# Relazione del progetto “Jumpig” per “Programmazione ad Oggetti”

Filippini Francesco
Verna Alessandro
Zattoni Francesco

# Indice

# Capitolo 1

# Analisi

## 1.1 Requisiti

L'applicazione, denominata "Jumpig", mira allo sviluppo di una rivisitazione del famoso gioco mobile "Doodle Jump"[1]. Il nome del nostro gioco riassume in una sola parola l'idea di base del gioco stesso: il protagonista è un maialino che dovrà saltare da una pedana a un'altra, cercando di non cadere, di schivare i nemici e di raccogliere il maggior numero di monete.

### Requisiti funzionali

- Il protagonista dovrà interagire con tutti gli elementi di gioco: dovrà raccogliere correttamente le monete sparse in tutto il mondo di gioco; dovrà evitare i vari nemici; soprattutto dovrà saltare coerentemente alla fisica del gioco sulle pedane che saranno di vario tipo e avranno una diversa interazione con il personaggio.
- L'utente dovrà essere in grado di gestire lo spostamento orizzontale del personaggio, quindi spostarlo verso destra o verso sinistra. Invece, lo spostamento verticale è dato esclusivamente dal rimbalzo del personaggio sulle pedane.
- La partita dovrà terminare quando il personaggio cade oppure quando termina le sue vite, perse a causa dei nemici.
- Dal menu sarà possibile accedere alla classifica dei migliori punteggi raggiunti.
- L'applicazione dovrà essere in grado di gestire il movimento della camera[2], che dovrà seguire in modo coerente il movimento del personaggio nel mondo di gioco.

[1] https://it.wikipedia.org/wiki/Doodle_Jump
[2] Con camera si intende la vista del mondo di gioco.

### Requisiti non funzionali

- L'applicazione dovrà gestire in modo efficiente la creazione continua degli elementi del gioco, potenzialmente infinito.

## 1.2 Analisi e modello del dominio

Dal menu sarà possibile accedere ai migliori punteggi della classifica oppure iniziare una nuova partita. Durante la partita il gioco dovrà gestire lo spostamento orizzontale del personaggio causato dall'utente, il termine della partita e il punteggio corrente. Ogni partita si svolge in un mondo di gioco, caratterizzato dalla gravità e dalle entità presenti. Al suo interno sono presenti le diverse entità del gioco, ognuna con una propria posizione nel mondo di gioco e una propria forma, ossia la hitbox: il personaggio giocabile (il maialino), le piattaforme, i nemici e le monete. Le difficoltà saranno la gestione delle interazioni fra le entità coerentemente alla fisica del mondo e la gestione della logica dello spostamento della camera in modo da seguire correttamente il personaggio.
Il mondo di gioco dovrà gestire la creazione dinamica delle entità, ossia una generazione che dipende dalla progressione nella partita.
Il requisito non funzionale riguardante l'efficienza nella creazione delle entità del gioco richiede uno studio più dettagliato sull'uso delle risorse disponibili che non potrà essere effettuato nel monte ore previsto.

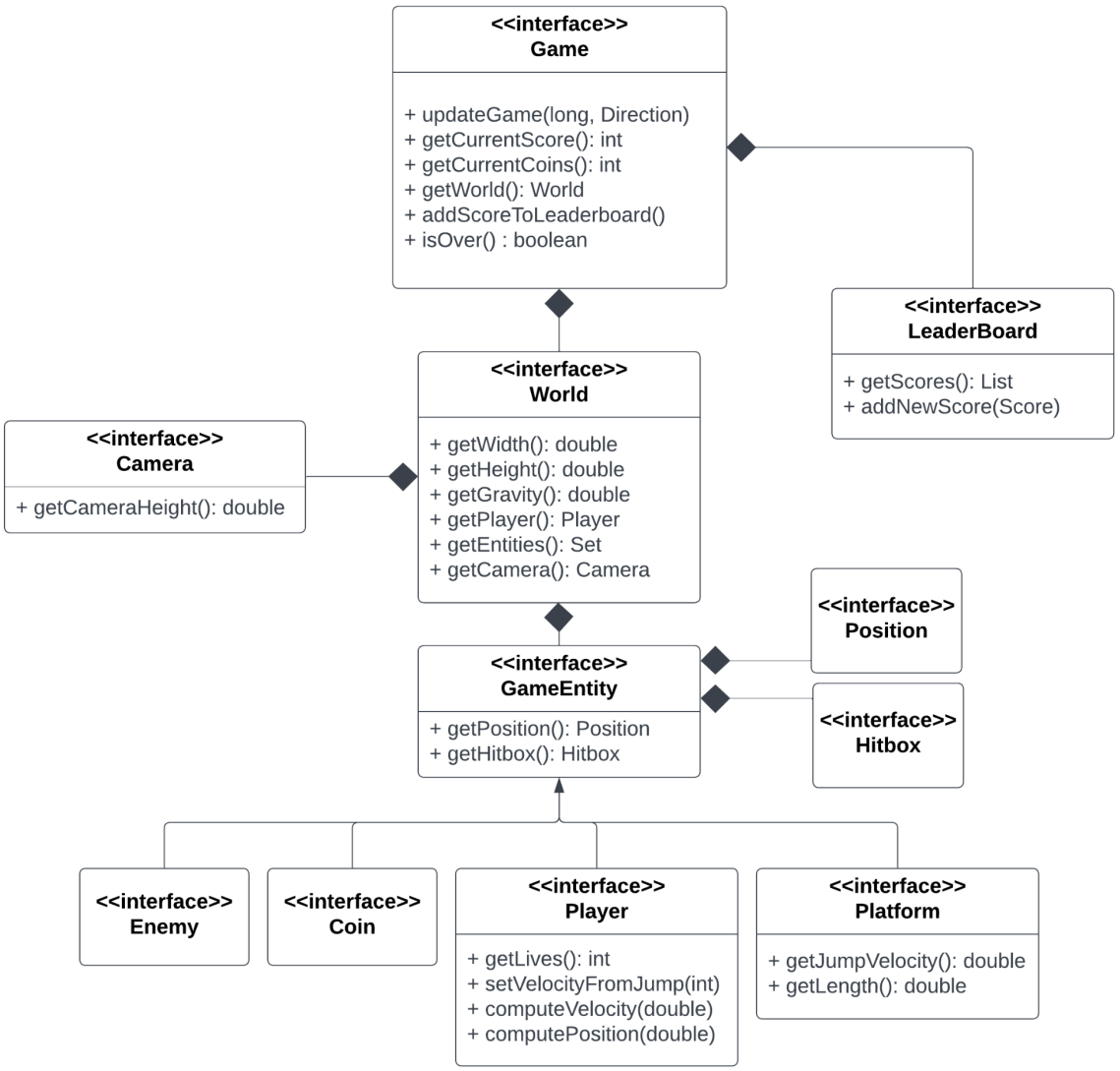


Figura 1.1: Schema UML dell'analisi del problema, con le entità principali e i rapporti fra loro.

