# Hop-Tales

Enrico Casadei, enrico.casadei18@studio.unibo.it
Simone Capacci, simone.capacci3@studio.unibo.it
Simone Gentili, simone.gentili4@studio.unibo.it
Eleonora Bartoletti, eleonora.bartoletti2@studio.unibo.it

February 2026

# Indice



# 1 Analisi

## 1.1 Descrizione e requisiti

Il gruppo si pone come obiettivo la realizzazione di un gioco platformer chiamato Hop-Tales. Il gioco si sviluppa in 3 livelli: nel primo il giocatore si muove in avanti evitando i nemici fino a raggiungere un castello posto alla fine del livello; nel secondo livello, simile al primo, si utilizza un secondo personaggio che a fine livello raggiunge lo stesso castello; nel terzo livello, da giocare in due, il gioco diventa un puzzle-platformer, in cui i due personaggi devono risolvere dei mini-puzzle per finalmente riunirsi.

### Requisiti funzionali

- Il giocatore sarà in grado di muoversi liberamente su due dimensioni per tutti e tre i livelli.

- I primi due livelli avranno lo stesso stile da classico platformer, in cui il giocatore procede verso una direzione finchè non completa il livello, mentre il terzo livello sarà composto da una stanza predefinita contenente dei puzzle.
- Il giocatore avrà la possibilità di interagire con diversi elementi del livello che avranno particolari effetti; in particolare nei primi due livelli forniranno un potenziamento al personaggio, mentre nel terzo livello permetteranno di risolvere i puzzle al fine di completare il livello.
- Tra gli elementi interagibili nei vari livelli ci saranno anche le monete, oggetti raccoglibili che potranno essere utilizzati nel menù per fini puramente decorativi.
- Sarà presente un menù principale da cui saranno accessibili i vari livelli oltre allo shop, in cui si potranno spendere le monete raccolte per comprare delle skin per i personaggi, e alle impostazioni, che riguarderanno principalmente effetti audio.
- Saranno presenti diversi tipi di nemici in grado di ostacolare il giocatore. Essi avranno anche la possibilità di "terminare" la partita, colpendo il giocatore un certo numero di volte.

**Requisiti non funzionali**

- Il gioco dovrà funzionare senza errori su diversi sistemi operativi, tra cui sicuramente una qualsiasi distribuzione Linux.

## 1.2 Modello del Dominio

All'avvio del gioco si crea una finestra dove viene mostrato il menù principale, da cui il giocatore può scegliere di aprire lo shop (per comprare delle skin per i personaggi), le opzioni (per modificare gli effetti audio) o giocare. Scegliendo l'ultima opzione a sua volta gli verrà chiesto quale livello vuole affrontare. In seguito la finestra mostrerà il livello selezionato e all'interno di esso si troveranno diversi elementi: ovviamente ci sarà il personaggio del giocatore, in grado di muoversi liberamente e saltare, inoltre saranno presenti dei nemici, anch'essi in grado di muoversi (anche se in maniera predefinita). Il giocatore presenta i punti vita, ovvero un indice numerico che si abbassa ogni volta che l'entità subisce danni. Qualora l'indice scenda a 0 o inferiore finisce la partita. Il giocatore inoltre avrà la possibilità di raccogliere dei "Power-up", che gli permetteranno di non subire danni la prossima volta che verrà colpito. Oltre alle entità saranno presenti oggetti che avranno caratteristiche diverse tra di loro. Tutti gli oggetti hanno un determinato comportamento quando vengono toccati in base al tipo. Gli elementi sopracitati con le loro relazioni sono riassunti nella Figura 1.1.

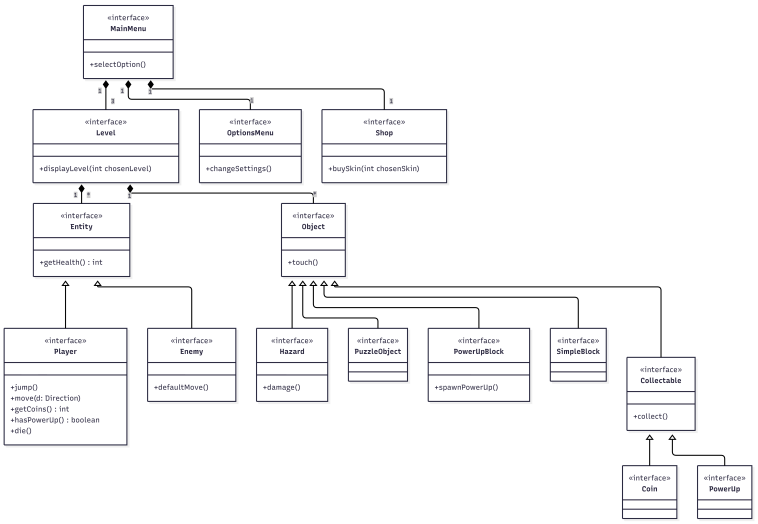
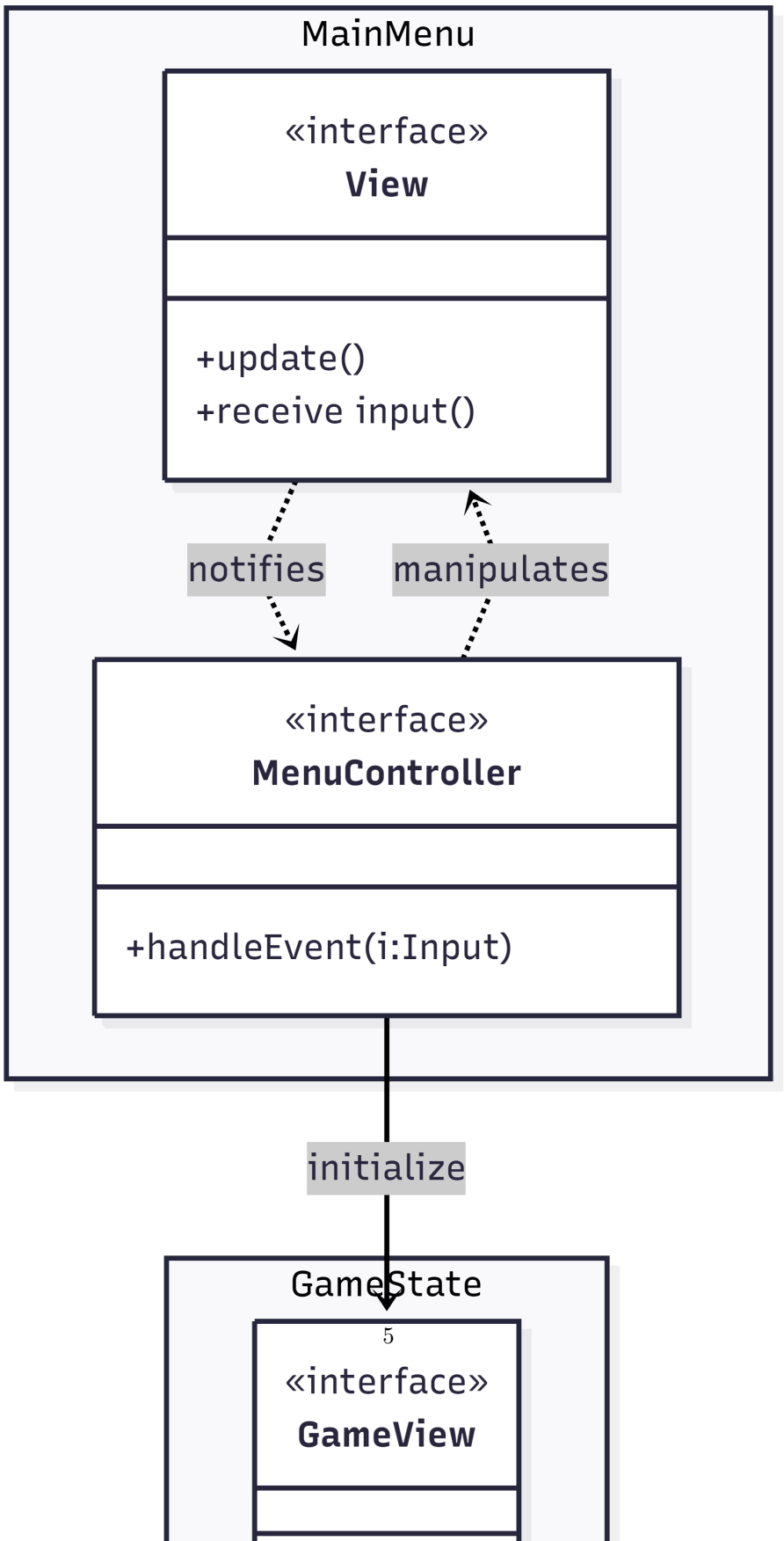



