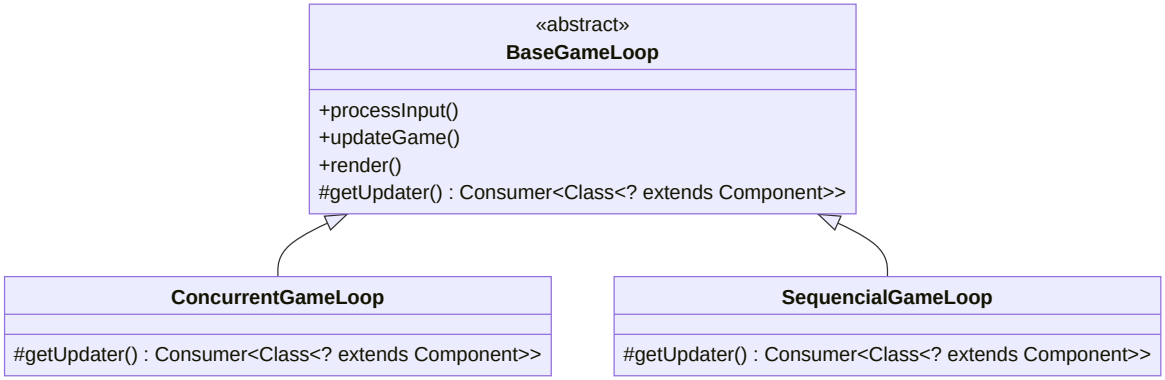

Figura 2.4: Rappresentazione UML del pattern Template Method per la creazione di GameLoops Base

Problema Esistono due versioni di BaseGameLoop, una nel quale i componenti vengono aggiornati parallelamente e un'altra sequenzialmente. Ci si è accorti che le due implementazioni condividevano gran parte del codice, avendo quindi una ripetizione.

Soluzione Essendo che le due implementazioni differiscono solo per il tipo di updater, è stato utilizzato il **Template Method** al fine di massimizzare il riuso. I metodi template sono `processInput()`, `updateGame()`, `render()`, che chiamano un metodo astratto e protetto `getUpdater()`.

Gestione delle collisioni fra figure di forma diverse

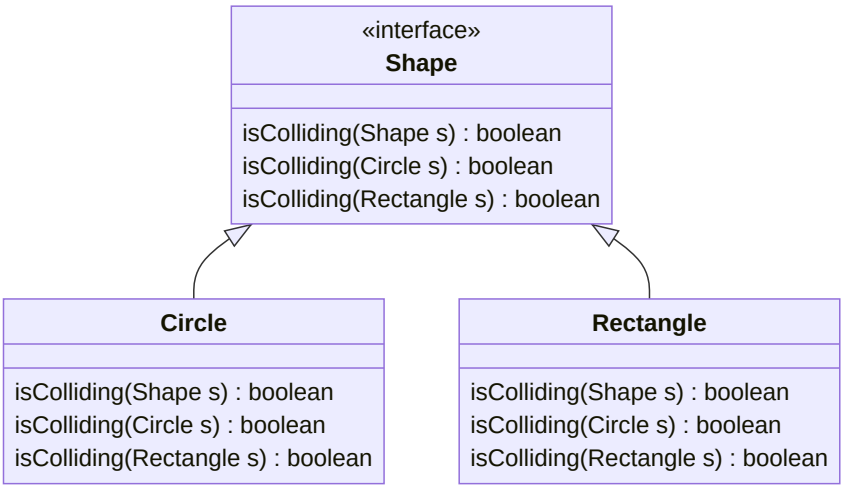

Figura 2.5: Rappresentazione UML del pattern Visitor Pattern per la gestione delle collisioni con figure di forma diversa.

Problema L'istanza della classe **Shape** ha bisogno di sapere l'implementazione della figura della quale sta verificando la possibile collisione, al fine di addattare i propri calcoli verificare se vi è stata o meno la collisione con quella figura.

Soluzione La soluzione scelta rispetta l'implementazione del pattern **Visitor**. Tale implementazione suggerisce di delegare alla Shape **s** il controllo della collisione con la Shape **this**. Questa soluzione rende possibile l'aggiunta di nuove figure in modo facile, senza dover creare un costrutto switch o if-else al fine di controllare la reale implementazione della shape s.

Nicolò D'Addabbo

Input Controller intercambiabili

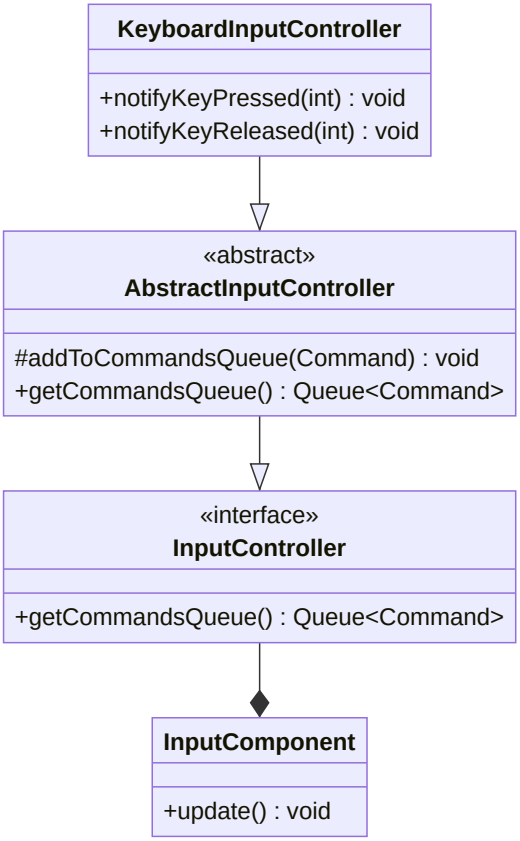

Figura 2.6: Rappresentazione UML del pattern Strategy per l'Input Component

Problema Un'entità può ricevere input da diverse sorgenti come ad esempio una tastiera o un joypad, la cui presenza non è di interesse dell'entità stessa. Infatti deve possedere solo un generico Input Component con un metodo update.

Soluzione La classe **InputComponent** implementa lo *Strategy pattern*, contenendo un riferimento a un *InputController* che funge da strategia per la gestione dell'input. Quest'ultimo può essere sostituito con un'implementazione diversa, permettendo l'utilizzo di strategie differenti.
Il metodo *update* dell'Input Component chiama il metodo *execute* su ogni

Command (come descritto nella sezione "Gestione dei comandi di gioco") presente nella coda di comandi restituita dall'*InputController* (come descritto nella sezione "Input Controller con molteplici stati").
Inoltre, grazie allo *Strategy pattern*, è possibile sviluppare un'intelligenza artificiale in modo semplice. Un'AI, infatti, non è altro che un *InputController* che genera diversi *Command* in base a un determinato algoritmo.