# Capitolo 2

# Design

## 2.1 Architettura

L'architettura di Jumpig segue il pattern architetturale MVC. Il punto di ingresso dell'applicazione è il MenuController il quale gestisce la schermata principale che offre la possibilità di iniziare una nuova partita o di visualizzare la classifica dei migliori punteggi realizzati. A seconda della scelta effettuata passerà il controllo a GameController o LeaderboardController. I tre controller citati in precedenza implementano l'interfaccia Controller, la quale permette ad ognuna delle tre specializzazioni la gestione dell'avvio e della chiusura della propria sezione dell'applicazione.

- GameController si occuperà di comunicare l'input dell'utente al model (Game) e di notificare alla view (GameViewScene) l'aggiornamento delle entità e il termine della partita.
- LeaderboardController avrà come compito la gestione della visualizzazione dei migliori punteggi nella view (LeaderboardView).

GameViewScene, LeaderboardView e MenuView sono tre interfacce implementabili senza dover apportare modifiche ad elementi della parte di Model e di Controller, come si può vedere dalla figura 2.2.

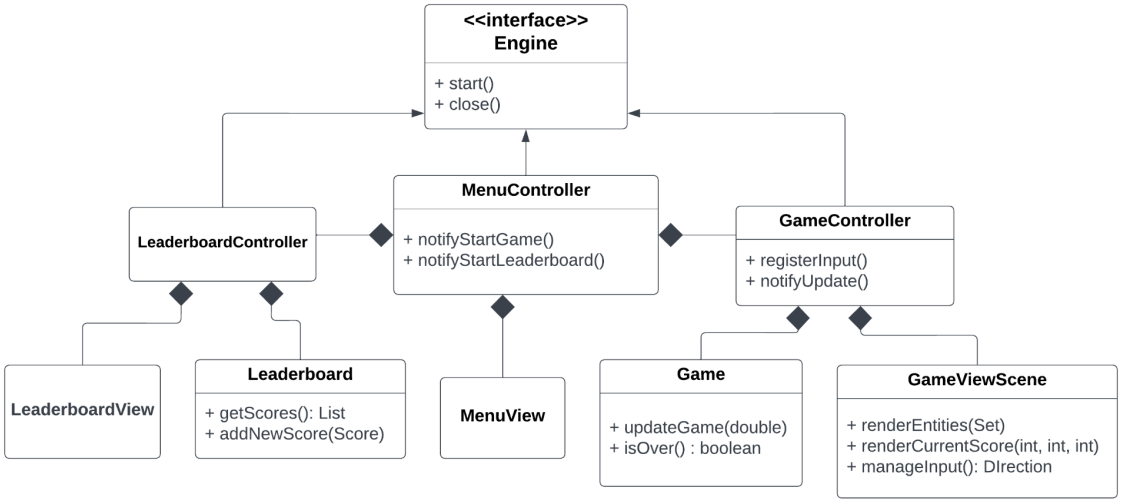


Figura 2.1: schema UML del pattern architetturale MVC scelto.

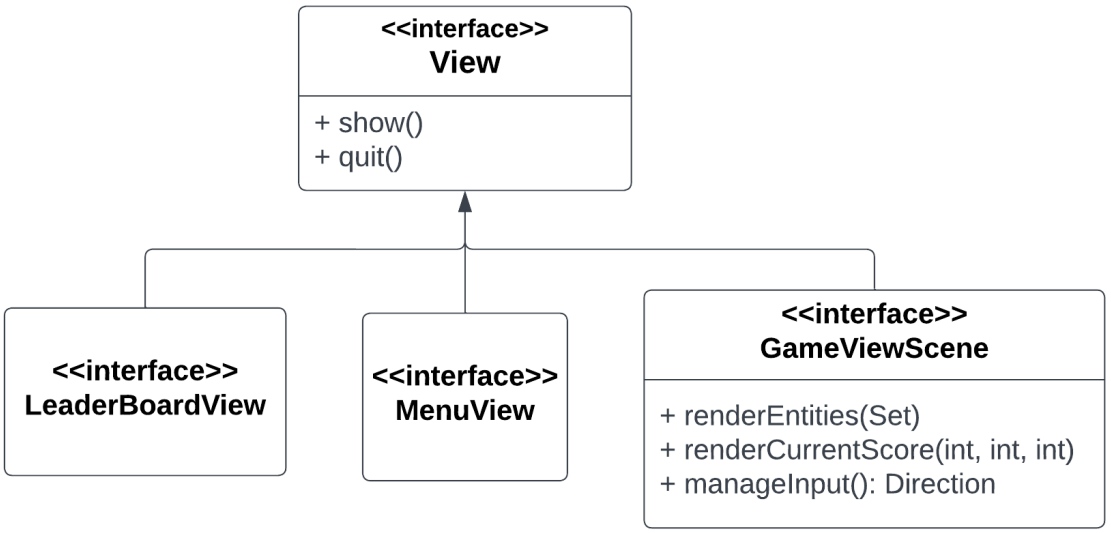


Figura 2.2: schema UML delle interfacce principali della parte di View nell'architettura MVC.