Figure 1.1: Diagramma UML dell'analisi del dominio

# 2 Design

## 2.1 Architettura

Considerando che il gioco è diviso in due sezioni decisamente diverse, ovvero la parte del menù e quella della partita, si è optato per dividere la struttura architetturale del gioco in due blocchi. Per l'architettura del menu di gioco si sfrutta una variante semplificata del pattern MVC. Lo stato del menu è infatti gestito implicitamente dal Controller, che decide quale schermata visualizzare in base agli input ricevuti. All'avvio del gioco viene creato il Menu, che rappresenta la *View* principale. Esso riceve gli input attraverso cui notifica il MenuController, che rappresenta il *Controller*, il quale gestisce i vari stati del menu che vengono modificati in base agli input dell' user. Ogni volta che si verifica il cambiamento di stato (ovvero l'utente seleziona la schermata interessata) Il MenuController si attiva chiamando direttamente la view mostrando dunque la schermata corretta. Per passare al secondo caso d'uso, ovvero lo stato di gioco vero e proprio, serve l'input del giocatore per avviare un livello, in modo tale che il Menu lo riceva, avvisi il MenuController e quest'ultimo permetta di accedere alla partita.


MainMenu
«interface»
View
+update()
+receive input()
notifies
manipulates
«interface»
MenuController
+handleEvent(i:Input)
initialize
GameState
«interface»
GameView

## 2.2 Design dettagliato

### 2.2.1 Capacci Simone

**Caricamento dei livelli tramite DTO, Mapper e Factory** Problema: Il gioco deve caricare i livelli da file esterni, descritti tramite una rappresentazione persistente (file JSON). Era necessario evitare che il modello di gioco dipendesse direttamente dalla struttura dei file. Inoltre, il sistema deve permettere l'aggiunta di nuovi livelli senza richiedere modifiche al codice del modello.

Soluzione: La soluzione adottata consente di utilizzare più file JSON per definire livelli differenti, permettendo l'aggiunta di nuovi livelli in futuro senza modificare il modello di gioco.

I file di livello vengono deserializzati dalla classe LevelLoader, che produce strutture dati intermedie prive di logica (LevelData, EntityData, MacroData). Tali classi svolgono il ruolo di DTO (Data Transfer Object) e rappresentano esclusivamente i dati necessari al trasferimento delle informazioni.

Pattern usato: DTO (Data Transfer Object) Un DTO è un oggetto che contiene solo dati, senza logica applicativa, ed è utilizzato per trasferire informazioni tra diversi strati dell'applicazione, mantenendo il modello indipendente dai dettagli di persistenza.

La conversione dai DTO agli oggetti del dominio è demandata a un componente dedicato, EntityFactory (Factory Method), che si occupa di istanziare le corrette entità di gioco a partire dai dati deserializzati, cosi che la logica di conversione è centralizzata in un unico punto.

Il Gamecontroller coordina il processo di caricamento utilizzando i DTO e il mapper per inizializzare lo stato del mondo di gioco.

Pattern usati:Factory Method, Mapper. Il Factory Method centralizza la creazione degli oggetti, nascondendo al chiamante le classi concrete utilizzate.

Il Mapper si occupa di convertire dati da una rappresentazione a un'altra, tipicamente da oggetti di trasferimento (DTO) a oggetti del dominio, mantenendo separata la logica di conversione dal modello.

Motivazione: Questa soluzione permette di mantenere il modello indipendente, centralizzare la logica di creazione delle entità, facilitare l'estensione del sistema con nuovi tipi di entità o nuovi formati di livello.