Gestione dei comandi di gioco

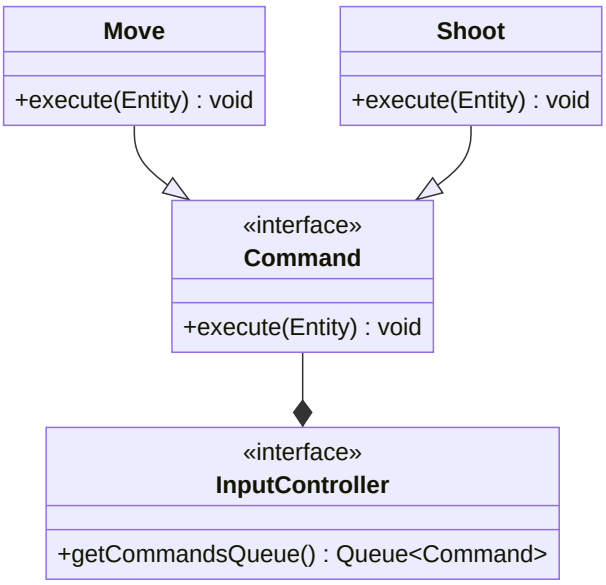

Figura 2.7: Rappresentazione UML del pattern Command per la gestione dei comandi

Problema Rendere riutilizzabili i comandi di gioco, separando la responsabilità di eseguirli dall'oggetto che li registra.

Soluzione Il sistema di gestione dei comandi di gioco utilizza il *Command pattern*. Questo pattern consente di incapsulare una richiesta, nel nostro caso un comando di input, in un oggetto. In questo modo il *Command pattern* fornisce un livello di astrazione tra gli oggetti che triggerano azioni e quelli che le eseguono, riducendo la dipendenza tra questi oggetti. La struttura del *Command pattern* è composta da:

- **Command**: l'interfaccia che definisce l'operazione da eseguire. In questo caso rappresentato da *Command*.
- **Concrete Command**: classe che implementa l'interfaccia Command e specifica la logica di esecuzione per una determinata richiesta. In questo caso rappresentato da *Move* e *Shoot*.
- **Receiver**: classe che contiene la logica vera e propria. I comandi gestiscono solo i dettagli di come una richiesta viene passata al *receiver,* è infatti il *receiver* che esegue il comando. In questo caso rappresentato dai diversi **Component**.
- **Invoker**: classe che inizializza la richiesta. In questo caso rappresentato da **InputController + InputComponent**.

Altri vantaggi dell'utilizzo del *Command pattern* sono la flessibilità e la riutilizzabilità. Infatti rende più facile l'aggiunta di nuovi comandi, la modifica di quelli esistenti o anche l'annullamento di comandi già eseguiti.

Riuso di Input Controller

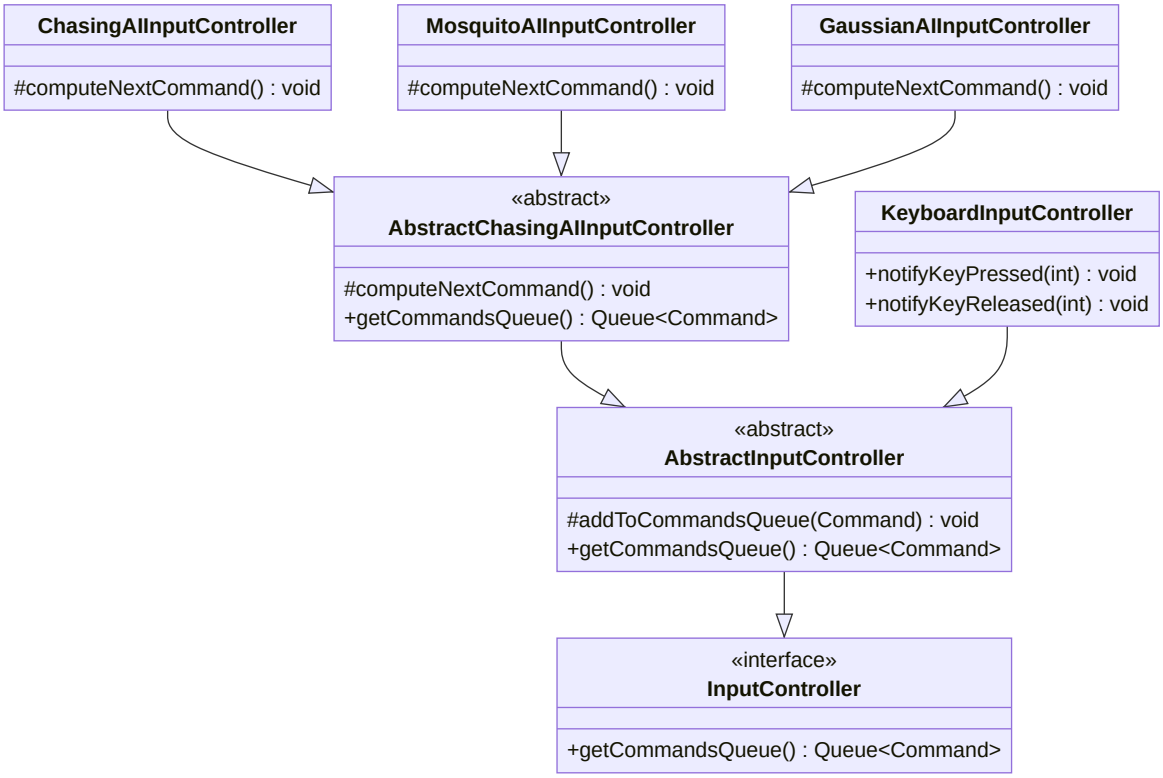

Figura 2.8: Rappresentazione UML del pattern Template Method per il riuso di Input Controller

Problema Rendere riutilizzabili i comandi di gioco, separando la responsabilità di eseguirli dall'oggetto che li registra.

Soluzione In *AbstractInputController* viene data un'implementazione di default al metodo *getCommandsQueue()*, da questa classe astratta si possono costruire Input Controller come *KeyboardInputController* che viene notificato ogni volta che un carattere da tastiera viene premuto o rilasciato. Dato che un AI Input Controller non viene notificato da nessuna periferica, deve generare lui stesso i comandi da inserire nella *commandsQueue*. Dato che diverse AI differiscono solamente per il metodo di generazione di comandi, è stato utilizzato il *Template Method pattern*.
Nella classe *AbstractChasingAIInputController* viene fatto l'override del metodo *getCommandsQueue()* rendendolo il *metodo di template* che, prima di

ritornare la Commands Queue, chiama il metodo astratto *computeNextCommand()*, il quale aggiunge in coda un comando generato seguendo l'algoritmo della data AI.