# 2.2 Design Dettagliato

## Francesco Zattoni

### Le interazioni tra le entità del gioco

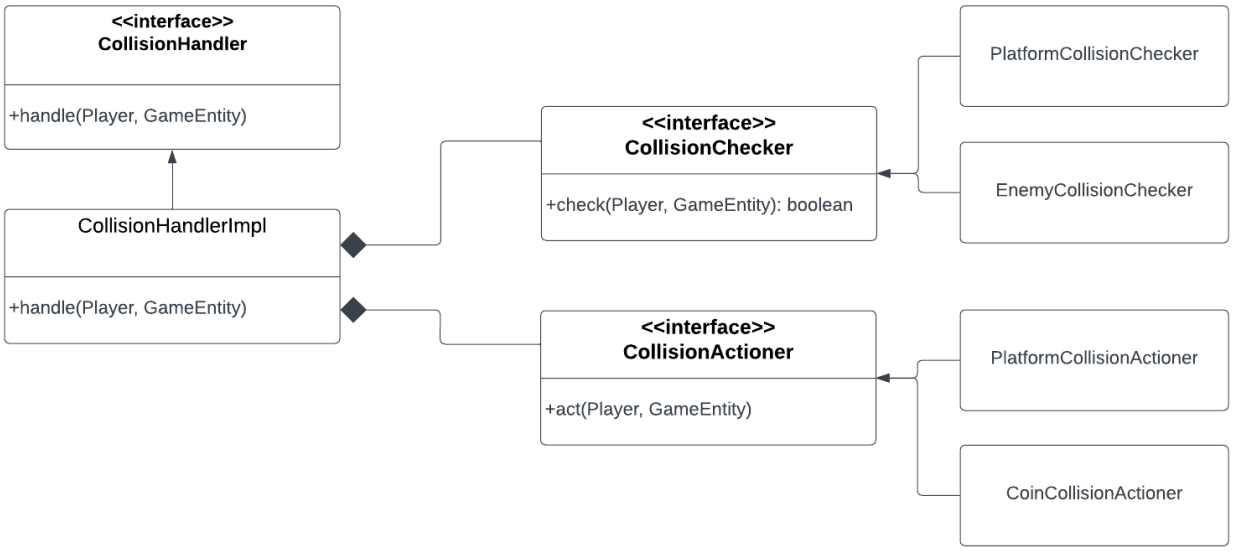


Figura 2.3: schema UML dello Strategy pattern utilizzato sia per il controllo della collisione che per gli effetti causati da essa.

**Problema:** una delle questioni da risolvere è la gestione delle interazioni del personaggio comandato dall'utente con le altre entità presenti nel mondo di gioco.

**Soluzione:** mi è stato più facile risolvere questo problema con le seguenti domande: quando c'è una collisione? E se dovesse esserci, cosa accadrebbe al giocatore e all'entità con cui ha colliso? La prima domanda implica la necessità di un controllo delle collisioni, mentre la seconda riguarda gli effetti causati da tale collisione. Perciò ho deciso di usare il pattern Strategy per creare due famiglie di algoritmi: una per il controllo, CollisionChecker, e l'altra per l'azione a seguito della collisione, CollisionActioner. In questo modo il CollisionHandler è completamente indipendente dalle implementazioni delle due strategy. Delegando il compito del controllo e degli effetti di una collisione a implementazioni esterne, in futuro, per avere un nuovo CollisionHandler per una nuova

entità del gioco non sarà necessario creare una sottoclasse di CollisionHandler, ma sarà sufficiente creare delle implementazioni di CollisionChecker e di CollisionActioner. Nell'UML non sono presenti tutte le sottoclassi, ma solo una parte.

## Il controllo delle collisioni

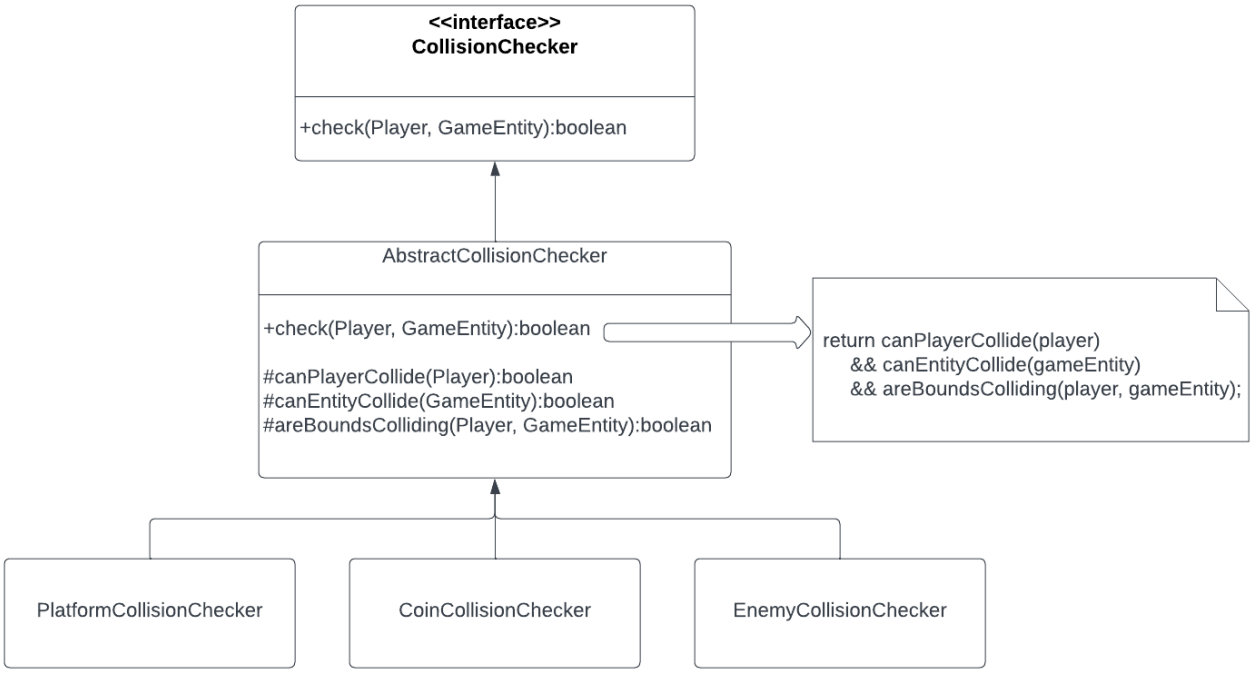


Figura 2.4: schema UML del Template Method pattern per il controllo delle collisioni tra le entità del gioco.

**Problema:** una volta trovata una soluzione alla questione precedente rimane il problema del controllo. La collisione si verifica solo a seguito di alcune condizioni, che variano a seconda dell'entità da controllare.

**Soluzione:** inizialmente avevo pensato a una Factory che creasse diversi CollisionChecker, ognuno con un proprio controllo a seconda dell'entità. Quest'idea non mi ha convinto, perché veniva così a formarsi una classe enorme che non permetteva una facile estensione e aveva ben più di una ragione per essere modificata. Perciò violava sia *open-closed principle* che *single responsibility principle.* Ho deciso di usare il pattern Template Method, poiché il controllo è diviso in tre fasi distinte: controllo dello stato del giocatore, controllo dello stato dell'entità e controllo delle

intersezioni tra il giocatore e l'entità. Quindi il metodo template, ossia *check*, definisce l'ordine dei controlli, garantendo che la sequenza non venga modificata. In questo modo l'ultimo controllo, il più oneroso, viene eseguito solo se necessario. Questa soluzione mi sembra migliore rispetto a quella iniziale, poiché in questo modo ogni sottoclasse ha un proprio compito ed è possibile creare nuove sottoclassi per controlli con nuove entità. Il contro è il proliferare delle classi, poichè ho dovuto creare alcune sottoclassi di AbstractCollisionChecker da poche righe.