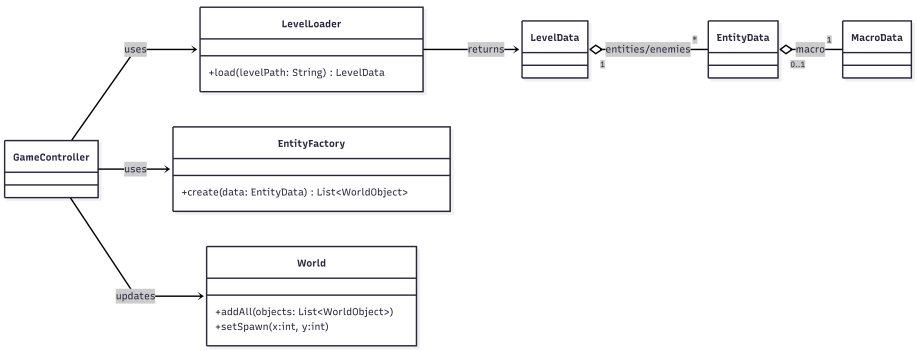



Figure 2.2: Diagramma UML del caricamento dei livelli tramite DTO e Factory

**Gestione degli stati dell'applicazione** Problema: L'applicazione presenta diverse modalità operative (menu principale, selezione del livello, negozio, opzioni e gioco), che devono essere gestite in modo coerente e esclusivo. Una gestione basata su condizioni sparse avrebbe reso il sistema difficile da estendere e manutenere.

Soluzione: Gli stati dell'applicazione sono rappresentati tramite un'enumerazione (State) che descrive in modo esplicito tutte le modalità operative. Il Controller-Menu utilizza tale rappresentazione per gestire le transizioni tra gli stati e decidere quale componente della view visualizzare.

Pattern usati: State (variante semplificata basata su enumerazioni) gli stati dell'applicazione sono rappresentati tramite un'enumerazione che rende esplicite le modalità operative possibili e consente al controller di gestire in modo centralizzato le transizioni tra gli stati.

Motivazioni: Si rendono espliciti gli stati dell'applicazione, centralizza la gestione delle transizioni e si semplifica l'aggiunta di nuove modalità operative.

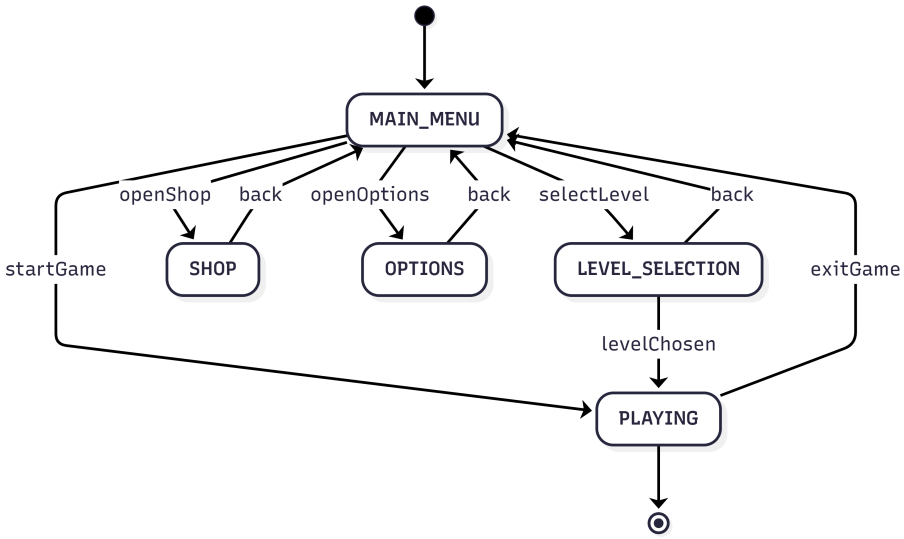



Figure 2.3: Diagramma UML della gestione degli stati dell'applicazione

**Gestione delle animazioni e delle risorse grafiche** Problema: Le entità di gioco richiedono animazioni basate su sequenze di immagini e una gestione efficiente delle risorse grafiche. Una gestione diretta delle immagini all'interno delle singole entità come inizialmente pensato avrebbe portato a duplicazione di codice e spreco di risorse.

Soluzione: La gestione delle animazioni è separata dalla gestione delle risorse grafiche: la classe Animation incapsula la logica temporale dei frame, mentre Draw delega ad Animation la selezione del frame corretto da visualizzare. Il caricamento e il caching delle immagini sono centralizzati nella classe Draw(Factory), che evita il caricamento ripetuto delle stesse risorse grafiche.

Pattern usati: Factory(per la creazione e gestione delle risorse)

Motivazione: Riduce la duplicazione del codice e migliora le prestazioni tramite caching;

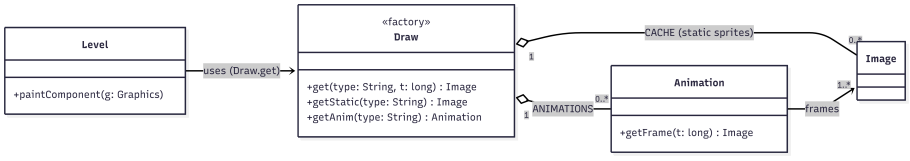



Figure 2.4: Diagramma gestione animazioni e risorse grafiche

### 2.2.2 Bartoletti Eleonora

**Gestione del Livello 3 (mappa, entità, interazioni)**

**Problema:** Il livello 3 richiede una mappa complessa con oggetti eterogenei (porte, massi, pulsanti, piattaforme mobili, teletrasporti) e una logica fisica specifica (gravità, collisioni, teletrasporto, interazioni con porte e pulsanti). È necessario mantenere il pattern MVC e non intrecciare il rendering con la logica di gioco.

**Soluzione adottata:**

- Model: 'LevelModel' mantiene lo stato del livello (mappa, entità, immagini, associazioni logiche).
- Controller: 'LevelLogic' esegue il tick di gioco, gestendo input, fisica, interazioni e condizioni di vittoria/sconfitta.
- View: 'FireboyWatergirlLevel' + 'LevelRenderer' mostrano la scena e delegano la logica al controller.