Input Controller con molteplici stati

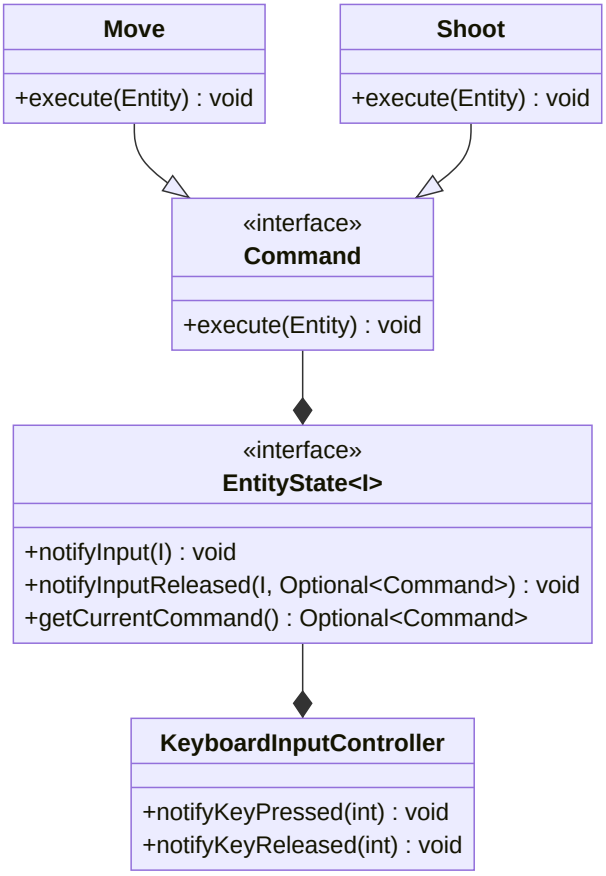

Figura 2.9: Rappresentazione UML del pattern State per la creazione di Input Controller con più FSM

Problema Rendere riutilizzabili i comandi di gioco, separando la responsabilità di eseguirli dall'oggetto che li registra.

Soluzione Un player al momento presenta due *macchine a stati finiti* (FSM): *moveState* e *shotState*. Ad ogni momento c'è un numero finito di stati in cui *moveState* può stare: UP, DOWN, LEFT, RIGHT e STAY. Ad ognuno di

questi stati corrisponde un comando (comportamento) diverso. *moveState* può passare da uno stato ad un altro tramite regole dette *transitions*, in questo caso le *transitions* sono definite da un *moveset* dove ad ogni input da tastiera corrisponde un comando. L'implementazione di questa FSM (e similmente anche per *shotState*) è stata fatta utilizzando lo *State pattern*, dove le implementazioni degli stati *(Concrete States)* sono oggetti della classe *Move* inizializzata con *Directions* differenti, mentre il *Context* è rappresentato da un implementazione di *EntityState*. Grazie a questo pattern viene ridotta notevolmete la ripetizione di codice e semplificata l'aggiunta di ulteriori stati. L'azione "sparare" è rappresentabile in quattro stati: UP, DOWN, LEFT, RIGHT; corrispondenti alla direzione del proitettile. Se "sparare" venisse trattato nella stessa FSM in cui viene gestito il movimento, per poter eseguire queste due azioni contemporaneamente, dovremmo duplicare il numero di stati. Per risolvere questo problema vengono usate delle *Concurrent State Machines*, ovvero due FSM che operano in maniera indipendente, in questo caso *moveState* e *shotState*.

Gestione degli eventi di gioco

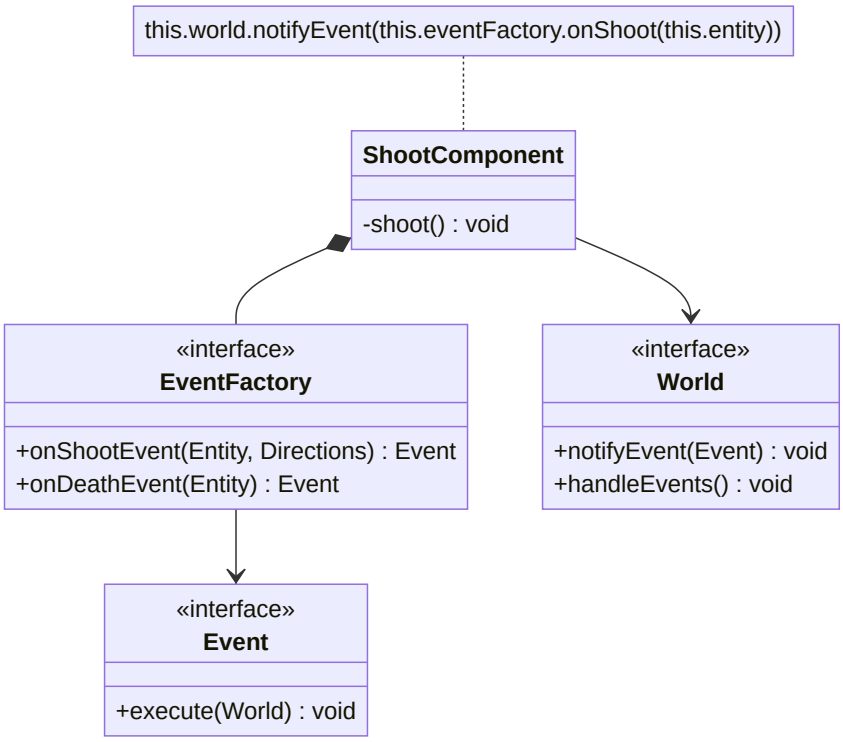

Figura 2.10: Rappresentazione UML della gestione degli eventi

Problema Dato che sono i Component a generare eventi, essi verrebbero generati quando viene chiamato il metodo *update* di un component, quindi all'interno di un ciclo, incorrendo nel rischio di eccezioni *"ConcurrentModificationException"*.

Soluzione Per disaccoppiare l'evento generato da quando viene eseguito, utilizziamo il *Pattern Event Queue*. Come descritto nel libro "Game Programming Patterns" di Robert Nystrom: *"Una coda memorizza una serie di notifiche o richieste in ordine FIFO (First-In First-Out). Inviare una notifica aggiunge la richiesta alla coda e ritorna. Il "Request Processor" elabora quindi gli elementi nella coda in un secondo momento. Le richieste possono essere gestite direttamente o inoltrate ad altri oggetti. Questo disaccoppia il mittente dal destinatario sia staticamente che nel tempo"* [1].

