Frogger

Barzanti Alan
Mondardini Marco
Federico Mariani
Lorenzo Ferri

Giugno 2025

Indice

1 ANALISI

1.1 Descrizione e requisiti

Il software è una rivisitazione del gioco "Frogger", in cui l'obiettivo del giocatore è quello di superare più livelli possibile. Si può morire nel caso si entri in contatto con ostacoli come macchine, camion e acquile, oppure se saltando da un tronco all'altro si finisce in acqua. È anche possibile possibile prendere power up e collezionare monete, spendibili in un negozio.

Requisiti funzionali:

- I livelli dovranno essere infiniti e randomici.
- Macchine, camion e tronchi dovranno essere in movimento e occorrerà gestire le varie collisioni.
- Dovranno essere presenti nel livello dei power up: "vita extra", "freeze degli ostacoli" e "moltiplicatore di punteggio".
- Dovranno essere presenti nel livello delle monete spendibili in un negozio.
- Dovranno essere create diverse interfacce utente (schermata principale, menù di pausa e menù di game over).

Requisiti non funzionali:

- Il software deve essere compatibile con ogni sistema operativo.

1.2 Modello del dominio

Ogni livello è composto da delle corsie, che possono essere: *terra*, *strada* e *acqua*. Nelle corsie di tipo strada ci sono *macchine* e *camion* che, se entrano in contatto con il giocatore, provocano la perdita di una vita.
Nelle corsie di acqua ci sono dei *tronchi* galleggianti, in questo caso è il contatto con l'acqua stessa a provocare la "morte" del giocatore che, quindi, dovrà saltare da un tronco all'altro.
Le corsie di terra sono considerate "neutrali" ma neanche in queste si è completamente in salvo. Infatti ogni livello ha un numero indefinito di *aquile* che volano attraverso il livello in verticale, passando sopra ad ogni tipo di corsia e, se entrano in contatto con il giocatore, ne provocano la morte. Le

aquile non partiranno subito all'avvio della partita ma in momenti separati e diversi per ogni aquila.
All'interno del livello inoltre sono presenti dei *power up* con probabilità differenti di spawning, ovvero oggetti raccoglibili che danno un effetto particolare, tra cui:
- *Freeze*: ferma il movimento degli ostacoli per 3 secondi
- *Double Score*: raddoppia i punti guadagnati per 5 secondi
- *Extra Life*: aggiunge una vita extra al giocatore
Tra gli oggetti raccoglibili all'interno del livello ci sono le *monete*, monete da 1 e gruppi di monete da 5, una volta raccolte si andranno a sommare al totale di quelle raccolte in tutte le partite fatte dall'avvio del programma. Esse sono utilizzabili nel *negozio* a cui si può accedere tramite il menù principale, in cui si possono acquistare skin differenti per il proprio personaggio.