Le associazioni "porta–pulsante" e "teletrasporto–destinazione" vengono costruite una sola volta da 'LevelBuilder' usando funzioni di mapping tra tile e oggetti.

**Alternative considerate:**

- God class unica che gestisce rendering e logica: scartata perché infrange MVC e rende difficile isolare bug.
- Ogni oggetto con logica autonoma (porte, pulsanti, teleporters autogestiti): scartata per eccessiva frammentazione e difficoltà di coordinamento temporale (tick unico).

**Pro/Contro:**

- Pro: separazione netta di responsabilità, logica testabile, rendering sostituibile.
- Contro: più classi e passaggi di dati (model-controller-view) rispetto a una soluzione monolitica.

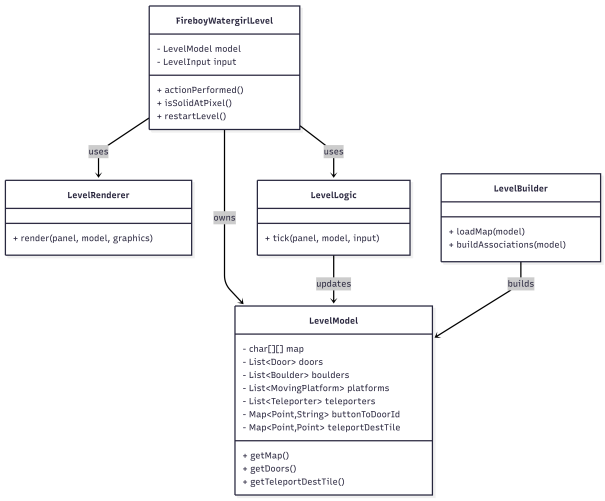



Figure 2.5: Vista d'insieme del design del Livello 3 (MVC) con costruzione iniziale delle entità.

**UML:**

**Pattern utilizzati:**

- Template Method (variante leggera): 'AbstractWorldEntity' definisce l'interfaccia comune 'draw(...)' e fornisce il metodo condiviso 'drawTiled(...)'. Le entità (Boulder, Door, MovingPlatform) specializzano il comportamento grafico. È una forma di Template Method: la "struttura" della renderizzazione è fornita dalla superclasse e le sottoclassi completano il comportamento.

**Associazioni logiche (porte, pulsanti, teleport)**

**Problema:** Nella mappa del livello 3 esistono oggetti che devono essere collegati logicamente: pulsanti che aprono porte , teleport che portano a specifiche coordinate. Il collegamento non può essere hardcoded nella logica del controller.

**Soluzione adottata:** Nel 'LevelBuilder' si costruiscono mappe:

- 'buttonToDoorId': associazione tile–id porta
- 'teleportDestTile': associazione tile–destinazione

Queste strutture vengono poi usate da 'LevelInteractions' durante il runtime.

**Alternative considerate:**

- Hardcode nel controller: scartato per bassa manutenibilità.
- Configurazione esterna (JSON): scartata per evitare modifiche strutturali al progetto.

**Pro/Contro:**

- Pro: maggiore chiarezza, logica separata e riutilizzabile.
- Contro: aggiunta di step di build (ma una tantum).

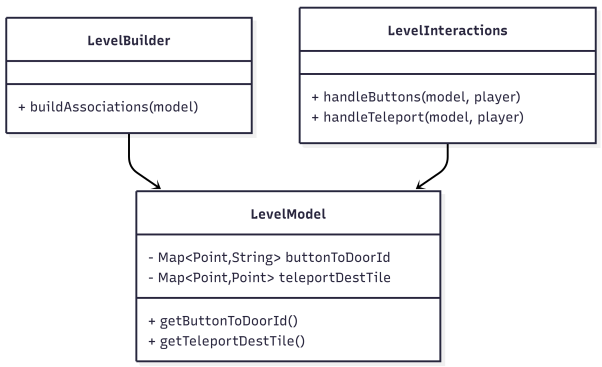



Figure 2.6: Associazioni logiche nel Livello 3: mapping tra pulsanti, porte e teletrasporti.

**UML:**

**Pattern utilizzati**

- Separazione delle responsabilità: 'LevelInteractions' centralizza le interazioni (porte, teleport) mantenendo la logica di movimento in 'LevelLogic' e il rendering nella view.

**Persistenza monete e skin (CoinStorage)**

**Problema:** Le monete raccolte e le skin acquistate devono essere persistenti fra sessioni. È necessario leggere e scrivere su file evitando duplicazioni e mantenendo compatibilità con l'economia del gioco.

**Soluzione adottata:** ‘CoinStorage‘ centralizza:

- conteggio monete
- lettura/scrittura su ‘coins.txt‘
- stato di acquisto delle skin (0/1)

Lo stato viene caricato all’avvio e aggiornato dopo ogni acquisto/collezione.

**Alternative considerate:**

- Scrivere file separati: scartato per maggiore complessità.
- Salvataggio in JSON: scartato per non introdurre dipendenze extra e mantenere semplicità.

**Pro/Contro:**

- Pro: persistenza stabile, file unico e facile da ispezionare.
- Contro: struttura rigida (3 righe per skin, ordine fisso).

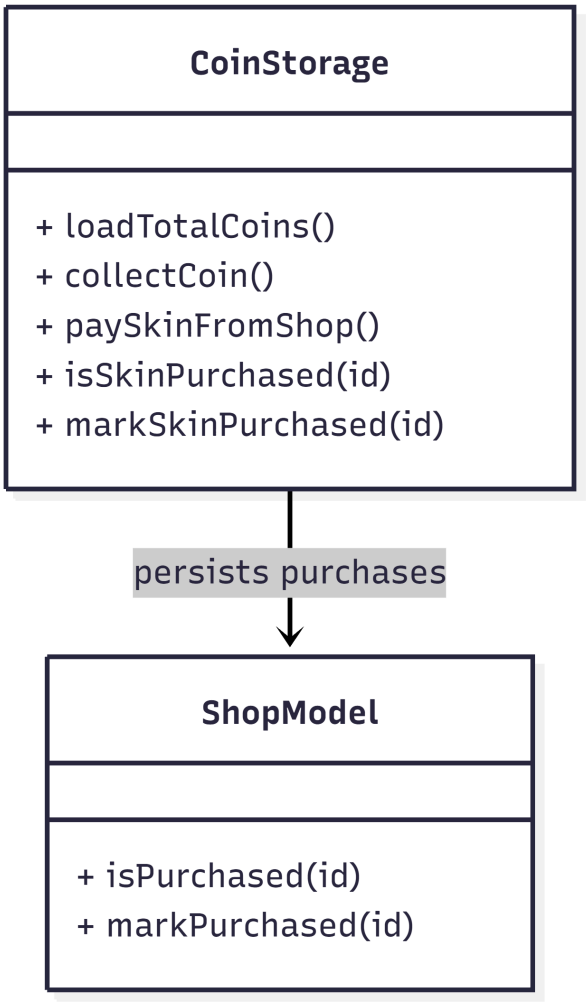



Figure 2.7: Gestione persistente delle risorse di gioco (monete e skin) e loro integrazione con lo shop.