[1] `https://gameprogrammingpatterns.com/event-queue.html#the-pattern`

Prendiamo come esempio l'evento *onShoot*. La gestione degli eventi è analoga anche per *onPowerUp* e *onDeath*. Quando uno ShootComponent vuole generare un proiettile, chiama il metodo notifyEvent(Event) del *Request Processor* World, passandogli un evento come parametro. Questo evento viene aggiunto alla coda della Event Queue. Gli eventi possono essere generati tramite una *Factory*.
Al prossimo update del Game, verrà chiamato il metodo handleEvents() del World, che eseguirà ogni comando presente nella coda, e poi la pulirà.

Luca Oskari Fiumanò

Gestione delle Wave

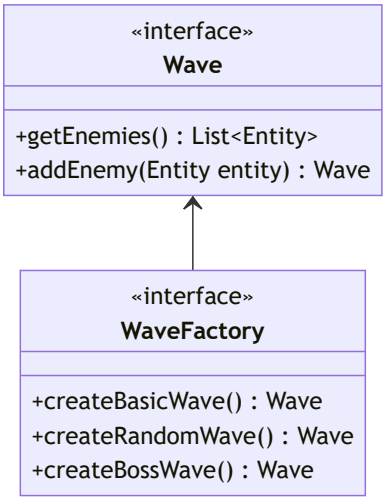

Figura 2.11: Rappresentazione UML del pattern Factory per la gestione delle Wave

Problema Esistono diversi tipi di Wave ognuna composta da nemici di vario tipo e numero.

Soluzione Inizialmente per risolvere questo problema volevo utilizzare il *Prototype* pattern creando una singola istanza per ogni tipo di nemico (prototipi) che sarebbero state clonate quando ritenute necessarie, inizialmente l'implementazione mi sembrava appropriata per risolvere il problema, ma non avevo preso in considerazione un problema fondamentale, infatti il metodo clone() che applicava il pattern non creava mai una nuova istanza ma passava solamente un riferimento del prototipo su cui era stato chiamato;

questo causava diversi problemi. Usando questo pattern i nemici avrebbero avuto tutti un riferimento agli stessi componenti, questo causava ad esempio la condivisione di danno ricevuto tra tutti i nemici che avevano clonato lo stesso prototipo. Ho capito quindi che sarebbe servito un pattern che creasse i diversi tipi di Wave formandole con istanze diverse di nemici; dopo un attenta ricerca la soluzione migliore mi è sembrata l'utilizzo di una *Factory*. Così facendo mi sarei ritrovato nella condizione di poter creare un diverso tipo di wave con la sola chiamata di un metodo della *Factory* senza incappare in errori di riferimenti comuni.

Gestione delle Window indipendentemente dall'architettura grafica scelta

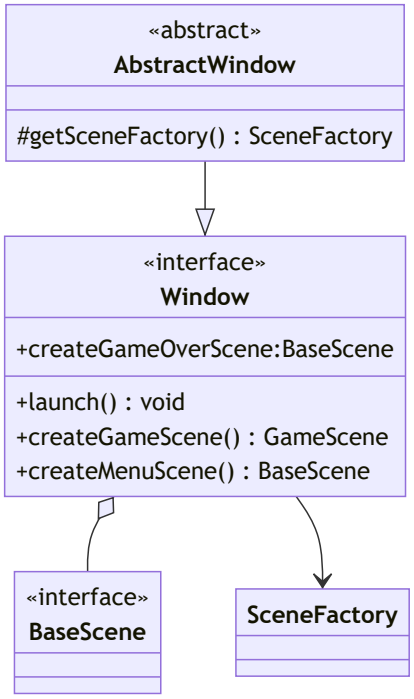

Figura 2.12: Rappresentazione UML dell'applicazione del pattern Template Method alla Window

Problema La gestione delle Window deve essere indipendente dall'architettura grafica scelta.

Soluzione Per risolvere il problema ho deciso di utilizzare una classe astratta AbstractWindow che implementa il *template method pattern* per definire

come verranno create le scene. I template method sono `createGameScene()`, `createMenuScene()` e `createGameOverScene()` che utilizzano il metodo astratto `getSceneFactory()` che sarà implementato concretamente in una classe che estende AbstractWindow.

Gestione delle diverse scene indipendentemente dall'architettura grafica scelta

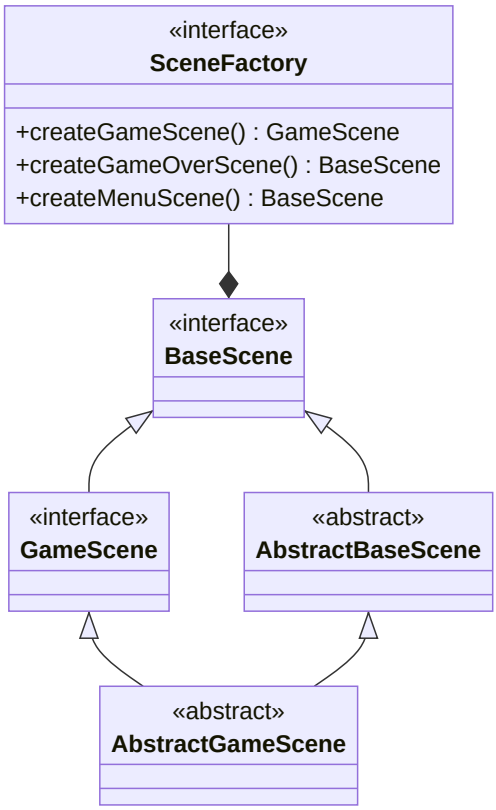

Figura 2.13: Rappresentazione UML dell'applicazione del pattern Factory-Method alla Scene

Problema La gestione e creazione delle Scene deve essere indipendente dall'architettura grafica scelta.

Soluzione Per venire a capo del problema ho deciso di utilizzare il *pattern Factory* per delegare la creazione delle diverse scene in maniera indipendente dall'architettura grafica scelta; infatti se si volesse utilizzare una determinata architettura si dovrebbe creare una *Factory* concreta per essa.

