

Relazione del progetto di  
“Programmazione ad Oggetti”

Giacomo Amadio, Daniel Pellanda e Davide Zandonella

02/07/2022

# Indice

<b>1</b>	<b>Analisi</b>	<b>2</b>
1.1	Requisiti . . . . .	2
1.2	Analisi e modello del dominio . . . . .	3
<b>2</b>	<b>Design</b>	<b>4</b>
2.1	Architettura . . . . .	4
2.2	Design dettagliato . . . . .	6
2.2.1	Giacomo Amadio . . . . .	6
2.2.2	Daniel Pellanda . . . . .	10
<b>3</b>	<b>Sviluppo</b>	<b>13</b>
3.1	Testing automatizzato . . . . .	13
3.2	Metodologia di lavoro . . . . .	14
3.2.1	Giacomo Amadio . . . . .	14
3.2.2	Daniel Pellanda . . . . .	16
3.3	Note di sviluppo . . . . .	18
3.3.1	Giacomo Amadio . . . . .	18
3.3.2	Daniel Pellanda . . . . .	19
<b>4</b>	<b>Commenti finali</b>	<b>20</b>
4.1	Autovalutazione e lavori futuri . . . . .	20
4.1.1	Giacomo Amadio . . . . .	20
4.1.2	Daniel Pellanda . . . . .	20
<b>A</b>	<b>Guida utente</b>	<b>22</b>
<b>B</b>	<b>Esercitazioni di laboratorio</b>	<b>23</b>
B.1	Daniel Pellanda . . . . .	23

# Capitolo 1

## Analisi

### 1.1 Requisiti

JetScape nasce come volontà di cimentarsi nella riproduzione del concetto di base di un classico dei giochi per smartphone, Jetpack Joyride.

#### Requisiti funzionali

- JetScape spingerà l'utente a padroneggiare le meccaniche di gioco, per destreggiarsi fra insidie e ostacoli nel percorso, alimentando la voglia di migliorarsi grazie ad un livello di difficoltà incrementale.
- JetScape dovrà avere un'elevata responsività a livello audio-visivo, così da proporre all'utente un'esperienza quanto più piacevole e immersiva.
- JetScape spronerà il giocatore ponendolo di fronte ai suoi migliori risultati.
- Jetscape si dovrà ricordare le preferenze dell'utente.

#### Requisiti non funzionali

- JetScape dovrà mantenere una fluidità accettabile su gran parte dei sistemi operativi.

## 1.2 Analisi e modello del dominio

JetScape metterà a disposizione dell'utente una selezione di opzioni per navigare fra le varie schermate (display). Ogni partita l'obiettivo dell'utente sarà quello di totalizzare dei punteggi sempre più alti, ottenibili evitando il contatto con gli ostacoli (obstacles) il più a lungo possibile. Sul percorso, tuttavia si possono anche trovare degli oggetti raccogliibili (pickable), che faciliteranno l'utente (player) nel raggiungimento del suo obiettivo.

Gli elementi costitutivi il problema sono sintetizzati in Figura 1.1.

A fine partita verranno aggiornate le statistiche ed eventualmente registrati nuovi record.

La difficoltà principale sarà la generazione consistente di ostacoli e percorsi in grado di mettere alla prova l'utente, mantenendo una coerenza audiovisiva adeguata e gestendo le interazioni fra il player e le varie entità.

Il requisito non funzionale riguardante la fluidità, richiederà test specifici delle performance di JetScape su diverse piattaforme, che non potranno essere effettuati all'interno del monte ore previsto. Tale feature sarà oggetto di futuri lavori, ma per il momento si limiterà ad avere massima fluidità sul sistema operativo Windows.

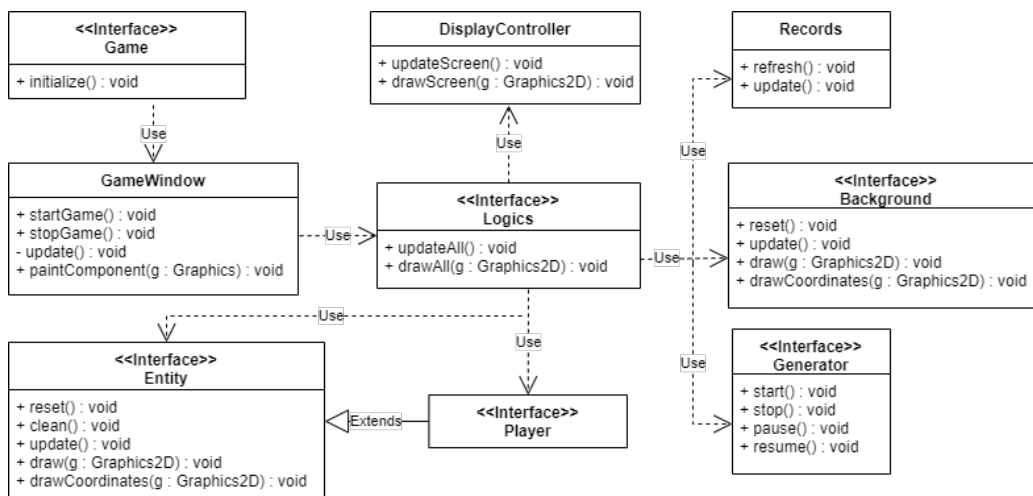


Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

# Capitolo 2

## Design

### 2.1 Architettura

L'architettura pensata per JetScape non segue alcun modello in particolare. L'idea è stata quella di affidare alla **GameWindow** la paternità del thread principale, delle istanze dei vari servizi richiamabili globalmente e delle principali classi di gestione, quali il **KeyHandler** e il **LogicsHandler** Figura 2.1.