**UML:**

**Pattern utilizzati**

- Separazione delle responsabilità: 'LevelInteractions' centralizza le interazioni (porte, teleport) mantenendo la logica di movimento in 'LevelLogic' e il rendering nella view.

**Contro-esempio UML (diagramma fatto male) :** Un diagramma "cattivo" è quello che mescola tutto in una sola classe o include dettagli implemen-

tativi inutili (metodi privati, variabili temporanee).

Esempio errato: un'unica classe 'Level3' con campi di rendering, logica, input, salvataggio monete e disegno UI.

Questo viola il principio di separazione delle responsabilità e rende impossibile capire la struttura reale.

### 2.2.3 Gentili Simone

**Struttura base dei nemici**

**Problema** I nemici presenti nel gioco appartengono a tipologie diverse, ma condividono la stessa struttura di base e gran parte del comportamento. Questo portava alla creazione di classi molto simili tra loro, con duplicazione di codice e difficoltà di manutenzione.

**Soluzione** Per risolvere il problema è stato utilizzato il *pattern Template Method*. È stata introdotta una classe astratta *AbstractEnemyImpl* che definisce lo scheletro comune del comportamento dei nemici. Il metodo template è **update()** che stabilisce l'ordine fisso delle operazioni di aggiornamento (movimento orizzontale, salto e gravità). La personalizzazione del comportamento avviene tramite:

- il metodo astratto **getSpeed()**, obbligatorio per ogni sottoclasse
- i metodi hook **moveHorizontalStep()**, **jumpStep()** e **gravityStep()** che possono essere sovrascritti dalle sottoclassi

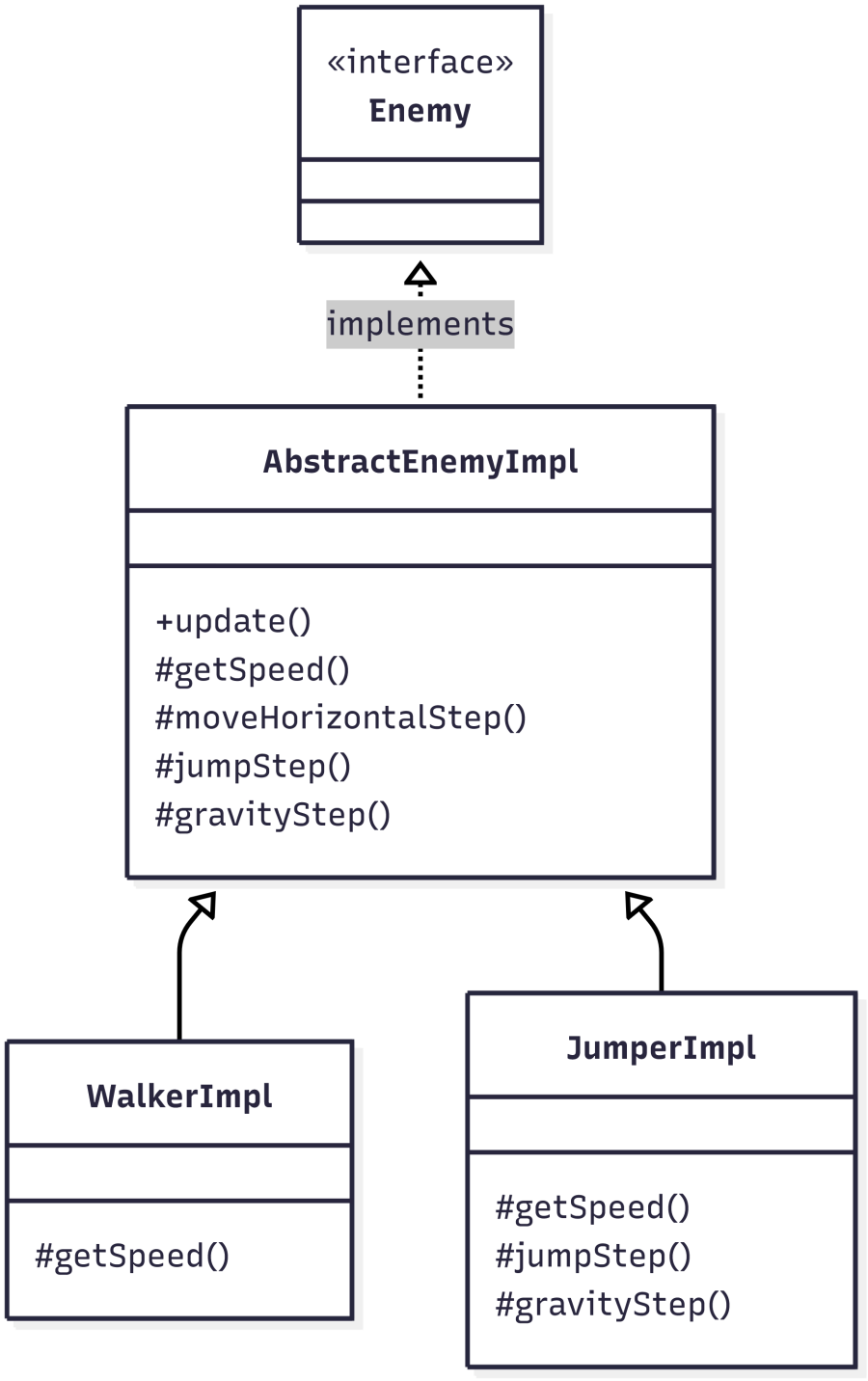



Figure 2.8: Diagramma UML della struttura dei nemici

Grazie a questa struttura l'aggiunta di un nuovo tipo di enemy è molto semplice e di facile manutenzione.