Gestione di multipli PowerUp

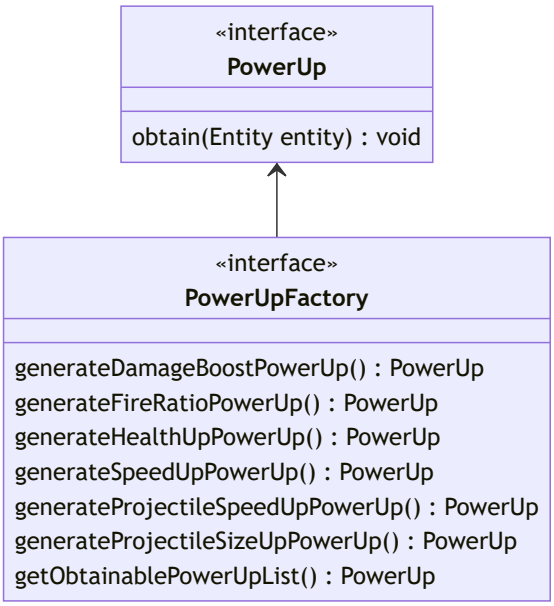

Figura 2.14: Rappresentazione UML dell'applicazione del pattern Factory e Strategy ai PowerUp

Problema Il gioco deve implementare diversi tipi di PowerUp, ognuno che lavora su componenti diversi, che devono essere gestiti.

Soluzione Dato che il nostro gioco doveva avere la possibilità di creare diversi tipi di PowerUp ognuno con comportamenti diversi ho deciso di utilizzare il *Factory pattern* per sostituire la costruzione diretta dei PowerUp con delle appropriate chiamate alla *Factory*. L'utilizzo del *Factory method pattern* si è portato dietro un altro vantaggio la possibilità di aggiungere facilmente nuovi PowerUp. Per risolvere il problema dei diversi comportamenti ho invece usato il pattern Strategy direttamente dentro la *Factory* mediante il suo metodo privato `fromFunction()`.

Leonardo David Matteo Magnani

Getione delle entità

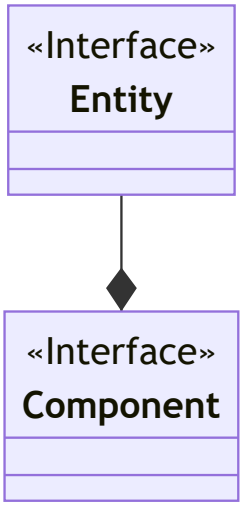

Figura 2.15: Rappresentazione UML dell'applicazione del Component Pattern alla enttità

Problema La classe `Entity` copre più domini rompendo il *Single Responsability Principle*. Inoltre l'interfaccia `Entity` potrebbe avere numerose implementazioni in base al comportamento che li si vuole dare. L'utilizzare l'ereditarietà comporterebbe la creazione di molte sottoclassi con la conseguenza di ripetizione di codice. Inoltre utilizzare l'ereditarietà multipla potrebbe portare numerosi problemi, come Deadly Diamond.

Soluzione Il *Component pattern* permette di creare dei componenti altamente riusabili e indipendenti. Utilizzando il *Component pattern*, quindi favorendo la composizione rispetto alla ereditarietà, si riesce a scalare molto velocemente sul numero di entità. Per descrivere il comportamento generale dell'entità gli vengono assegnati i componenti desiderati. Dividendo il dominio e assegnandolo ad un specifico componente viene rispettato il *Single Responsability Principle*. Questo comporta una migliore manutenibilità dovuta all'assenza di ripetizione di codice.
NOTA: La progettazione alla soluzione del problema è stata fatta in collaborazione con Cecchini Andrea.

Comunicazione tra le componenti dell'entità

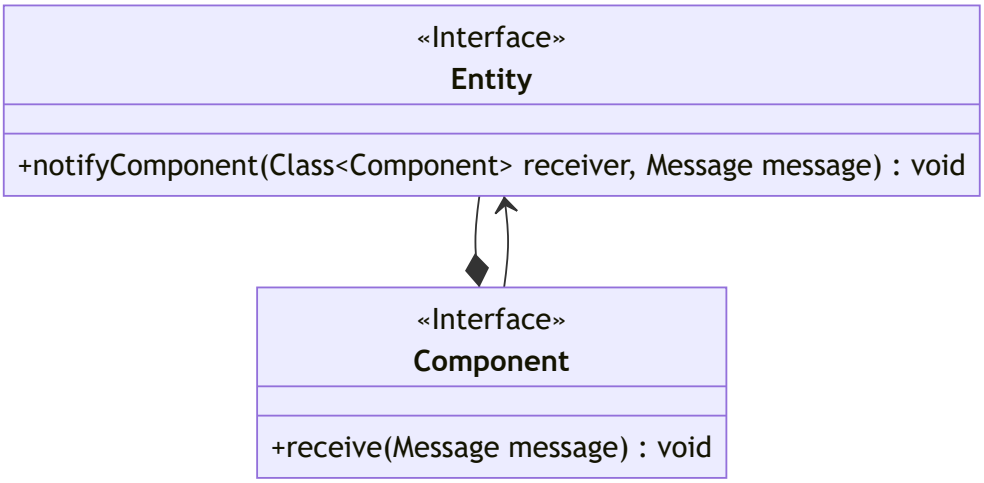

Figura 2.16: Rappresentazione UML dell'applicazione del Mediator pattern per la comunicazione tra le componenti

Problema I componenti hanno bisogno di comunicare tra di loro. Nel progetto:

- InputComponent deve notificare il movimento da effettuare al PhysicsComponent.
- PhysicsComponent deve notificare il CollisionComponent della nuova posizione dopo il movimento.
- CollisionComponent deve notificare HealthComponent per indicare il danno ricevuto dalla collisione.

Soluzione Il *Mediator Pattern* è un modo efficace di gestire la comunicazione tra componenti all'interno di un'entità. Utilizzando tale pattern, ciascun componente può inviare messaggi a un altro componente tramite la sua entità/mediatore. Ogni componente ha una referenza alla sua entità, consentendogli di inviare messaggi a un componente specifico utilizzando il metodo `notifyComponent()`. Una volta ricevuto il messaggio, l'entità si occupa di passarlo al componente desiderato tramite il metodo `receive()`, consentendo così al componente ricevente di decidere come gestire il messaggio. Questo pattern offre la versatilità necessaria per gestire la comunicazione

tra componenti in modo efficace e affidabile.
NOTA: La progettazione alla soluzione del problema è stata fatta in collaborazione con Cecchini Andrea.