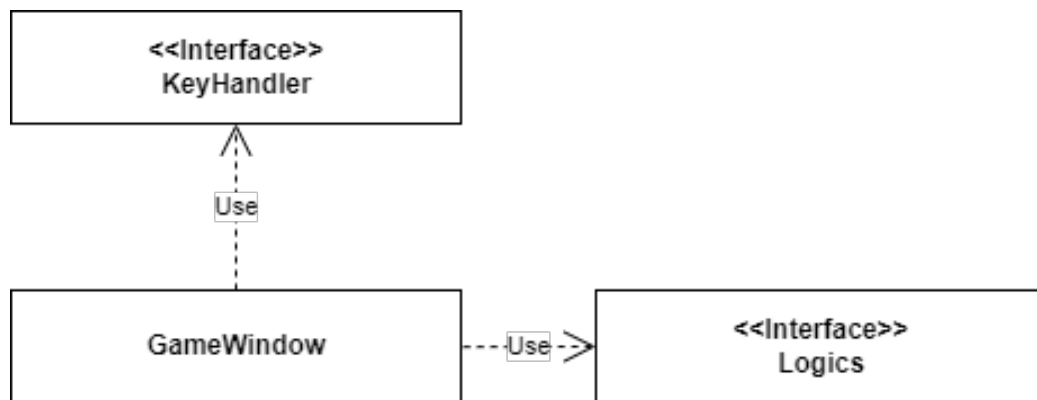


Figura 2.1: Rappresentazione UML dell'architettura utilizzata.

La prima è incaricata del rilevamento dell'input da tastiera, che viene poi gestito nelle varie classi. La seconda è la classe di controllo principale, che si occupa di aggiornare e disegnare le varie **Entity** generate da **TileGenerator** e di istanziare **DisplayController**, che si occupa dei menù.

Le **Entity** rappresentano diversi oggetti nell'ambiente di gioco, implementando due metodi principali, che si occupano di svolgere i compiti di View e Model Figura 2.2.

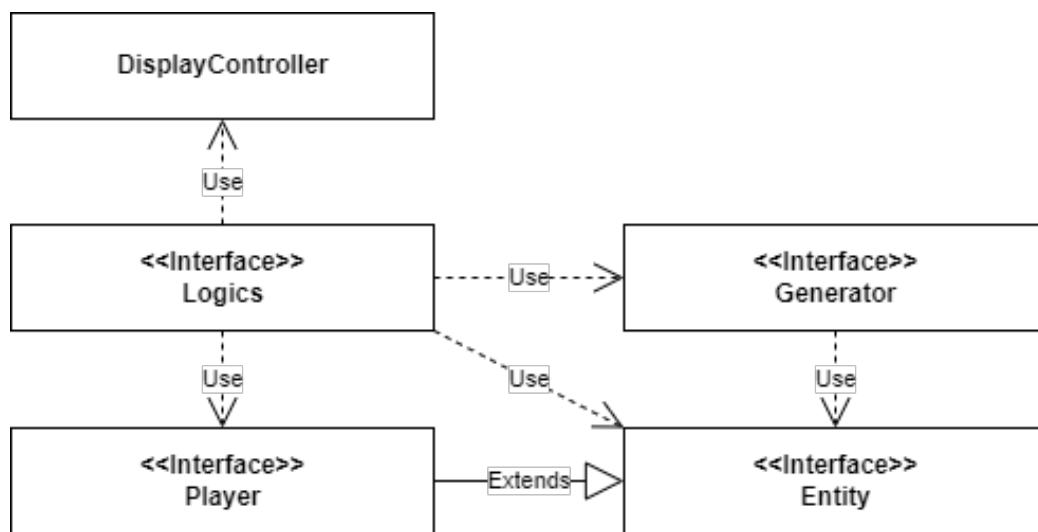


Figura 2.2: Rappresentazione UML dell'architettura utilizzata.

La parte architetturale del `DisplayController` invece, aderisce al pattern MVC, reputato il più adatto per la soluzione. Nello specifico il `DisplayController` rappresenta la parte di control che mette in relazione la parte di model del `DisplayHandler` e quella di view delle varie istanze di `Display`.

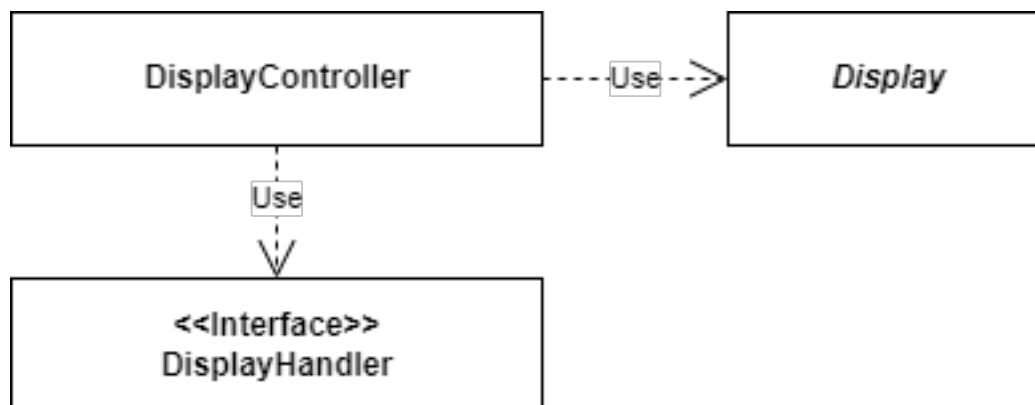


Figura 2.3: Rappresentazione UML dell'architettura utilizzata.