# Francesco Filippini

## Generazione delle entità di gioco

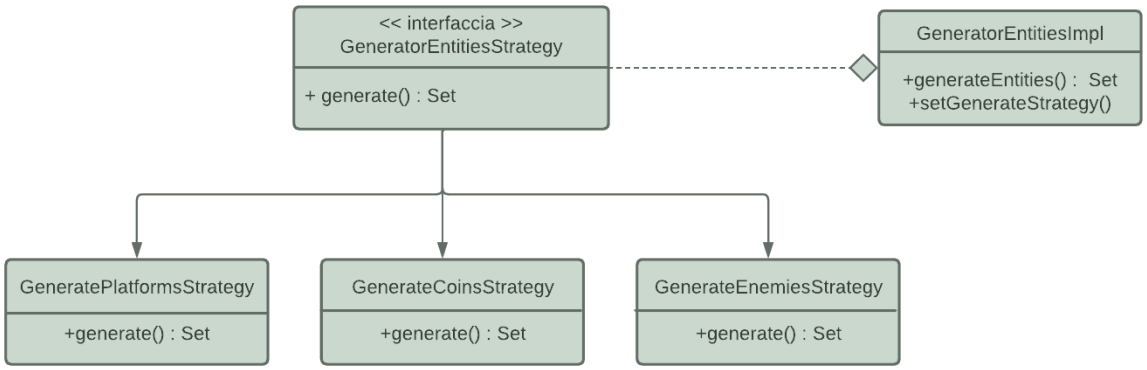


Figura 2.5: schema UML dello strategy design pattern utilizzato.

**Problema:** uno dei problemi principali è la gestione della creazione e rigenerazione continua delle entità di gioco.

**Soluzione:** Inizialmente ero partito realizzando, tutto nella stessa classe, un metodo per ogni entità da creare che restituiva un set di quella entità. In fase di refactoring però ho notato la ridondanza logica che c'era nell'implementazione dei diversi metodi che facevano le stesse operazioni e differivano solo nelle istanze che creavano. Ho deciso così, per evitare di violare il principio DRY (Don't Repeat Yourself) e per essere più chiaro, di creare un'interfaccia che incapsuli la strategia di generazione: GeneratorEntitiesStrategy . Nel fare ciò ho sfruttato lo strategy design pattern in modo da poter avere diverse implementazioni della strategia di

generazione (nel mio caso le implementazioni sono GeneratePlatformsStrategy, GenerateCoinsStrategy e GenerateEnemiesStrategy come si vede dallo schema UML in figura 2.5). In questo modo da “fuori” cioè dalla classe di contesto che è quella in cui devo generare le entità (ossia da GeneratorEntitiesImpl) basterà settare la strategia scelta e chiamare il metodo generate().
Questa scelta ha reso sicuramente più chiaro ed estensibile il codice dell’applicazione. Infatti è più chiaro perchè chiunque conoscendo l’UML del pattern noto si può già immaginare come verrà utilizzato; è più estensibile perchè se per qualche motivo un giorno si vorrà generare un set di una nuova entità allora basterà creare una nuova classe che sarà anch’essa un’implementazione dell’interfaccia che rappresenta la strategia di generazione.

## Gestione della camera

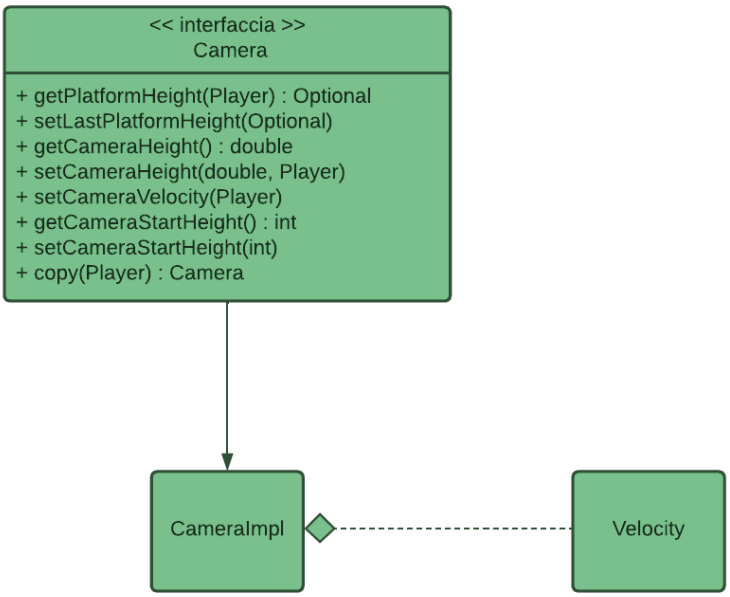


Figura 2.6: schema UML corrispondente

**Problema:** un'altra questione importante da gestire è il movimento della camera coerente non solo alla rigenerazione continua ma anche alla fisica del mondo di gioco.

**Soluzione:** Per quanto riguarda la gestione coerente del movimento della camera con la rigenerazione continua delle entità ho deciso di generare due schermi del mondo di gioco ogni uno visualizzato. Mi spiego meglio: all'inizio del gioco vengono generati due schermi del mondo e parte il gioco. Quando il personaggio arriva all'altezza massima del primo schermo si inizia a vedere il secondo schermo (che è già stato generato in precedenza quindi non ci sono problemi) e contemporaneamente vengono rigenerati altri due schermi che sono il terzo e il quarto. Poi quando il personaggio arriverà all'altezza massima del terzo schermo rigenero il quinto e il sesto e così via.
In questo modo nonostante la camera si sposti verso l'alto coerentemente con il movimento del personaggio non ci saranno mai problemi (pezzi di schermo in alto vuoti perché non generati).
Sistemato questo aspetto implementativo ho dovuto pensare ad un modo per far muovere la camera coerentemente con il movimento del personaggio evitando che fosse "scattosa" cioè che si muovesse a scatti. Per fare ciò ho tenuto un campo Velocity dentro la classe CameraImpl (come mostrato in figura 2.6) seguendo l'Item 16 di Effective Java che dice di preferire la composizione all'ereditarietà. Quindi ho dato una velocità alla camera che è la stessa velocità che ha il personaggio quando questo si muove verso l'alto. Quando il personaggio invece di salire verso l'alto cade, cioè si muove verso il basso, la velocità che ho dato alla camera è zero perchè questa non deve muoversi verso il basso (dato che se il personaggio va sotto l'altezza della camera "muore", cioè si verifica la condizione di GameOver).
A questo punto la camera si muove di moto rettilineo uniforme con velocità e posizione aggiornata ad ogni frame della partita.