Creazione di collisioni con comportamenti diversi

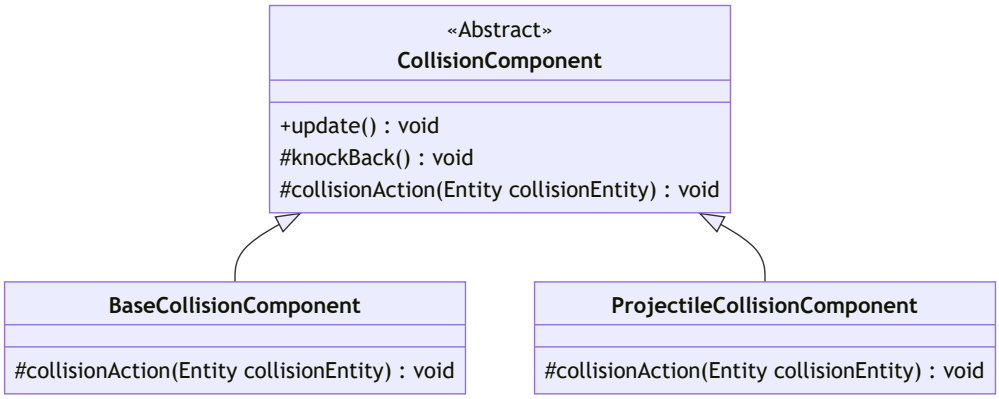

Figura 2.17: Rappresentazione UML dell'applicazione del Template Method per la gestione di CollisionComponent con comportamenti diversi

Problema Alcune entità devono agire in modo diverso in caso di collisione. Nel progetto, alcune entità perdono la vita in base al danno causato dalla collisione, mentre altre (i proiettili) si distruggono all'impatto.

Soluzione Per gestire diverse entità che devono comportarsi in modo diverso in caso di collisione, nel progetto è stato utilizzato il *Template Method pattern*. Il metodo template è `update()`, che chiama un metodo astratto `collisionAction()` per gestire il comportamento che l'entità deve prendere in caso di collisione, e un metodo protetto `knockback()` che gestisce il comportamento della collisione con entità rigide. In questo modo, si è creato un modo riutilizzabile e flessibile per gestire le collisioni tra entità che devono agire in modo diverso.
Il metodo `knockback()` anche se non utilizzato nell'implementazioni è stato messo per futura estendibilità.

Creazione delle entità

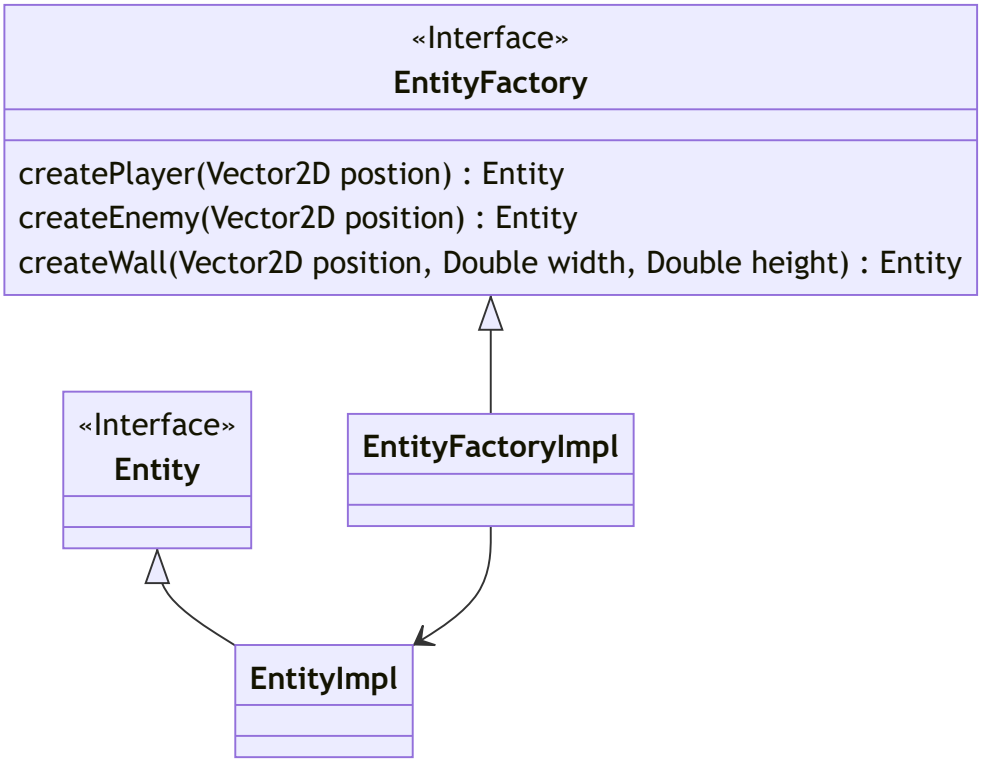

Figura 2.18: Rappresentazione UML dell'applicazione del Factory Method per la creazione di diversi tipi di entità,
NOTA: sono mostrate solo tre funzioni nella factory per rapprestazione

Problema Ci possono essere più entità con componenti diversi e quindi comportamenti diversi.

Soluzione Per la creazione di entità è stato utilizzato il *Factory Method pattern*, è un modo semplice e flessibile per creare entità con componenti diversi e quindi comportamenti diversi. Ad esempio, una factory può creare un'entità che ha una fisica e collisione, oppure un'entità che abbia solo una collisione (come il muro di gioco). Una factory crea una superficie di abstracation per la creazione delle entità e nasconde la loro logica di implementazione. Tramite questo pattern, un codice chiamante può passare dei parametri specifici e ricevere un'istanza di un'entità pre-configurata. Questo significa che non è necessario scrivere il codice di creazione di un'entità in più

posti, ma basta creare una factory una volta e passare i parametri a seconda dei casi d'uso.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per effettuare i test abbiamo utilizzato JUnit 5.
I test che abbiamo effettuato sono:

- **ShapeTest**, questa classe controlla le intersezioni tra i differenti Shape, assicurandosi che funzionino come previsto.
- **StopWatchTest**, questa classe verifica il funzionamento dello StopWatch nel caso in cui cerca di avviarlo quando già avviato oppure di fermarlo quando già fermato.
- **Vector2DTest**, questa classe verifica che le operazioni di Vector2D siano esatte.
- **GameEngineTest**, testati i getter del gameEngine, in modo tale da controllare che non ritornassero elementi nulli. E' stato controllato l'accesso ai metodi run e stop, in modo tale da verificare se venissero lanciate le giuste eccezioni quando il programmatore effettua errori di programmazione.
- **MessageTest**, in questa classe viene testato la messaggistica tra le componenti. Di preciso ho testato la comunicazione e ricevimento dei messaggi del PhysicsComponent e CollisionComponent
- **GameTest**, sono state testate le funzionalità basiche della classe Game, in particolare i vari getters, l'update e il metodo isOver(). Inoltre sono stati testati i metodi della factory di game, in modo da verificare che tali metodi ritornassero gli oggetti della classe Game con le caratteristiche volute