## 2.2 Design dettagliato

### 2.2.1 Giacomo Amadio

Controllo e gestione delle collisioni

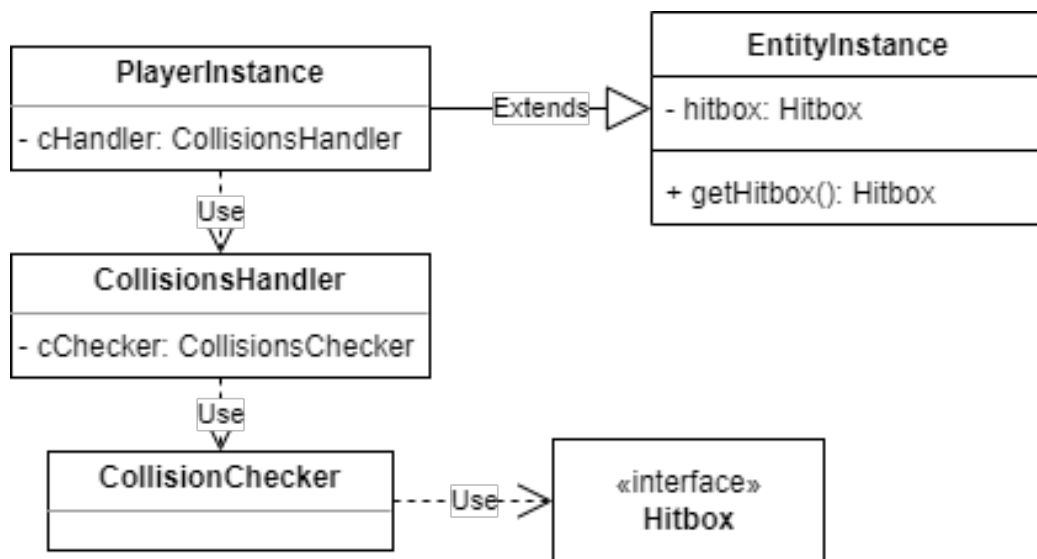


Figura 2.4: Rappresentazione UML della soluzione descritta.

**Problema** JetScape ha diverse entità in movimento, che al momento del contatto, anche simultaneo, con il giocatore devono essere in grado di interagirvi.

**Soluzione** Il sistema per il rilevamento delle collisioni è schematizzato in Figura 2.4: le implementazioni di **Entity** possiedono delle istanze di **Hitbox**, aventi dimensioni e forme differenti, adattabili in base alle necessità delle **Entity** stesse. Nello specifico **PlayerInstance** ha accesso all'istanza di **CollisionsHandler** che "consuma" le eventuali collisioni, registrate da **CollisionChecker** in caso di intersezioni fra le **Hitbox** di **PlayerInstance** e quelle di altre **Entity**, per poi avviare la routine di gestione associata.

## Gestione menù e cambiamento schermate

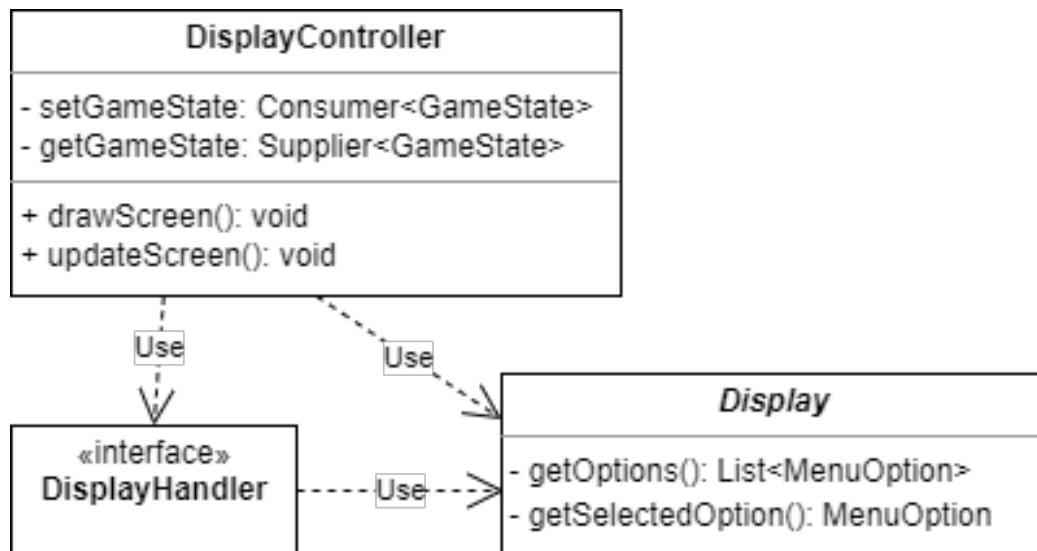


Figura 2.5: Rappresentazione UML della soluzione descritta.

**Problema** JetScape deve avere dei menù facilmente navigabili e responsivi, che gestiscono le transizioni fra le varie schermate di gioco.

**Soluzione** La transizione fra le schermate viene controllata dal **DisplayController**, che sceglie quale istanza di **Display** mostrare all'utente, in base al **GameState** attuale. Ogni istanza di **Display** viene gestita da un **MenuHandler** che si occupa dello scorrimento del cursore e di comunicare l'attuale **GameState**, che potrebbe venire alterato da un'eventuale selezione di **MenuOption**. Rappresentazione della soluzione in figura Figura 2.5.