# Alessandro Verna
## Logica del giocatore (Player)

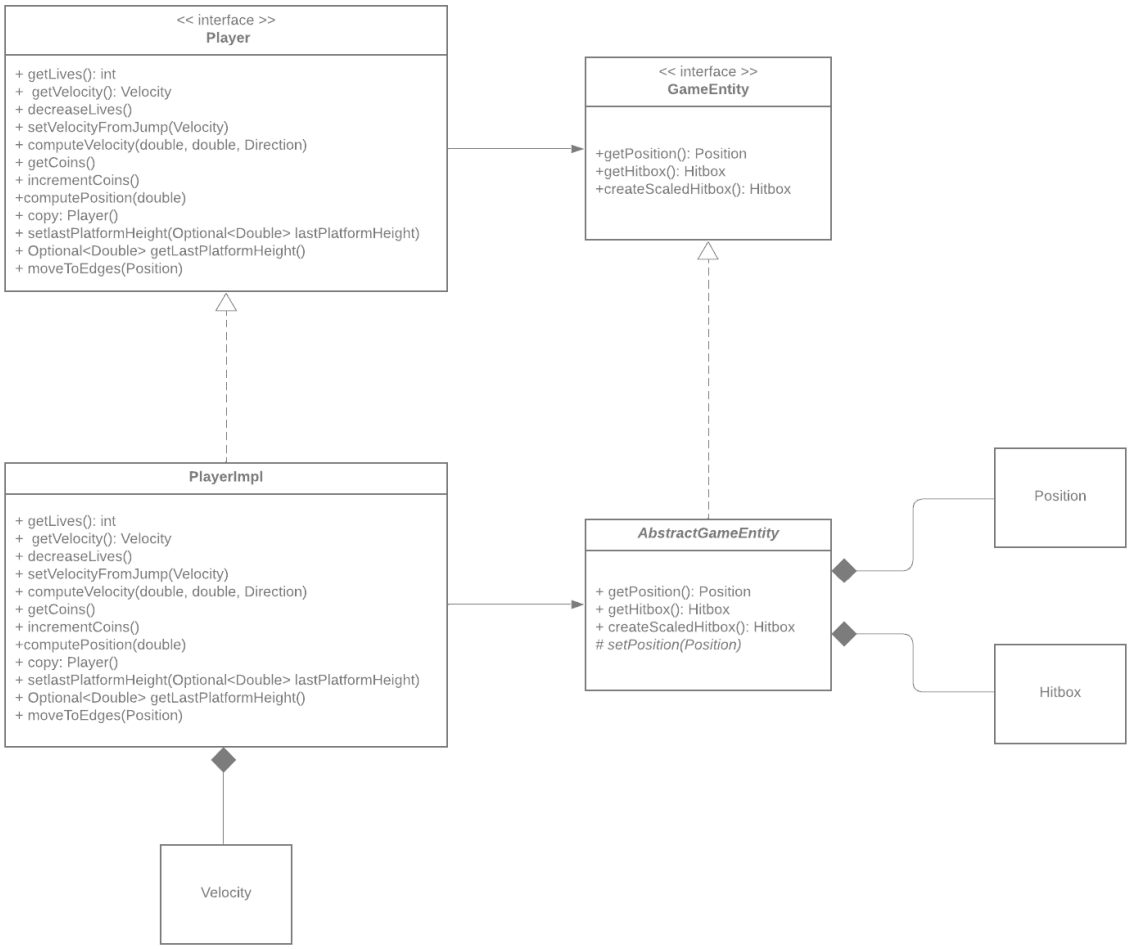


Figura 2.7: schema UML corrispondente alla logica del giocatore (player)

**Problema:** Uno degli aspetti principali del gioco è la logica del giocatore (Player) del videogioco.

**Soluzione:** Come prima cosa mi sono chiesto: quali aspetti logici dovrà possedere il Player? La risposta risiedeva su un oggetto che potesse far muovere altri oggetti (nel mio caso il Player).
Perciò ho iniziato prima di tutto con la creazione di una classe *Velocity* che si occupasse di gestire la velocità, aggiunta come campo alla futura classe *PlayerImpl*, seguendo l'Item 16 di Effective Java, che preferisce di usare la composizione rispetto all'ereditarietà.

Una volta realizzato tale aspetto, mi sono concentrato sulla realizzazione vera e propria del *Player*.
Per fare ciò, mi sono servito dell'interfaccia *GameEntity* e della sua implementazione *AbstractGameEntity*.
Esse definiscono comportamenti comuni a qualsiasi entità del gioco.
Inizialmente pensai di creare la classe *PlayerImpl* indipendentemente da *AbstractGameEntity* (classe astratta che si occupa principalmente della creazione di hitbox), ma mi sono subito accorto che alcune implementazioni di metodi venivano ripetute e di conseguenza si violava il principio DRY (Don't Repeat Yourself).
Perciò, oltre a rispettare il contratto dell'interfaccia *Player*, la classe *PlayerImpl* estende la classe *AbstractGameEntity*: in tal modo si possono delegare metodi comuni alle entità di gioco in un'unica sopraclasse, come ad esempio getPosition e getHitbox.