### Creazione dei nemici

**Problema** Nel gioco esistono diversi tipi di nemici, ciascuno con comportamenti e caratteristiche specifiche, che devono essere inseriti nei livelli in posizioni precise. Senza una gestione centralizzata la creazione di ogni nemico comporterebbe la duplicazione di codice e un aumento delle dipendenze tra classi, rendendo difficile la manutenzione.

**Soluzione** Per risolvere il problema è stato utilizzato il *pattern Factory Method*. È stata introdotta la classe **EnemyFactory**, responsabile della creazione degli oggetti *Enemy*. La factory decide quale implementazione concreta istanziare in base al tipo di nemico, definito dalla classe di tipo ENUM **EnemyType**, restituendo sempre un riferimento all'interfaccia *Enemy*. In questo modo la logica di creazione è centralizzata e le dipendenze dalle classi sono ridotte. Questo comporta:

- una facilità d'uso; quando c'è bisogno di creare un nemico basta richiamare il metodo **createEnemy()** della factory
- la logica di creazione dei nemici è centralizzata nella factory, evitando che altre classi debbano conoscere i dettagli delle implementazioni concrete.

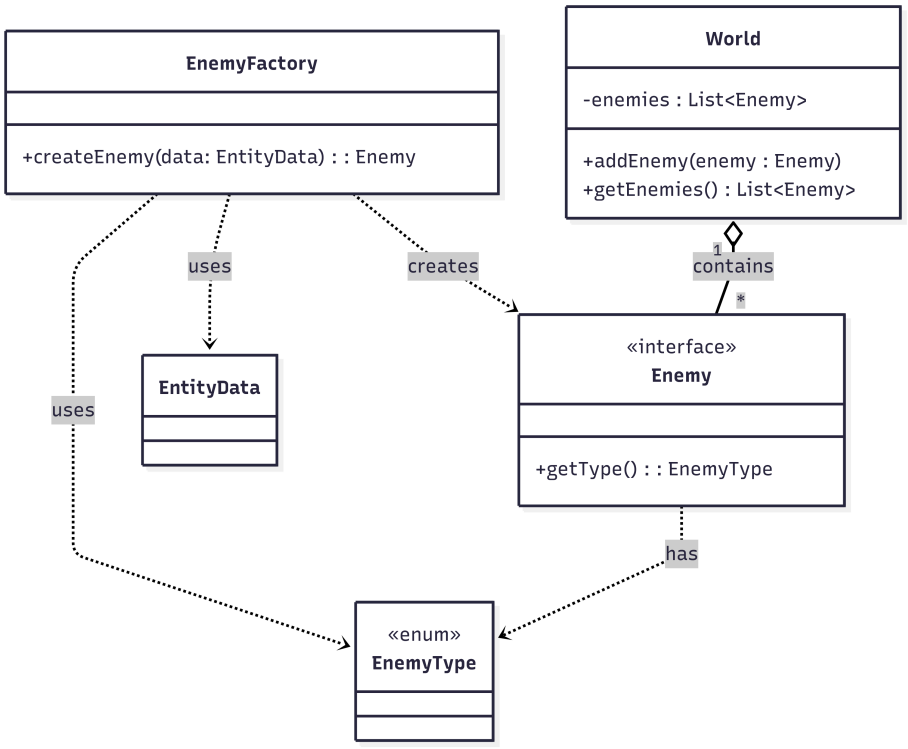



Figure 2.9: Diagramma UML della creazione dei nemici

La **EnemyFactory** rimane separata dalla (**EntityFactory**) perché i nemici e le entità statiche hanno logiche diversi; ciò violerebbe il *Single Responsibility Principle* e renderebbe il codice più difficile da mantenere ed estendere.

### 2.2.4 Casadei Enrico

**Gestione del controller in fase di gioco** **Problema** Nella fase di gioco la parte implementativa legata al Model è piuttosto corposa, di conseguenza la logica per ogni update chiamato dal Controller potrebbe creare delle classi troppo grandi con molti compiti diversi.

**Soluzione** Suddivisione della parte del Controller legata alla fase di gioco in più classi, seguendo l' *Observer pattern*. La classe GameController sarà *Observable*, mentre le classi PlayerController, EnemyController e CollectablesController saranno gli *Observers*. Il GameController si serve di un Timer che richiama ad una certa frequenza il metodo del GameController che notifica gli *Observers*. Questi ultimi quindi richiamano l'update della parte del Model che li riguarda. Per il PlayerController ho preferito gestire la logica al suo interno per non appesantire troppo la classe del Player, mentre per il CollectablesController mi sono servito di una classe helper comune ai vari *Collectables* chiamata

CollectableManager. La realizzazione è riassunta in Figura 2.10.

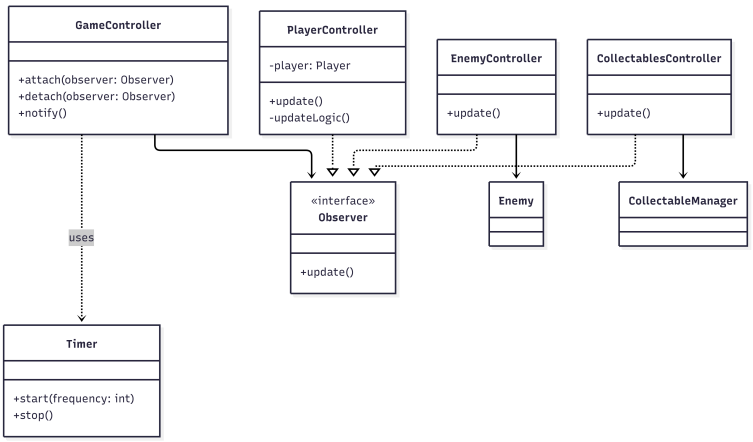



Figure 2.10: Diagramma UML del pattern *Observer* utilizzato per la rappresentazione del Controller

**Gestione delle collisioni** **Problema** Bisogna gestire ogni tipo di collisione, in maniera comprensibile e facile da riutilizzare in diverse situazioni.