## Caricamento Font e riuso codice schermate

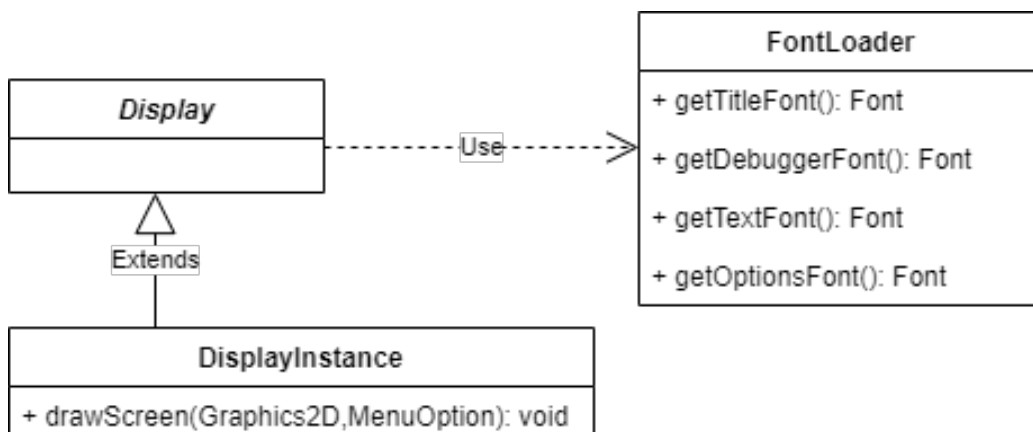


Figura 2.6: Rappresentazione UML delle classi di tipo Display, che implementano un comportamento differente per disegnare ogni schermata.

**Problema** In fase di sviluppo sono state create per JetScape varie schermate di gioco e tipi di scritte. Ci si è accorti che vi erano diverse similarità fra di esse, portando ad avere classi molto simili e a duplicazione di codice. Inoltre i font utilizzati non erano necessariamente installati di default su tutte le macchine, causando così degli effetti indesiderati qualora il problema si fosse presentato.

**Soluzione** Per ridurre al minimo le duplicazioni di codice si è optato per la creazione di una classe astratta **Display**, uniformando la parte estetica e predisponendo il tutto per possibili aggiunte future. Questa uniformazione estetica è stata applicata anche ai font, che grazie al **FontLoader** previene anche eventuali problematiche dovute alla non presenza di determinati font su macchine diverse, caricandoli direttamente dalle risorse di JetScape. Soluzione raffigurata in Figura 2.6.

## Gestione suoni

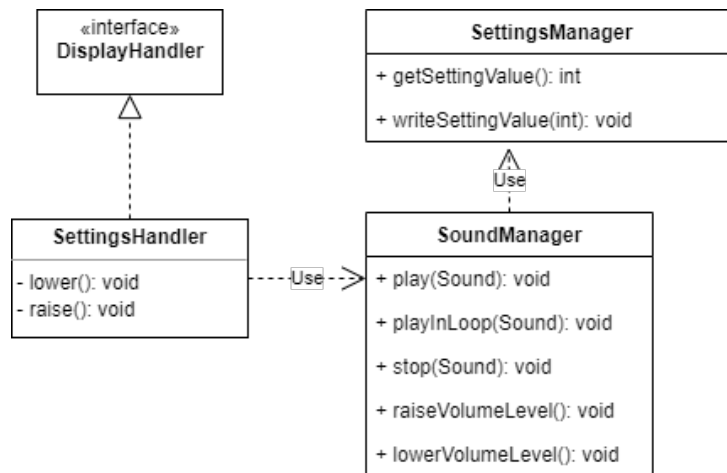


Figura 2.7: Rappresentazione UML della soluzione descritta.

**Problema** JetScape deve rendere possibile l'ascolto di più tracce audio in contemporanea (musica e vari livelli di suoni), con volume differente e modificabile a piacimento, "ricordando" le preferenze dell'utente.

**Soluzione** Dato che l'idea era quella di mantenere separati il volume dei suoni da quello delle musiche, si è deciso di rendere istanziabile **SoundManager** partendo dalla tipologia di audio desiderata, per poi andare a leggere e scrivere modifiche sulle preferenze ad esse associate, con l'ausilio di **SettingsManager**. Per rendere più flessibile il codice (anche in vista di future estensioni), si è optato per rendere le 2 istanze richiamabili globalmente, così da apportare modifiche alle impostazioni e far ascoltare le tracce con più facilità. La soluzione è raffigurata in Figura 2.7.

### 2.2.2 Daniel Pellanda

#### Caricamento e gestione della finestra di gioco

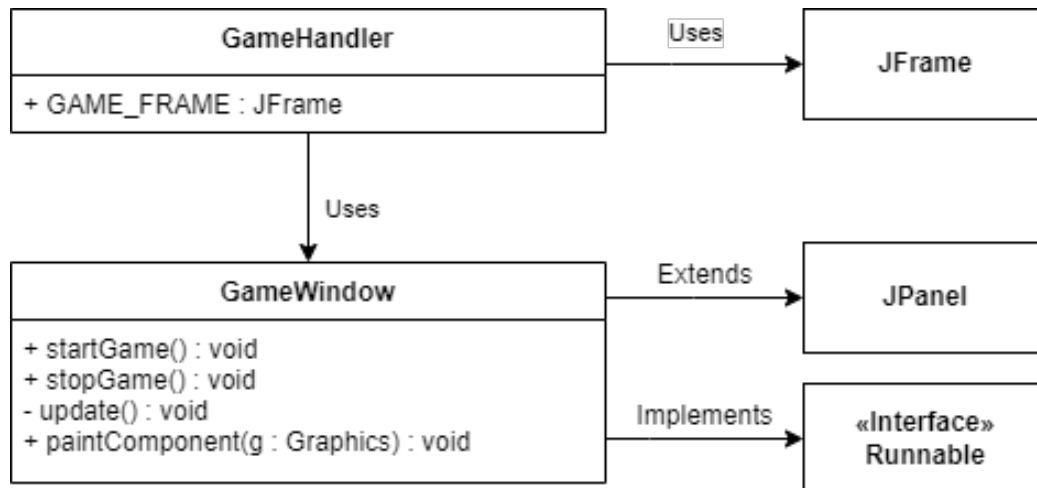


Figura 2.8: La classe **GameHandler** mantiene il frame principale nella quale è contenuta la finestra di gioco.