- **StateTest**, si testa l'implementazione della logica e dello stato della partita, in particolare
  - L'incremento del round e il suo relativo get.
  - Controllo dello stato attuale della partita attraverso il metodo isOver.
  - Controllo dello stato della wave del round corrente
- **EntityTest**: questa classe verifica che si ottengano i componenti giusti dall'entità quando richiesti.
- **EventTest**, in questa classe vengono testati i vari tipi di eventi, controllando che vengano eseguiti dal World e non prima.
- **HealthComponentTest**, si testa il funzionamento del componente HealthComponent controllando specificatamente cosa succede quando si riceve un danno minore della vita attuale e cosa succede se si riceve un danno che porterebbe la vita sotto lo zero.
- **InputTest**, in questa classe viene testato l'input da testiera ec una delle AI.
- **PowerUpTest**, si testa il corretto funzionamento di tutti i possibili power up.
- **WaveTest**, in questa classe di test viene testato la corretta generazione di una wave controllando specificatamente la generazione di una wave basica e di una wave con boss, viene inoltre testato il la correttezza dell'aggiunta di un Entity alla wave.
- **WorldTest**, sono state testate funzioni basi come l'aggiunta e rimozione di entità, il metodo set e get dell'oggetto Wave all'interno del mondo. Inoltre sono stati testati i metodi della factory di world, in modo da verificare che tali metodo ritornassero world con le caratteristiche volute

Dal punto di vista della grafica non sono stati svolti test automatizzati visto che la totalità della renderizzazione grafica è stata affidata al componente Canvas di JavaFX e al suo GraphicContext, risultavano quindi veramente minime le funzionalità da testare. Inoltre ciò che doveva essere visibile alla grafica veniva prelevato dal controller portando eventuali altri test a essere solo banali ripetizioni di test già effettuati.

3.2 Metodologia di lavoro

Prima di cominciare con lo sviluppo abbiamo impostato l'architettura del gioco per avere delle solide fondamenta, iniziando con la definizione delle interfacce.
La suddivisione iniziale del processo è stata fatta in maniera adeguata, consentendoci di lavorare parallelamente; riunendoci solo quando era necessario interlacciare le diverse implementazioni o per discutere su eventuali aggiunte.
Abbiamo deciso di utilizzare il DVCS con il semplice approccio spiegato in classe, difatti dopo la creazione del repository ognuno di noi ne ha fatto un clone per lavorarci in autonomia condividendo le aggiunte mediante pull e push.

Cecchini Andrea

In autonomia mi sono occupati di:

- GameEngine e GameLoop (package t2sgame.core.engine)
- Implementazione del gioco (package t2sgame.game)
- Implementazione del mondo di gioco (package t2sgame.game.model)
- Implementazione della logica di gioco (package t2sgame.game.logics)

In collaborazione mi sono occupato:

- Con Leonardo Magnani:
  - Gestione delle collisioni: mi sono occupato di trovare un modo flessibile e mantenibile per gestire la collisione fra forme di tipo diverso.
  - Gestione dei componenti.
  - Gestione della messagistica.

Nicolò D'Addabbo

In autonomia mi sono occupato di:

- Input Component (in t2sgame.game.ecs.impl)
- Gestione degli input da periferica (package t2sgame.input)
- AI (package t2sgame.input)

- Comandi di gioco (package t2sgame.input)
- Gestione degli eventi (Event ed EventFactory in t2sgame.game.logics e metodi notifyEvent e handleEvent in World)

Luca Oskari Fiumanò

In autonomia mi sono occupato di:

- GraphicComponent e rispettiva sezione in ComponentFactory (in t2sgame.game.ecs.impl)
- HealthComponent (in t2sgame.game.ecs.impl)
- Gestione dei PowerUp e rispettiva Factory (in t2sgame.game.model.impl)
- Gestione delle Wave e rispettiva Factory (in t2sgame.game.model.impl)
- Gestione della totalità della View (package t2sgame.view)
- Gestione del launcher (in t2sgame.view.api e t2sgame)

In collaborazione mi sono occupato di:

- EntityFactory insieme a Magnani Leonardo David Matteo (in t2sgame.game.model.impl)

Magnani Leonardo David Matteo

In autonomia mi sono occupato di:

- Le entità di gioco (Entity in t2sgame.game.ecs)
- Alcune componenti delle entità (DamageComponent e ShootComponent in t2sgame.game.ecs.impl)
- La fisica del gioco (PhysicsComponent in t2sgame.game.ecs.impl, Vector2D in t2sgame.common)
- Le implemtazioni delle collisioni (BaseCollisionComponent e ProjectileCollisionComponent in t2sgame.game.ecs.impl, Circle e Rectangle in t2sgame.common.shapes)
- Uno stopwatch per la gestione del tempo trascorso (StopWatch in t2sgame.common)

In collaborazione con Luca Oskari Fiumanò:

- EntityFactory (t2sgame.game.model.impl)

In collaborazione con Andrea Cecchini:

- Gestione delle collisioni (CollisionComponent in t2sgame.game.ecs.impl, Shape in t2sgame.common)
- Gestione dei componenti.
- Gestione della messagistica.

3.3 Note di sviluppo

Cecchini Andrea

Utilizzo di Stream

L'uso di streams pervade la maggior parte del codice. Il seguente è solo un esempio:
`https://github.com/nicolodaddabbo/OOP22-t2s-game/blob/6dc6e41a127a2c02e07ca14e62`
`src/main/java/it/unibo/t2sgame/core/engine/impl/GameEngineImpl.java#`
`L115-L119`

Creazione di interfacce funzionali

Esempio di una classe funzionale utilizzata per incapsulare messaggi tra componenti:
`https://github.com/nicolodaddabbo/OOP22-t2s-game/blob/a2aae31b62e6f41b9cc2d0ee80`
`src/main/java/it/unibo/t2sgame/game/ecs/api/Message.java#L3-L15`

Utilizzo di Lambda e Method references

Sono numerosi gli utilizzi di lambda e method references. In seguito alcuni esempi:
`https://github.com/nicolodaddabbo/OOP22-t2s-game/blob/6dc6e41a127a2c02e07ca14e62`
`src/main/java/it/unibo/t2sgame/core/engine/impl/GameEngineImpl.java#`
`L145-L155`

Nicolò D'Addabbo

Utilizzo di Stream, lambda expression e method reference

Usate molto spesso in vari punti del progetto:

- Permalink: https://github.com/nicolodaddabbo/OOP22-t2s-game/blob/d1453f9e5f56d2cdfec21561311304a309e99ed9/src/main/java/it/unibo/t2sgame/game/logics/impl/EventFactoryImpl.java#L37
- Permalink: https://github.com/nicolodaddabbo/OOP22-t2s-game/blob/d1453f9e5f56d2cdfec21561311304a309e99ed9/src/main/java/it/unibo/t2sgame/input/api/AbstractChasingAIInputController.java#L45-L48
- Permalink: https://github.com/nicolodaddabbo/OOP22-t2s-game/blob/d1453f9e5f56d2cdfec21561311304a309e99ed9/src/main/java/it/unibo/t2sgame/input/api/AbstractChasingAIInputController.java#L45-L48