## Creazione dei punteggi e della classifica

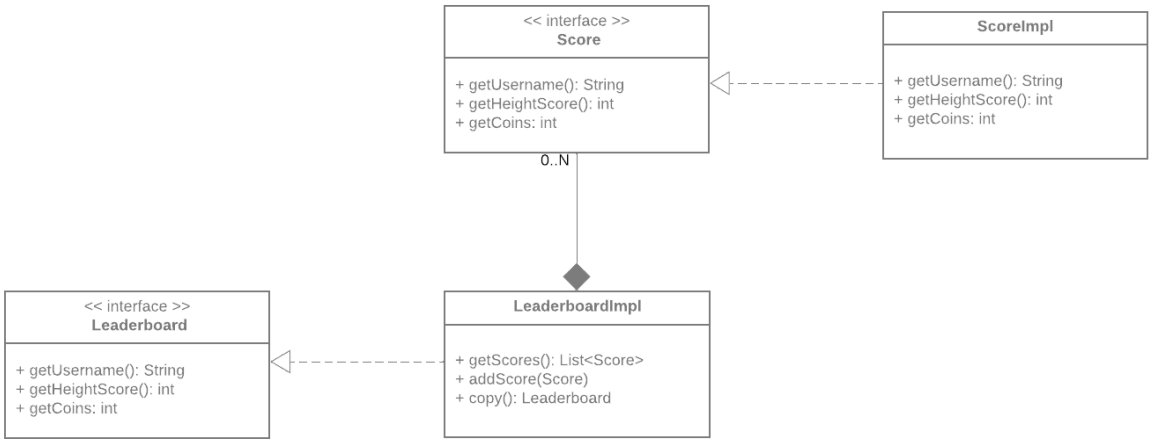


Figura 2.8: schema UML della creazione di punteggi e classifica

**Problema:** un altro aspetto principale del gioco è proprio il calcolo del punteggio di un giocatore, e il salvataggio di questo all'interno di una classifica.

**Soluzione:** Inizialmente si è pensato di creare solamente l'aspetto di Leaderboard, ma ciò comportava diversi problemi che potevano compromettere la struttura di gioco: ad esempio si violava il SRP (Single

Responsibility Principle), dato che la Leaderboard si occupa solamente di aggiungere Score e di classificarli.
Quindi sono partito innanzitutto con la realizzazione della classe *ScoreImpl* che implementa l'interfaccia *Score*.
Si basa su un semplice modello di punteggio, caratterizzato da un nome-utente (username), da un punteggio totale in base all'altezza a cui è arrivato il Player (heightScore) e dal numero totale di monete raccolte (coins).
Conclusa la creazione di tale aspetto, mi sono concentrato sull'aspetto del *Leaderboard*.
Tale modello viene caratterizzato da una lista di *Score* in modo tale da rendere più agibile l'aggiunta di un nuovo punteggio all'interno della classifica, e da uno stream all'interno del metodo getScores() per ordinare gli *Score* ed evitare ripetizioni di username prima di restituirli al chiamante.

# Capitolo 3

# Sviluppo

## 3.1 Testing Automatizzato

Abbiamo creato diverse classi che usano la libreria JUnit 5 per il test automatizzato:

- PlatformCollisionHandlerTest:
  Questa classe esegue vari test per verificare la correttezza delle collisioni tra il Player e le sottoclassi di Platform in diverse situazioni.
- CoinCollisionHandlerTest:
  Questa classe esegue vari test per verificare la correttezza delle collisioni tra il Player e le sottoclassi di Coin.
- EnemyCollisionHandlerTest:
  Questa classe esegue vari test per verificare la correttezza delle collisioni tra il Player e le sottoclassi di Enemy.
- GeneratorEntitiesTest:
  Questa classe esegue diversi test per verificare la correttezza della generazione di tutte le entità del mondo di gioco.
- PlayerTest:
  Questa classe esegue differenti test per verificare il corretto comportamento del Player quando subisce un cambio di spostamento, velocità, aumento delle monete e diminuzione delle vite.
- LeaderboardTest:
  Questa classe esegue vari test per verificare la correttezza nel salvataggio di Score in classifica (Leaderboard).

## 3.2 Metodologia di lavoro

Abbiamo cercato il più possibile di seguire un approccio “top-down”, dedicando molto tempo inizialmente all’analisi e al design del dominio e iniziando solo successivamente a scrivere codice. Quindi abbiamo creato la repository del DVCS Git e ci siamo divisi le interfacce principali da cui iniziare, ognuno in un proprio branch. Questi branch iniziali sono stati utilizzati per la creazione di tali interfacce. Una volta poste le basi, ognuno ha lavorato in un proprio branch, identificato dalle feature da sviluppare.
Sicuramente non ha aiutato l’abbandono di un membro del gruppo a poco più di un mese prima della deadline, causando la mancanza di alcune funzionalità: la presenza di nemici di diverso tipo e comportamento e il salvataggio permanente della classifica.
Durante lo sviluppo, c’è stata qualche leggera modifica al design architetturale, vista la presenza di problemi non identificati in fase di design, come ad esempio la relazione tra Game e Leaderboard, inizialmente progettata in modo diverso. Possiamo concludere che l’approccio utilizzato è “a spirale”.

### Francesco Zattoni

Ho lavorato per lo più nel branch *interactions-and-platforms* in cui mi occupavo delle collisioni e delle piattaforme del gioco. Mi è stato utile il DVCS in un paio di situazioni in cui ho dovuto correggere errori commessi diversi commit prima. Allora ho creato un branch per ogni errore significativo da correggere: *fix-collision*, per correggere un problema nell’effettivo controllo delle collisioni durante la partita e *refactoring-collision-handler* per usare correttamente il pattern Strategy.

- Classi e interfacce del package it.unibo.jumpig.model.api.collision e del package it.unibo.jumpig.model.impl.collision.
- Interfaccia Platform e le classi che la implementano.
- GamePanel in it.unibo.jumpig.view.impl.
- Maggior parte di SwingRenderer in it.unibo.jumpig.view.impl.
- Quattro metodi private per l’aggiornamento delle collisioni in WorldImpl del package it.unibo.jumpig.model.impl.

## Francesco Filippini

Ho lavorato per lo più nel branch camera-coin-gameloop in cui mi sono occupato di: gestione del movimento della camera che deve seguire il personaggio coerentemente alla fisica del gioco, generazione e rigenerazione continua delle entità del mondo di gioco, gestione delle monete. Elenco di seguito le classi principali realizzate:

- Classi CircleHitbox, RectangleHitbox, CoinHitbox e EnemyHitbox del package it.unibo.jumpig.common.impl.hitbox
- Classe GameControllerImpl del package it.unibo.jumpig.controller.impl
- Classi CameraImpl, GenerateEnemiesStrategy, GenerateCoinsStrategy, GeneratePlatformsStrategy, GeneratorEntitiesImpl, la maggior parte di WorldImpl e GameImpl del package it.unibo.jumpig.model.impl
- Classe BasicCoin del package it.unibo.jumpig.model.impl.gameentity
- Classe GameViewImpl del package it.unibo.jumpig.view.impl