**Problema** Caricare e gestire una finestra dove il videogame verrà modellato graficamente. Creare un ambiente che permette la continua esecuzione del videogame, cercando di evitare la chiusura prematura del programma.

**Soluzione** L'oggetto **GameHandler** gestisce il frame principale del gioco. Fra le componenti di questo frame viene usato un oggetto speciale **GameWindow** derivato da **JPanel** il quale si occupa di gestire i contenuti grafici e l'esecuzione del gioco. Il componente **GameWindow** interagisce con il **GameHandler** gestendo e controllando i contenuti del frame principale, senza la necessità di accedere all'oggetto direttamente. La Figura 2.8 mostra il modo in cui le precedenti classi sono state strutturate.

## Gestione e logica delle entità

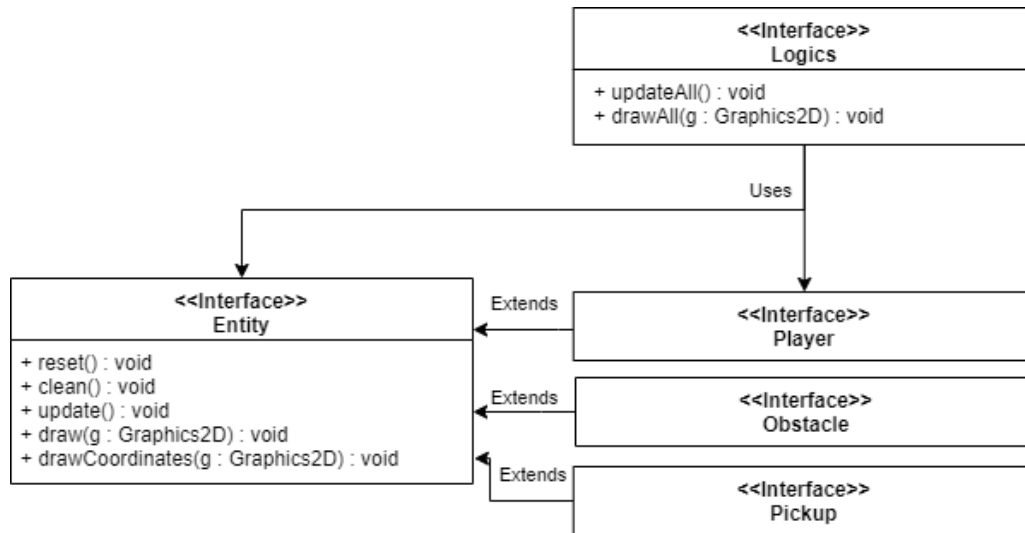


Figura 2.9: Rappresentazione UML della struttura e organizzazione delle classi relative alle entità all'interno della logica di gioco.

**Problema** Gestire il comportamento degli oggetti interattivi, meglio detti entità, nella logica di gioco.

**Soluzione** L'interfaccia **Logics** include una struttura dati che tiene conto di tutte le entità presenti nell'ambiente di gioco. Ciascuna entità è rappresentata da una classe specifica che deriva da **Entity** o da un suo sottotipo di interfaccia (**Obstacle** o **Pickup**). La necessità di dover dichiarare una classe per ciascun tipo di oggetto è data dalla definizione dei diversi comportamenti di ciascuna entità. **Logics** tiene anche un riferimento all'entità giocatore (rappresentata dall'interfaccia **Player**), essendo oggetto fondamentale per la giocabilità del videogame.

## Generazione delle entità

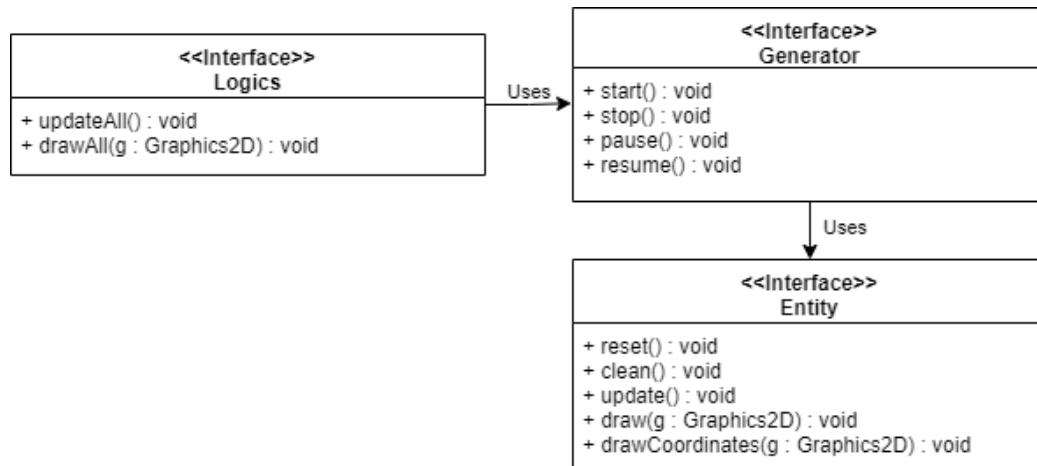


Figura 2.10: Istanziamento del generatore all'interno di **Logics** in modo da permettere la creazione automatica di nuove entità.

**Problema** Gestire una generazione automatica e costante di entità nell'ambiente di gioco. Garantire che l'attività di generazione venga effettuata nell'opportuna fase di gioco e non interferisca nelle altre interazioni.

**Soluzione** L'interfaccia **Logics** si affida ad un'altra interfaccia **Generator** che si occuperà di generare nuove entità, per farlo necessiterà di utilizzare l'opportuna interfaccia per le entità che dovrà creare. L'interfaccia **Generator** avrà il controllo generale sulle entità da visualizzare a schermo (eccetto per l'entità del giocatore) dato che deciderà lui quali entità generare. Tuttavia questo suo controllo sarà limitato dall'interfaccia **Logics** la quale sarà in grado di gestire i suoi periodi di attività, potendo decidere se fermare o continuare la generazione di entità.