Classe con Generics

- Permalink: https://github.com/nicolodaddabbo/OOP22-t2s-game/blob/d1453f9e5f56d2cdfec21561311304a309e99ed9/src/main/java/it/unibo/t2sgame/input/impl/EntityStateImpl.java#L14

Luca Oskari Fiumanò

Utilizzo di Stream, lambda expression e method reference

Usate frequentemente

- Permalink: https://github.com/nicolodaddabbo/OOP22-t2s-game/blob/69dda3855df80a277137dc0033e6d9ae3a6d71a3/src/main/java/it/unibo/t2sgame/game/model/impl/WaveFactoryImpl.java#L96-L100
- Permalink: https://github.com/nicolodaddabbo/OOP22-t2s-game/blob/9a90fd37a7e43a7f97ee824fb1158284e79280f7/src/main/java/it/unibo/t2sgame/view/impl/GameSceneJavaFXImpl.java#L115-L119
- Permalink: https://github.com/nicolodaddabbo/OOP22-t2s-game/blob/69dda3855df80a277137dc0033e6d9ae3a6d71a3/src/main/java/it/unibo/t2sgame/game/model/impl/WaveFactoryImpl.java#L26-L29
- Permalink https://github.com/nicolodaddabbo/OOP22-t2s-game/blob/69dda3855df80a277137dc0033e6d9ae3a6d71a3/src/main/java/it/unibo/t2sgame/T2S.java#L9

Utilizzo JavaFX

Utilizzata per realizzare la parte di View

- Permalink: https://github.com/nicolodaddabbo/OOP22-t2s-game/blob/69dda3855df80a277137dc0033e6d9ae3a6d71a3/src/main/java/it/unibo/t2sgame/view/impl/GameSceneJavaFXImpl.java#L92-L95

Creazione interfaccie funzionali

- Permalink: https://github.com/nicolodaddabbo/OOP22-t2s-game/blob/69dda3855df80a277137dc0033e6d9ae3a6d71a3/src/main/java/it/unibo/t2sgame/game/model/api/PowerUp.java#L7-L14

Leonardo David Matteo Magnani

Utilizzo di stream, optional, method reference e metodo generico

- Permalink: https://github.com/nicolodaddabbo/OOP22-t2s-game/blob/48fe160af0c4cdbfbf7fe791b784f6f4ad7b185e/src/main/java/it/unibo/t2sgame/game/ecs/impl/EntityImpl.java#L46-L51

Utilizzo di stream, optional, lambda e metodo generico

- Permalink: https://github.com/nicolodaddabbo/OOP22-t2s-game/blob/48fe160af0c4cdbfbf7fe791b784f6f4ad7b185e/src/main/java/it/unibo/t2sgame/game/ecs/impl/EntityImpl.java#L69-L71

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Andrea Cecchini

Sono contento dell'opportunità avuta nel realizzare questo progetto.
Per quanto riguarda il lavoro in gruppo, sono molto soddisfatto del coordinamento che vi è stato. Lo scambio di opinioni e di idee è una delle caratteristiche fondamentali che rappresenta il nostro gruppo.
Questo progetto mi ha dato la possibilità di sperimentare moltissimo sul scrivere codice "pulito" e "riusabile". Nonostante l'inesperienza nello sviluppo di un gioco, sono abbastanza contento del risultato. Ritengo comunque che ci siano parti da migliorare e come esercizio, una volta effettuata la consegna, avrò la possibilità di imparare ancora.

Nicolò D'Addabbo

Sono soddisfatto del nostro progetto, che è stato un ottimo esercizio di lavoro di gruppo. Abbiamo collaborato in maniera efficace grazie a una buona fase di analisi, che è stata fondamentale per individuare quali problemi risolvere e come affrontarli. Questo progetto mi ha permesso di sviluppare abilità che possono essere applicate a progetti futuri, come la gestione del codice in gruppo ed una buona organizzazione del lavoro. Ho lavorato alla parte Model e ho collaborato al Controller. Purtroppo non ho avuto la possibilità di approfondire JavaFX, che mi sarebbe interessato molto, poiché non ho avuto parti di View.

Luca Oskari Fiumanò

Mi ritengo abbastanza soddisfatto del lavoro svolto nel progetto nonostante numerose possibili aggiunte e migliorie effettuabili. All'interno del gruppo svolgevo il compito di referente per la gestione della grafica, dei nemici e dei power up. Ritengo che questo progetto rimarrà solamente un progetto didattico e di non portarlo avanti a meno di eventuali risvolti futuri. Nonostante questo ritengo di aver imparato diverse competenze applicabili per un futuro lavorativo, quali l'utilizzo di DVCS (Git) e il lavoro di gruppo.

Leonardo David Matteo Magnani

Sono soddisfatto del risultato finale considerando il tempo limitato a disposizione. La buona analisi effettuata prima dello sviluppo ha permesso di creare una solida base per il progetto. Nonostante alcune parti siano state riscritte durante la fase di sviluppo per migliorare il codice, l'architettura iniziale è rimasta intatta. Tuttavia, a causa di impegni accademici e del tempo limitato, non ho avuto la possibilità di apportare miglioramenti a certe parti del mio lavoro. Un'area di miglioramento potrebbe essere la gestione delle collisioni a cui ero molto interessato a fare efficientemente. Anche se questo progetto mi ha permesso di acquisire competenze organizzative e architetturali e di mettere alla prova le mie capacità come ingegnere, personalmente ritengo che non sia il progetto giusto per me da portare avanti in futuro.

Appendice A

Guida utente

Quando viene fatto partire l'applicativo ci si ritroverà in una scena con due bottoni:

- **Single Player**: Versione del gioco in cui il giocatore è solo contro le ondate di nemici.
- **With Companion**: Versione del gioco semplificata in cui il giocatore è aiutato da un Companion (una semplice IA, che attacca i nemici)

I comandi di gioco sono i seguenti:

- **ESC** (Escape): chiude l'applicazione
- **W**: Movimento verso l'alto.
- **A**: Movimento verso sinistra.
- **S**: Movimento verso il basso.
- **D**: movimento verso destra
- ↑ (Up Arrow): sparo verso l'alto.
- ↓ (Down Arrow): sparo verso il basso.
- ← (Left Arrow): sparo verso sinistra.
- →(Right Arrow): sparo verso destra.

Bibliografia

- Nella fase di analisi abbiamo utilizzato come spunto il git mostrato durante la lezione svolta dal professore A. Ricci. `https://github.com/pslab-unibo/oop-game-prog-patterns-2022`