**Soluzione** Creazione della classe Collider che gestisce al suo interno tutta la logica che riguarda il controllo delle possibili collisioni, rendendo l'utilizzo dei suoi metodi veloce ed intuitivo. La classe sfrutta il pattern *Facade*, in quanto fornisce alle altre classi un'interfaccia semplice tramite i propri metodi, nascondendo la logica implementativa. La realizzazione è riassunta in Figura 2.11.

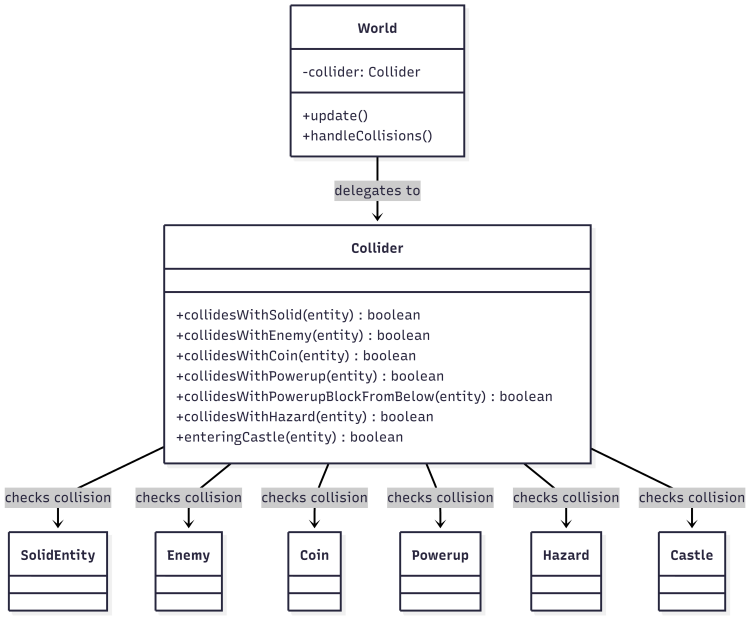



Figure 2.11: Diagramma UML del Collider e delle classi con cui interagisce

**Gestione degli effetti sonori** **Problema** Serve una funzionalità in grado di manipolare in modo semplice e intuitivo i vari file audio presenti nel gioco.

**Soluzione** Sviluppo della utility class AudioManager, che anch'essa sfrutta il pattern *Facade* per offrire una interfacciabilità comoda tramite i propri metodi statici. La classe si serve della libreria *javax.sound.sampled* per implementare pochi ma efficienti metodi per il progetto. La realizzazione è riassunta in Figura 2.12.

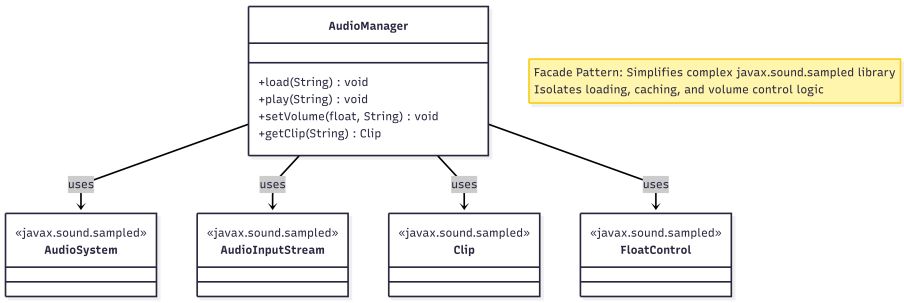



Figure 2.12: Diagramma UML del pattern *Facade* utilizzato per AudioManager

**Implementazione del Player** **Problema** Necessità di creare la classe che gestisce il giocatore, rendendola semplice e possibilmente modificabile per even-

tuali cambiamenti futuri.

**Soluzione** Sfruttare il pattern *Strategy* per creare una gerarchia di classi utilizzabili in contesti diversi che non richiedono di conoscere l'implementazione. In questo caso si ha una interfaccia generale Entity, sfruttata anche per l'implementazione dei nemici, e una interfaccia Player che la estende e viene implementata da PlayerImpl, per rendere più semplici eventuali implementazioni future più complesse. La classe PlayerImpl inoltre, come spiegato in precedenza, delega alla classe PlayerController la logica di aggiornamento del proprio stato, per ridurre la propria complessità. La realizzazione è riassunta in Figura 2.13.

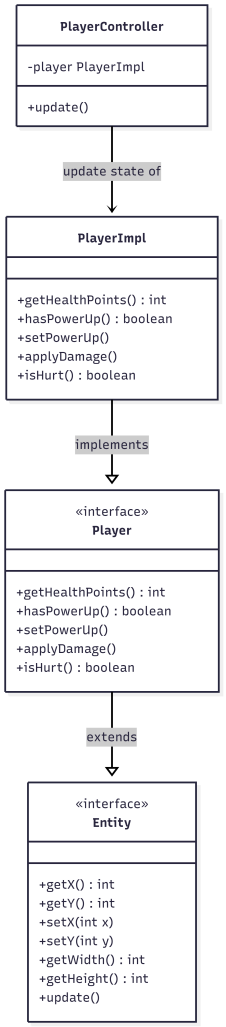



Figure 2.13: Diagramma UML del pattern *Strategy* utilizzato per Entity, Player e PlayerImpl

# 3 Sviluppo

## 3.1 Testing automatizzato

Il testing automatico serve per evitare regressioni e garantire che le funzionalità principali rimangano corrette dopo ogni modifica. Nel progetto sono stati adottati test completamente automatici basati su JUnit, senza intervento umano. I test coprono in modo minimale ma significativo il Livello 3 e la persistenza delle risorse:

- **Enemy**: nel quale viene testato il movimento dei nemici in modo correto, la correta crazione dei tipi di nemici nel livello.
- **Player**: nel quale viene testate le interazioni del player con i powerup e il danno subito.
- **Livello 3**: testiamo che la mappa venga caricata correttamente, che le entità principali vengano create (porte, massi, bottoni, teleport, piattaforme) e che le associazioni logiche (pulsante–porta, teleport–destinazione) siano consistenti con le costanti di gioco.
- **Map**: nel quale viene testata il correto caricamento delle mappe dei livelli 1 e 2, creazione delle entita al suo interno e delle interazioni nel world.
- **CoinStorage**: testiamo la lettura/scrittura da file, verificando il caricamento delle monete e dei flag di acquisto delle skin e la persistenza di un acquisto.
- **Collider**: nel quale viene testata le collisioni con nemici, powerup, monete, blocchi solidi e blocchi hazard.