# Capitolo 3

## Sviluppo

### 3.1 Testing automatizzato

Il testing di JetScape viene effettuato in maniera completamente automatica attraverso l'utilizzo di JUnit e ciò consiste in una serie di verifiche sugli aggiornamenti e comportamenti di alcune classi determinanti per il funzionamento corretto del videogame.

- Per le classi relative alle entità si fanno principalmente verifiche sui valori dei flag in determinate circostanze, il controllo sul comportamento dei metodi di clean e reset e il controllo sull'assegnazione del corretto tipo di entità; inoltre si controllano i risultati di alcuni metodi specifici relativi all'entità.
- Per il generatore si verificano solo i flag al cambiare del suo stato di attività.
- Per le collisioni viene generata un'entità sul giocatore verificandone l'avvenimento facendo dei controlli sullo stato del giocatore e sul punteggio.
- Per i suoni viene verificata la corretta gestione del volume e in caso di eventi specifici, la corretta introduzione della traccia nella mappa dei suoni in riproduzione.
- Per i menù è stato verificato il corretto scorrimento del cursore su un'istanza della schermata di pausa

Test manuali significativi:

- Per far fronte all'eventualità di dover svolgere test manuali è stata creata la classe **Debugger** di supporto con varie funzionalità, attivabili manualmente dal programmatore, che permettono di avere un feedback visivo e dei log sul terminale per controllare vari avvenimenti e funzionalità di JetScape.
- Sono stati effettuati manualmente dei test sui suoni su sistemi Linux, poiché era sorta una problematica sulla lettura dei file audio dopo una frequente riproduzione di tracce.

## 3.2 Metodologia di lavoro

### 3.2.1 Giacomo Amadio

#### Controllo e gestione delle collisioni

L'implementazione di questa sezione, è stata effettuata sul branch Hitbox, separato dalla linea di sviluppo principale.

Inizialmente l'idea era quella di creare diverse istanze di una classe astratta che definisse il comportamento delle hitbox e che avesse gli estremi di collisione modificabili. Si è deciso dunque di mettere a disposizione delle implementazioni un metodo per la creazione di tali estremi e di mantenere anche una mappa, che associasse ogni rettangolo allo spostamento (x,y) del punto in alto a sinistra rispetto a quello dello sprite, così da gestire con comodità l'aggiornamento delle posizioni dei rettangoli. Ogni frame, dopo il calcolo delle nuove posizioni viene, eseguito un controllo sui rettangoli delle hitbox di tutte le entità. In caso di intersezione viene aggiunta l'entità colpita in una coda delle collisioni da elaborare, per poi essere "consumata" e gestita con la collaborazione del collega Pellanda. Questo modello è poi stato integrato per far fronte ad un nuovo concetto introdotto con il miglioramento della logica, i gruppi di entità. Per far fronte a questa nuova introduzione è stato creato un metodo in grado di costruire la hitbox estraendo le informazioni essenziali da altre hitbox, rendendola volutamente non aggiornabile, così da non dover adattare il normale metodo di spostamento.

#### Gestione suoni

L'implementazione di questa sezione, è stata effettuata sul branch SFX, separato dalla linea di sviluppo principale.

L'idea di partenza era quella di fare in modo di avere una classe che gestisse i suoni il cui volume fosse regolabile. Si è deciso di rendere le istanze

richiamabili globalmente, per semplificare la riproduzione o l'arresto delle tracce dopo determinati eventi di gioco. Quando viene effettuata una richiesta di riproduzione di un suono, tramite apposita enumerazione con associato il nome del file audio, viene dedicata dal sistema della memoria per caricarvi il file sopracitato (necessariamente .wav).

Per questa implementazione tuttavia in fase di test abbiamo riscontrato dei problemi su Linux. Dopo varie ricerche si è giunti alla conclusione che, dopo la riproduzione, una traccia andava necessariamente "liberata" con l'apposito metodo. Quindi prima di effettuare qualsiasi richiesta ai file audio viene fatto un controllo sulla mappa dei suoni caricati, "liberando" la memoria occupata da tutte le tracce già "consumate" e le entry corrispondenti. Per evitare il problema al verificarsi di eventi multipli in brevi intervalli di tempo, viene anche imposto che possa essere ascoltato al massimo un suono per ogni tipo.

Per quanto riguarda il volume, inizialmente vengono caricati i dati salvati sul file settings.json da una classe specializzata. La sezione settings del menù principale permette all'utente di regolare il livello di volume che viene poi registrato sulla classe in esame e salvato su file. Per rendere l'esperienza dell'utente più piacevole e il codice della classe meno complicato si è deciso di crearne 2 istanze per poter regolare il volume di musica e suoni in maniera separata.(possibile miglioramento futuro gestire il tutto con una singola istanza)

## **Gestione menù, cambiamento schermate e caricamento font**

L'implementazione della gestione menù e cambiamento schermate, è stata effettuata sul branch interface, separato dalla linea di sviluppo principale.

L'idea portata avanti per questa parte modella il concetto di stato di gioco (GameState), implementato come enumerazione, grazie al quale viene scelta quale schermata mostrare all'utente, con l'ausilio del DisplayController che mette in relazione la parte di logica dei menù con la parte di grafica.

La parte di logica viene gestita dalle implementazioni dell'interfaccia DisplayHandler che si occupano dello spostamento del cursore e la gestione dei vari input dell'utente. Talvolta questi possono causare un'alterazione del GameState la cui gestione viene delegata ad un metodo presente nel LogicsHandler, così da avere accesso diretto alle informazioni di interesse per avviare la routine di gestione. L'implementazione di tale metodo è poi stata integrata dal collega Pellanda con il procedere del progetto.