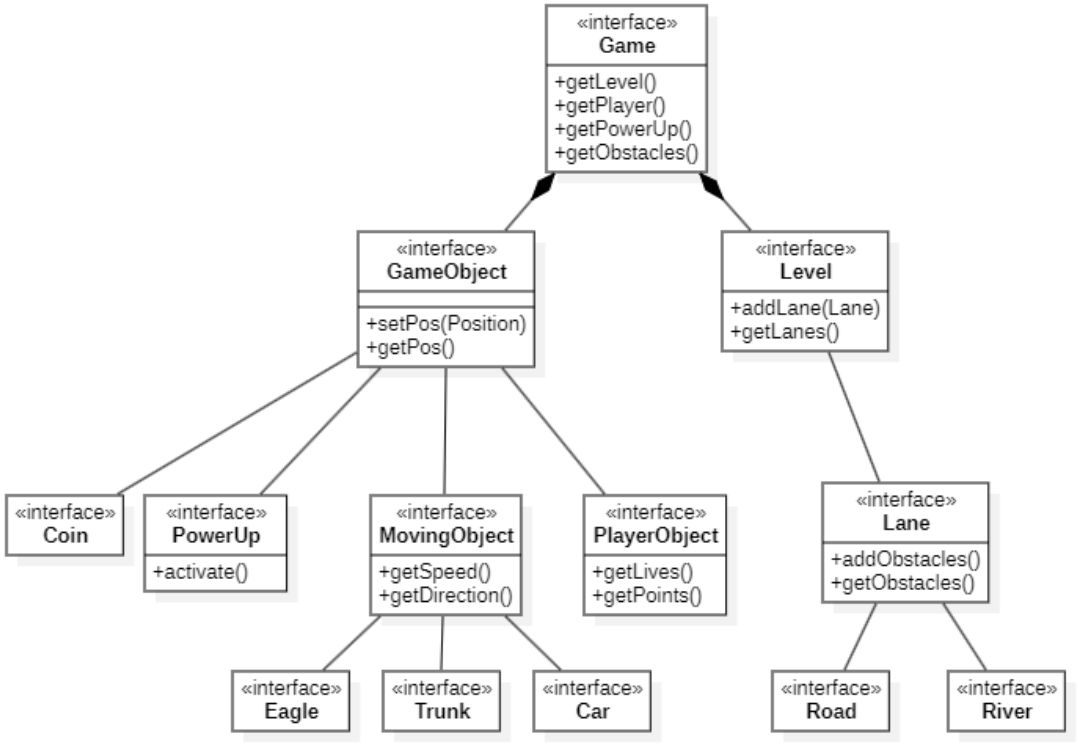

Figura 1: UML del dominio applicativo.

2 DESIGN

2.1 Architettura

Abbiamo seguito la struttura del pattern architetturale *MVC*, dividendo la parte di modellazione del dominio applicativo (*model*) da quella di rappresentazione grafica (*view*), con un sistema che funga da intermediario tra le due parti (*controller*).
Il sistema è progettato per cambiare il controller attivo in base allo "stato" del gioco. Per farlo abbiamo creato un *MainController*, la cui funzione è quella di lasciare la gestione del frame al controller appropriato (*GameController*, *ShopController* o *MenuController*), sarà poi compito di quest'ultimo visualizzare il contenuto del gioco.
Gli **Entry Point** per il model sono le interfacce:

- **Game**, contenuta in *GameController*.
- **Shop**, contenuta in *ShopController*.
- **Menu**, contenuta in *MenuController*.

Per quanto riguarda le interazioni tra *Controller* e *View* abbiamo pensato di riservare ad ogni controller uno specifico "panel" (quindi *GamePanel*, *ShopPanel*, *MenuPanel*).
Così facendo il controller corrente, ricevuto il frame principale da *MainController*, è in grado di istanziare la giusta classe di *View* del quale potrà richiamare il metodo per disegnare la schermata.
Pensiamo che con questo tipo di architettura effettuare una sostituzione in blocco della *View*, passando ad esempio da Swing a JavaFX, comporterebbe uno sforzo relativamente basso visto che la dipendenza con il *Controller* è minima.