## Alessandro Verna

Ho lavorato principalmente nel branch *player-menu-score* in cui mi sono occupato di: modellare il player che rispetti coerentemente la fisica del gioco, gestione del menu di gioco e creazione di Score (con successivo salvataggio nella Leaderboard). Ecco un elenco delle principali classi e interfacce realizzate:

- Classe PositionImpl ed Enum Direction nel package it.unibo.jumpig.common.impl
- Classi MenuControllerImpl e LeaderboardControllerImpl nel package it.unibo.jumpig.controller.impl
- Interfacce Camera, Game, Leaderboard, Score, Velocity e World nel package it.unibo.jumpig.model.api
- Classe PlayerImpl nel package it.unibo.jumpig.model.impl.gameentity

- Classi LeaderboardImpl, ScoreImpl, VelocityImpl e alcune parti di GameImpl nel package it.unibo.jumpig.model.impl
- Classi LeaderboardViewSceneImpl, MenuViewSceneImpl PlayerKeyListener e ScorePanel nel package it.unibo.view.impl

## 3.3 Note di sviluppo

### Francesco Zattoni

- ***Polimorfismo Vincolato***
  Utilizzato in varie interfacce e classi come nei seguenti esempi: https://github.com/francesco-zatto/OOP22-jumpig/blob/55c12d060534bf1f74d9582937e519ee27e19a7e/src/main/java/it/unibo/jumpig/model/api/collision/CollisionHandler.java#L13; https://github.com/francesco-zatto/OOP22-jumpig/blob/55c12d060534bf1f74d9582937e519ee27e19a7e/src/main/java/it/unibo/jumpig/model/api/collision/CollisionChecker.java#L14; https://github.com/francesco-zatto/OOP22-jumpig/blob/55c12d060534bf1f74d9582937e519ee27e19a7e/src/main/java/it/unibo/jumpig/model/impl/collision/PlatformCollisionChecker.java#L12.
- ***Java Bounded Wildcard***
  Permalink: https://github.com/francesco-zatto/OOP22-jumpig/blob/55c12d060534bf1f74d9582937e519ee27e19a7e/src/main/java/it/unibo/jumpig/model/impl/WorldImpl.java#L190

- ***Lambda***
  Permalink: https://github.com/francesco-zatto/OOP22-jumpig/blob/55c12d060534bf1f74d9582937e519ee27e19a7e/src/test/java/collision/EnemyCollisionHandlerTest.java#L44
- ***Method Reference***
  Permalink:

https://github.com/francesco-zatto/OOP22-jumpig/blob/55c12d060534bf1f74d9582937e519ee27e19a7e/src/main/java/it/unibo/jumpig/model/impl/WorldImpl.java#L191
https://github.com/francesco-zatto/OOP22-jumpig/blob/55c12d060534bf1f74d9582937e519ee27e19a7e/src/test/java/collision/EnemyCollisionHandlerTest.java#L46

- ***Constructor Reference***
  Permalink:
  https://github.com/francesco-zatto/OOP22-jumpig/blob/55c12d060534bf1f74d9582937e519ee27e19a7e/src/test/java/collision/EnemyCollisionHandlerTest.java#L76
- ***Stream***
  Permalink:
  https://github.com/francesco-zatto/OOP22-jumpig/blob/55c12d060534bf1f74d9582937e519ee27e19a7e/src/main/java/it/unibo/jumpig/model/impl/WorldImpl.java#L191;
  https://github.com/francesco-zatto/OOP22-jumpig/blob/55c12d060534bf1f74d9582937e519ee27e19a7e/src/test/java/collision/PlatformCollisionHandlerTest.java#L69

## Francesco Filippini

- ***Stream***
  Utilizzati in vari punti come ad esempio:
  https://github.com/francesco-zatto/OOP22-jumpig/blob/d80b5304b7a236db43af969eabd9d794648d62eb/src/main/java/it/unibo/jumpig/model/impl/GenerateCoinsStrategy.java#L54
- ***Lambda expressions***
  Utilizzati in vari punti come ad esempio:
  https://github.com/francesco-zatto/OOP22-jumpig/blob/d80b5304b7a236db43af969eabd9d794648d62eb/src/main/java/it/unibo/jumpig/model/impl/WorldImpl.java#L216
- ***Optional***

Permalink:
https://github.com/francesco-zatto/OOP22-jumpig/blob/c6a35c6a53c6c967528ceb37f3d8be4ffd7eb37a/src/main/java/it/unibo/jumpig/model/impl/CameraImpl.java#L34

- ***Metodo con tipo di ritorno generico***
  Permalink:
  https://github.com/francesco-zatto/OOP22-jumpig/blob/d80b5304b7a236db43af969eabd9d794648d62eb/src/main/java/it/unibo/jumpig/model/impl/GeneratorEntitiesImpl.java#L47
- ***Java Bounded WildCard***
  Permalink:
  https://github.com/francesco-zatto/OOP22-jumpig/blob/d80b5304b7a236db43af969eabd9d794648d62eb/src/main/java/it/unibo/jumpig/model/impl/WorldImpl.java#L38
- ***Thread***
  Permalink:
  https://github.com/francesco-zatto/OOP22-jumpig/blob/d80b5304b7a236db43af969eabd9d794648d62eb/src/main/java/it/unibo/jumpig/view/impl/MenuViewSceneImpl.java#L128
- ***Method Reference***
  Permalink:
  https://github.com/francesco-zatto/OOP22-jumpig/blob/d80b5304b7a236db43af969eabd9d794648d62eb/src/main/java/it/unibo/jumpig/view/impl/MenuViewSceneImpl.java#L127
- ***Metodo di default in un' interfaccia***
  Permalink:
  https://github.com/francesco-zatto/OOP22-jumpig/blob/c6a35c6a53c6c967528ceb37f3d8be4ffd7eb37a/src/main/java/it/unibo/jumpig/model/api/GeneratorEntitiesStrategy.java#L44

Ho preso ispirazione dalla repository https://github.com/pslab-unibo/oop-game-prog-patterns-2022.git su Github di https://github.com/aricci303. Di seguito il link che ho utilizzato per lo scheletro della classe GameControllerImpl:

https://github.com/pslab-unibo/oop-game-prog-patterns-2022/blob/eaeb2f311da8fdd2129eae3acb875cf90fb491c6/step-01-game-loop/src/rollball/core/GameEngine.java#L11.