Successivamente per migliorare la parte di logica è stata creata un'ulteriore enumerazione per rappresentare le varie opzioni dei menù. A queste MenuOptions sono associati i relativi GameState da assegnare in caso di se-



lezione. Questa aggiunta serve a "snellire" notevolmente il codice nella parte di logica e grafica, dato che prima l'associazione veniva fatta all'interno delle classi stesse, causando anche duplicazione.

La parte di grafica inizialmente è stata modellata con varie implementazioni dell'interfaccia `Display`. Questo però portava a ripetizioni eccessive nel codice, perciò dopo rielaborazioni dei colleghi Pellanda e Zandonella si è giunti alla versione attuale, dove `Display` è diventata una classe astratta che possiede tutti i metodi di disegno necessari per la creazione di nuove schermate.

Successivamente è sorto un problema riguardante la possibile mancanza di determinati font su macchine diverse. Per questo è stato creato un ulteriore branch `fonts`. Qui è stata creata una classe che carica tutti i vari font direttamente dalle risorse di gioco, mettendoli così a disposizione delle classi che si occupano di grafica.

### 3.2.2 Daniel Pellanda

#### Caricamento e gestione della finestra di gioco

Nella classe `GameHandler` si istanzia un `JFrame` che definisce le "fondamenta" della finestra di gioco. Nel `JFrame` viene inserito un unico componente speciale `GameWindow` il quale si prenderà carico di gestire sia i contenuti grafici, sia l'esecuzione del videogame. `GameWindow` al suo interno implementa un thread nella quale è eseguito un ciclo infinito, detto loop di gioco, che permette la continua esecuzione del videogame. Dentro al loop sono eseguiti un determinato numero di cicli di operazioni al secondo, questo numero è chiamato "Frame per secondo" (fotogrammi per secondo, il quale implica impropriamente l'aggiornamento solo a livello grafico quando in verità è usato anche per l'aggiornamento a livello logico). Per permettere di mantenere un'esecuzione di gioco fluida e costante, è necessario gestire correttamente i momenti in cui deve essere eseguito un ciclo di operazioni fra un secondo e l'altro, per fare ciò viene definito un intervallo di tempo costante per cui il thread debba sospendersi alla fine di ogni ciclo, così facendo si riescono a suddividere equamente i momenti di attività del thread rispettando il vincolo di operazioni per secondo. Nonostante questo metodo funzioni perfettamente su Windows, non sembra essere il caso su Linux nel quale si sono riscontrati diversi problemi di fluidità di gioco. Abbiamo investigato abbastanza, ma sfortunatamente non siamo riusciti a trovare la causa di questo problema. In ciascun ciclo di operazioni (o frame) sono eseguiti due tipi principali di operazioni: `Update` e `Draw`. `Update` esegue controlli e aggiornamenti sui valori del videogame, invece `Draw` genera il nuovo fotogramma da visualizzare

nella finestra. L'inizio dell'esecuzione del thread di gioco non avviene prima di aver inizializzato tutte le componenti necessarie al videogame.

## Gestione e logica delle entità

L'implementazione di questa sezione è stata effettuata sui branch `Obstacles` e `Pickups`, separato dalla linea di sviluppo principale. Nell'interfaccia `Logics` viene tenuta una struttura dati nella quale sono contenute le entità attive nell'ambiente di gioco. Come già detto ciascuna entità è rappresentata con una propria definizione di oggetto le quali derivano tutte dall'interfaccia `Entity`. Fra gli oggetti entità sono inoltre definite 2 altri sottotipi di entità: gli ostacoli e i pickups. Gli ostacoli hanno lo scopo di intralciare il giocatore causandogli danno al contatto, i pickups invece garantiscono al giocatore dei vantaggi per aiutarlo a raggiungere un punteggio più alto. Ogni oggetto entità deve includere diversi metodi che permettono il loro controllo esterno, fra i più importanti ci sono le direttive `Reset`, `Update` e `Draw`. `Reset` richiede all'entità di reimpostare la propria posizione e valori a quelli di inizio, `Update` aggiorna e controlla i valori e `Draw` graficizza l'entità nella schermata di gioco. Questi metodi di controllo sono utilizzati in particolare dall'interfaccia `Logics` per la loro gestione run-time.

## Generazione delle entità

La generazione è gestita per set di entità, le quali verranno sempre importate nell'ambiente di gioco in gruppo. Durante l'inizializzazione del generatore, le disposizioni dei set di entità saranno reperiti da un file `tiles.json` e verranno in apposite strutture dati suddivise in base al loro tipo. Ogni entità contenuta in set è quindi sempre tenuta in memoria e pronta ad essere inserita nell'ambiente di gioco. Il tipo di entità da generare è deciso in maniera parzialmente casuale: la randomicità di generazione influenzata da dei parametri di probabilità che cambiano in base al tipo (per esempio: un ostacolo di tipo `Zapper`, essendo più comune, avrà una possibilità più grande di essere generato rispetto ad un ostacolo di tipo `Missile`). A differenza del tipo, il set delle entità da generare è invece deciso in maniera totalmente casuale. L'interfaccia `Generator`, per permettere una continua generazione di entità, implementa un thread con ciclo infinito nel quale viene effettuata una generazione ogni intervallo di tempo stabilito. Per permettere il controllo del thread sono previsti diversi metodi che permettono di mettere in pausa o riprendere l'esecuzione di esso.

## 3.3 Note di sviluppo

### 3.3.1 Giacomo Amadio

Gestione menù e cambiamento schermate:

- Il controller delle varie schermate fa uso di Consumers e Suppliers ( passati tramite lambda expressions) per ricevere e comunicare informazioni sullo stato del gioco.
- Utilizzo e inizializzazione classe Font per contenere informazioni relative ai font caricati dalle risorse di gioco.