In questo modo il test automatico intercetta errori tipici (mappa incoerente, associazioni mancanti, persistenza non salvata) e rende possibile rilevare regressioni in modo affidabile.

## 3.2 Note di sviluppo

### 3.2.1 Capacci Simone

**Uso di librerie di terze parti** usati in :
`https://github.com/burraco04/OOP25-hop-tales/blob/a4aaf040e954816289de773d2bdcfe55a6639e04/src/main/java/deserialization/level/LevelLoader.java#L8`

**Uso di method reference** `https://github.com/burraco04/OOP25-hop-tales/blob/a4aaf040e954816289de773d2bdcfe55a6639e04/src/main/java/view/utils/Draw.java#L39C2-L39C61`

**parte di codice riadattata da internet** FontFactory.

### 3.2.2 Bartoletti Eleonora

**Feature avanzate Java utilizzate (con esempi di codice):**

- Lambdas e interfacce funzionali: uso di 'BiFunction', 'BiConsumer' e funzioni lambda per costruire associazioni (porte, bottoni, teletrasporti) in modo dichiarativo, riducendo duplicazioni e migliorando la leggibilità.

  `https://github.com/burraco04/OOP25-hop-tales/blob/bff6094f7ce7ab1c8f222baf7a095938b35ef692/src/main/java/model/level/LevelBuilder.java#L186`

- Method reference: impiego di 'AbstractWorldEntity::getX' per ordinare le piattaforme in modo conciso e type-safe.

  `https://github.com/burraco04/OOP25-hop-tales/blob/bff6094f7ce7ab1c8f222baf7a095938b35ef692/src/main/java/model/level/LevelBuilder.java#L134`

- Generics con collezioni e comparator tipizzati: uso di 'List', 'Map', 'Comparator' e 'Deque' con tipi concreti per garantire sicurezza a compile-time e semplificare la logica del builder.

  `https://github.com/burraco04/OOP25-hop-tales/blob/bff6094f7ce7ab1c8f222baf7a095938b35ef692/src/main/java/model/level/LevelBuilder.java#L26`

- Annotazioni di terze parti (SpotBugs): utilizzo di '@SuppressFBWarnings' per documentare e controllare esposizioni intenzionali di riferimenti live nel modello MVC.

  `https://github.com/burraco04/OOP25-hop-tales/blob/bff6094f7ce7ab1c8f222baf7a095938b35ef692/src/main/java/model/level/LevelModel.java#L18`

- Graphics2D e trasformazioni affini (JDK avanzato): uso di 'Graphics2D' e 'AffineTransform' per gestire scaling e offset della scena in modo robusto e indipendente dalla risoluzione.

  `https://github.com/burraco04/OOP25-hop-tales/blob/bff6094f7ce7ab1c8f222baf7a095938b35ef692/src/main/java/view/impl/LevelRenderer.java#L28`

**Codice riadattato o preso da fonti esterne** Nessun codice copiato integralmente da risorse esterne. Le scelte di implementazione per il livello 3 sono originali; eventuali ispirazioni generali (es. gestione del loop di gioco) derivano da conoscenze pregresse e documentazione standard Java/Swing.

# 4 GENTILI SIMONE

**Uso di lambda expressions** utilizzati ad esempio in:
https://github.com/burraco04/OOP25-hop-tales/blob/e8d6cd9bf68d214d4bf5f25293e49066bc76d247/src/main/java/controller/impl/EnemyController.java#L31

**Uso di librerie di terza parte(SpotBugs / FindBugs)** https://github.com/burraco04/OOP25-hop-tales/blob/e8d6cd9bf68d214d4bf5f25293e49066bc76d247/src/main/java/model/entities/impl/AbstractEnemyImpl.java#L3
https://github.com/burraco04/OOP25-hop-tales/blob/e8d6cd9bf68d214d4bf5f25293e49066bc76d247/src/main/java/controller/impl/EnemyController.java#L4

# 5 Commenti finali

## 5.1 Autovalutazione e lavori futuri

### 5.1.1 Capacci Simone

All'interno del gruppo mi sono occupato principalmente della progettazione e dell'implementazione delle componenti legate al caricamento dei livelli primo e secondo, In particolare la creazione di tiles e della struttura intera della mappa relativa al primo livello. alla gestione delle risorse grafiche e delle animazioni, nonché di parti legate interfaccia utente (menu , negozio(creazione di skin e possibilità di comprarle), opzioni).

Non sono completamente soddisfatto del mio lavoro in quanto per mancanza di tempo non sono riuscito a modellare lo shop e la gestione skin come volevo per mancanza di tempo. Nonostante ciò, penso sia stata un ottima esperienza formativa che mi ha permesso di consolidare e migliorare le mie competenze di programmazione.

### 5.1.2 Bartoletti Eleonora

La realizzazione di questo progetto per me è stata rivelatrice: in particolare, mi ha aiutato a capire che il lavoro di gruppo è fondamentale, e che lo sarà sempre di più nel mondo del lavoro. Mi è servito a realizzare che il teamworking è una cosa su cui devo ancora lavorare, ma sono convinta di essere nella giusta strada. Un gruppo unito e che si aiuta a vicenda ha sicuramente una riuscita migliore del lavoro del singolo. Detto questo, ritengo che il gruppo nella totalità abbia lavorato molto bene, nonostante le normali e, molte volte necessarie, divergenze riguardo il da farsi.

### 5.1.3 Gentili Simone

Mi sono occupato principalmente della progettazione dei nemici, della loro cortteta creazione e movimento allinterno dei livello 1 e 2. Ho inoltre lavorato alla totale struttora del livello 2 con i propri oggeti e alla correta visione del player e degli oggeti allinterno dei primi due livelli durante la fase di gioco.