## Alessandro Verna

- ***Streams***
  Permalink:
  https://github.com/francesco-zatto/OOP22-jumpig/blob/d80b5304b7a236db43af969eabd9d794648d62eb/src/main/java/it/unibo/jumpig/model/impl/LeaderboardImpl.java#L38
- ***Optional***
  Permalink:
  https://github.com/francesco-zatto/OOP22-jumpig/blame/d80b5304b7a236db43af969eabd9d794648d62eb/src/main/java/it/unibo/jumpig/model/impl/gameentity/PlayerImpl.java#L24
- ***Lambda expressions***
  Permalink:
  https://github.com/francesco-zatto/OOP22-jumpig/blob/d80b5304b7a236db43af969eabd9d794648d62eb/src/main/java/it/unibo/jumpig/view/impl/MenuViewSceneImpl.java#L125

- ***Method reference***
  Permalink:
  https://github.com/francesco-zatto/OOP22-jumpig/blob/d80b5304b7a236db43af969eabd9d794648d62eb/src/main/java/it/unibo/jumpig/model/impl/LeaderboardImpl.java#L41

- ***Utilizzo di Predicate***
  Permalink:
  https://github.com/francesco-zatto/OOP22-jumpig/blob/d80b5304b7a236db43af969eabd9d794648d62eb/src/main/java/it/unibo/jumpig/model/impl/GameImpl.java#L92

# Capitolo 4

# Commenti finali

## 4.1 Autovalutazione e lavori futuri

### Francesco Zattoni

Devo ammettere che inizialmente non è stato per niente facile partire, trovare una buona idea e un'analisi che ci convincessero.
Però mi è piaciuto molto questo progetto perchè mi ha dato la prima vera occasione per lavorare effettivamente in gruppo. Mi ha permesso di imparare le basi di Git e GitHub, che prima di questo corso conoscevo solo di fama. Ci sono stati diversi momenti difficili, in cui pensavo di essere in un vicolo cieco, ma proprio da questi ho ritrovato la voglia di continuare e di migliorarmi.
Sono soddisfatto di ciò che ho fatto e spero di aver fatto un buon lavoro. Non penso che Jumpig verrà supportato da parte mia dopo la consegna, ma lo terrò sul desktop come ricordo di questa esperienza.

### Francesco Filippini

Per me questa è stata la prima volta che ho lavorato su un progetto di gruppo di dimensioni così importanti. Devo ammettere che è stato in un certo senso affascinante scoprire come collaborare in gruppo utilizzando il DVCS Git e vedere come pian piano siamo riusciti a costruire un gioco partendo da zero.
Nonostante i miei sforzi, invani a causa dello svolgimento delle lezioni dei corsi del secondo anno secondo semestre, di provare a lavorare su questo progetto un po' tutti i giorni, riconosco che ci sono degli accorgimenti a cui non ho prestato massima attenzione. Questo è stato però dovuto alla

mancanza di sufficiente tempo a disposizione e mi riferisco alla mia parte di generazione delle entità del mondo di gioco. Infatti avessi avuto modo sicuramente avrei aggiunto qualche controllo in più come ad esempio evitare che vengano generate entità leggermente sovrapposte l'una sulle altre. Per questo motivo segnalo per un eventuale lavoro futuro di controllare e sistemare la parte di generazione del mondo di gioco. In particolare il primo controllo da fare sarà quello di verificare che una nuova piattaforma sia generata non troppo distante dalla precedente in modo che il personaggio possa sempre avere una piattaforma su cui saltare.
Detto questo credo (e spero) comunque di aver fatto un buon lavoro e, se così non fosse, per lo meno è sicuramente un lavoro di cui sono fiero.
Per concludere il mio punto di vista su Jumpig voglio dire che non ci sarà (o almeno sicuramente non nell'immediatezza) un eventuale sviluppo futuro.

## Alessandro Verna

Dato che è la mia prima volta in cui lavoro in gruppo ad un progetto di queste dimensioni, devo ammettere inizialmente è stato impegnativo.
D'altro canto, sono riuscito in poco tempo ad orientarmi bene attraverso l'utilizzo del DVCS Git e GitHub e a scoprire il loro vero potenziale.
Durante questo percorso , ci sono stati momenti in cui mi sono bloccato in alcune parti, ma non mi sono arreso al primo problema incontrato e sono riuscito a procedere, riuscendo a comprendere, migliorando le mie conoscenze e aumentando la mia voglia di scoprire di più.
In conclusione sono soddisfatto del risultato finale e spero di aver svolto un buon lavoro.
Per quanto riguarda Jumpig, non penso che ci sarà un prossimo sviluppo, almeno, nell'immediatezza, ma penso che sarà soddisfacente in futuro tornare a guardare il progetto e ricordarsi di questa esperienza.

## 4.2 Difficoltà incontrate e commenti per i docenti

Riteniamo il progetto di OOP essenziale per avere un primo e vero contatto con l'analisi, il design di un dominio e con i paradigmi della programmazione a oggetti così da completare quanto visto durante le lezioni, trovate molto chiare ed interessanti.
Tuttavia andrebbe rivisto perché questo progetto ruba davvero molto tempo agli altri corsi del secondo semestre del secondo anno.
Probabilmente la deadline di giugno non è la migliore, ma comunque rimane un bel problema per cui riteniamo che bisognerebbe diminuire il monte ore del progetto. 80 ore aggiuntive solo per il progetto ci sembrano eccessive se confrontiamo il corso di programmazione ad oggetti ad altri corsi anch'essi da 12 crediti.
Un'altra critica che ci sentiamo di fare è sulla parte di analisi e design, sulla quale secondo noi non eravamo abbastanza preparati.
Ultima difficoltà incontrata che riteniamo particolarmente rilevante è stata la perdita di tempo dovuta alle catastrofiche conseguenze dell'alluvione che ha colpito la provincia di Forlì-Cesena.

# Appendice A

## Guida utente

Appena si avvia il gioco, si apre il menu in cui inserire un nome utente per la partita, avviare la partita stessa, aprire la classifica oppure uscire dal gioco. Avviando la partita senza inserire un nome, il punteggio della partita verrà salvato nella classifica come “user + numero random”. I comandi del gioco sono molto semplici, è sufficiente usare le freccette della tastiera per muoversi verso destra(>) o verso sinistra(<).
All’interno del gioco l’utente dovrà schivare gli chef Jumpier, i nemici, e dovrà stare attento alle piattaforme: le piattaforme verdi sono sicure, le piattaforme rosse contraddistinte dal punto esclamativo dopo il primo salto spariscono, mentre le piattaforme marroni rotte non sono affidabili, appena ci si prova a saltare sopra si rompono, quindi attenzione!