Gestione suoni:

- Utilizzo e inizializzazione classe Clip per contenere informazioni relative a file audio (necessariamente in formato .wav) caricati dalle risorse di gioco. Spunto per la creazione della classe: <https://stackoverflow.com/a/953752>
- Utilizzo stream e lambda expressions per rilascio risorse audio di sistema (fatto manualmente poichè creava problemi su sistemi Linux) e pulizia della mappa dei suoni attualmente in riproduzione.

Lettura e scrittura impostazioni su file:

- Utilizzo libreria esterna **Json-simple-4.0.1** per lettura e scrittura su file json. Spunto per la creazione della classe: [https://www.tutorialspoint.com/json\\_simple/json\\_simple\\_quick\\_guide.htm](https://www.tutorialspoint.com/json_simple/json_simple_quick_guide.htm).

Rilevamento e gestione collisioni:

- Utilizzo e inizializzazione classe Rectangle per fornire una rappresentazione sommaria degli estremi di collisione.
- Utilizzo stream e lambda expressions per aggiornamento posizioni del gruppo di rettangoli e per controllo intersezioni fra hitbox di player e altre entità.
- Utilizzo Optional per gestire e consumare le eventuali collisioni registrate.

### 3.3.2 Daniel Pellanda

Caricamento e gestione della finestra di gioco:

- Utilizzo di thread per l'esecuzione principale di gioco.

Gestione e logica delle entità:

- Utilizzo di BiConsumers e Predicates per gestire la rimozione di entità al fuori dell'ambiente di gioco.
- Utilizzo di lambda expressions per l'inserimento di BiConsumers e Predicates.
- Utilizzo di stream per il controllo sulla priorità e il tipo delle entità.

Generazione delle entità:

- Utilizzo di thread per permettere la generazione continua di entità.
- Utilizzo di Functions e BiFunctions per la creazione delle entità.
- Utilizzo di lambda expressions per l'inserimento di Functions e BiFunctions.
- Utilizzo di stream per la gestione dei tipi di entità da memorizzare.
- Utilizzo di Optional in caso di mancanza delle Function per la creazione delle entità.
- Utilizzo della libreria esterna **Json-simple-4.0.1** per la lettura del file tiles.json contenente le informazioni sui set di entità generabili.

# Capitolo 4

## Commenti finali

### 4.1 Autovalutazione e lavori futuri

#### 4.1.1 Giacomo Amadio

Per quanto riguarda la mia parte, credo di aver suddiviso in modo consoni gli elementi fondanti dei vari problemi, moderando l'uso di feature avanzate del linguaggio ove il codice potesse beneficiarne in leggibilità. Sono tuttavia consapevole del fatto che avrei dovuto dedicare più tempo a trovare soluzioni meglio strutturate, in particolar modo mi rincresce l'aver documentato il codice in maniera molto concisa, che talvolta potrebbe lasciare il lettore con dei dubbi sul funzionamento.

Una cosa che mi sembra doveroso rimproverare al gruppo è stata la poca coesione fra i membri, che talvolta è sfociata in una totale assenza di spirito collaborativo. Infatti sono abbastanza sicuro di poter affermare che, nei mesi di sviluppo, non ci sia stato nemmeno un'occasione in cui tutti i componenti stessero lavorando su parti vitali del progetto. Questo a mio modo di vedere ha causato una scarsa uniformità nel codice, costringendo talvolta a riadattare il proprio lavoro o quello di altri per produrre soluzioni funzionanti. Per questi motivi, non ho intenzione di portare avanti il progetto in futuro, in quanto ritengo siano mancati degli elementi fondamentali per qualificare il lavoro come "professionale".

#### 4.1.2 Daniel Pellanda

Penso di aver svolto un lavoro decente con la mia parte, riuscendo a dare un forte impatto verso la riuscita (anche se discreta) del progetto. Pur non avendo sviluppato un codice eccellente, credo ugualmente di essere riuscito a costruire delle soluzioni efficaci e funzionanti. Bisogna però sottolineare di

non aver elaborato un altrettanto gran lavoro con la progettazione, la quale è stata in gran parte tralasciata durante l'intero progetto. All'inizio avevo una forte motivazione verso la realizzazione di questo progetto, però col tempo questa venne lentamente persa a causa della scarsa volontà di collaborazione di alcuni membri del gruppo. Ho visto questo progetto come un importante opportunità per acquisire esperienza nel campo della programmazione ed è veramente un peccato che ciò non sia stato il caso.

Mi duole dover affermare che il nostro progetto non vedrà luce al di fuori di questo corso, e ciò è dovuto dalla mancanza della armonia e passione necessaria al gruppo per continuare a lavorarci.

# Appendice A

## Guida utente

Nei menù:

- Freccette ”**SU e GIU**” permettono di navigare fra le varie opzioni
- Freccette ”**DESTRA e SINISTRA**” consentono di modificare a proprio piacimento le impostazioni nella sezione a loro dedicata.

In partita:

- Con la pressione prolungata della ”**BARRA SPAZIATRICE**” il personaggio inizierà a volare
- Il tasto ”**P**” mette in pausa il gioco.

Extra:

- Il tasto ”**Z**” mette a disposizione dell’utente una vista aggiuntiva, nella quale vengono fornite informazioni specifiche sullo stato attuale del gioco.

# Appendice B

## Esercitazioni di laboratorio

### B.1 Daniel Pellanda

- Laboratorio 05: <https://virtuale.unibo.it/mod/forum/discuss.php?d=87881>
- Laboratorio 06: <https://virtuale.unibo.it/mod/forum/discuss.php?d=87880>
- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=88829>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=89272>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=90125>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=91128>