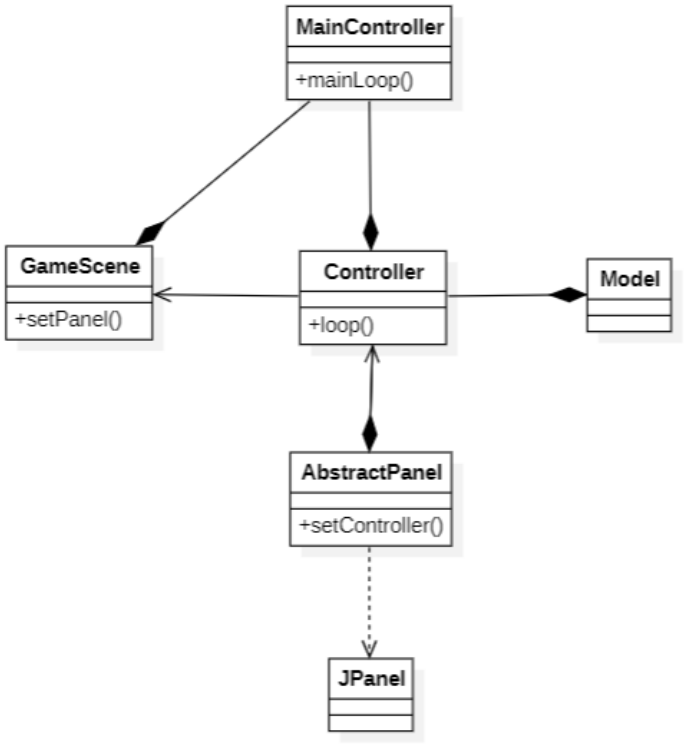

Figura 2: UML base dell'architettura del progetto

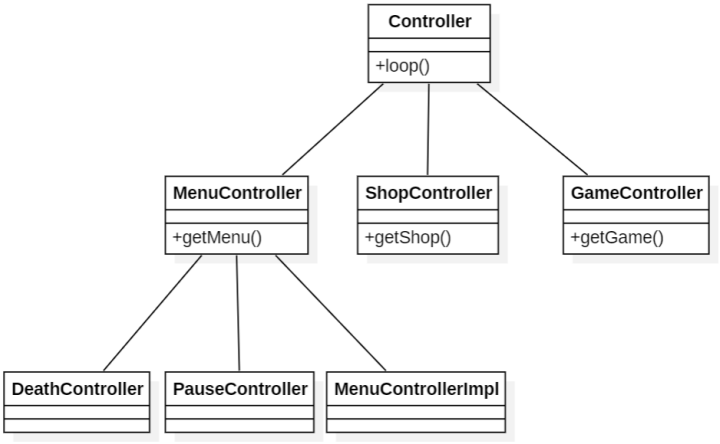

Figura 3: UML della modellazione dei Controller

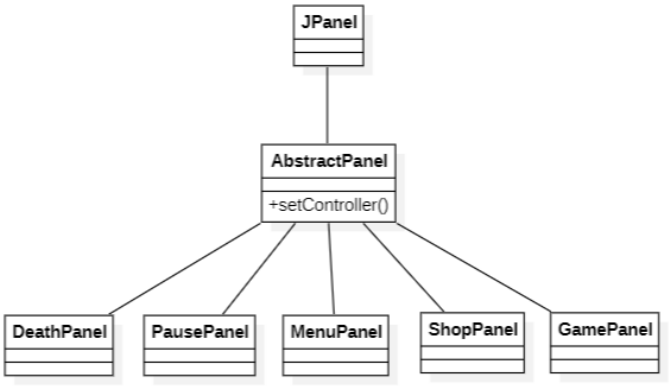

Figura 4: UML della modellazione della view

2.2 Design dettagliato

2.2.1 Barzanti Alan

Problema Modellare le diverse tipologie di corsie e la struttura interna del livello.

Soluzione Il livello si compone di una lista di *Lane*, interfaccia che definisce un contratto comune a cui tutte le tipologie di corsia aderiscono. Per massimizzare il riuso del codice ho deciso di utilizzare una classe astratta *AbstractLaneImpl* che implementa Lane e tutti i suoi metodi ad eccezione di *addMovingObject(MovingObject)* che viene implementato dalle sottoclassi *Ground*, *River* e *Road* con il loro comportamento specifico. Utilizzo una classe astratta invece di una concreta per impedire che venga istanziata (avrei potuto farlo anche in una classe concreta con un costruttore privato ma così evito di scrivere codice inutile). Si nota che allo stato attuale del progetto, non essendoci particolari differenze di comportamento tra le varie tipologie di Lane, anche un'unica classe generica "LaneImpl" per ogni tipo sarebbe stata una valida opzione. Ho scelto comunque di differenziare le classi nonostante potesse esserci qualche ridondanza o banalità a livello di codice per tre motivi principali: modellare meglio a livello concettuale il dominio, migliorando la leggibilità, permettere l'estensibilità futura del progetto, ad esempio aggiungendo comportamenti specifici per un tipo, ed evitare di basare la logica delle diverse tipologie su una concatenazione di if-else, che appunto sarebbe ancora peggio nel caso di un'estensione con nuove Lane.

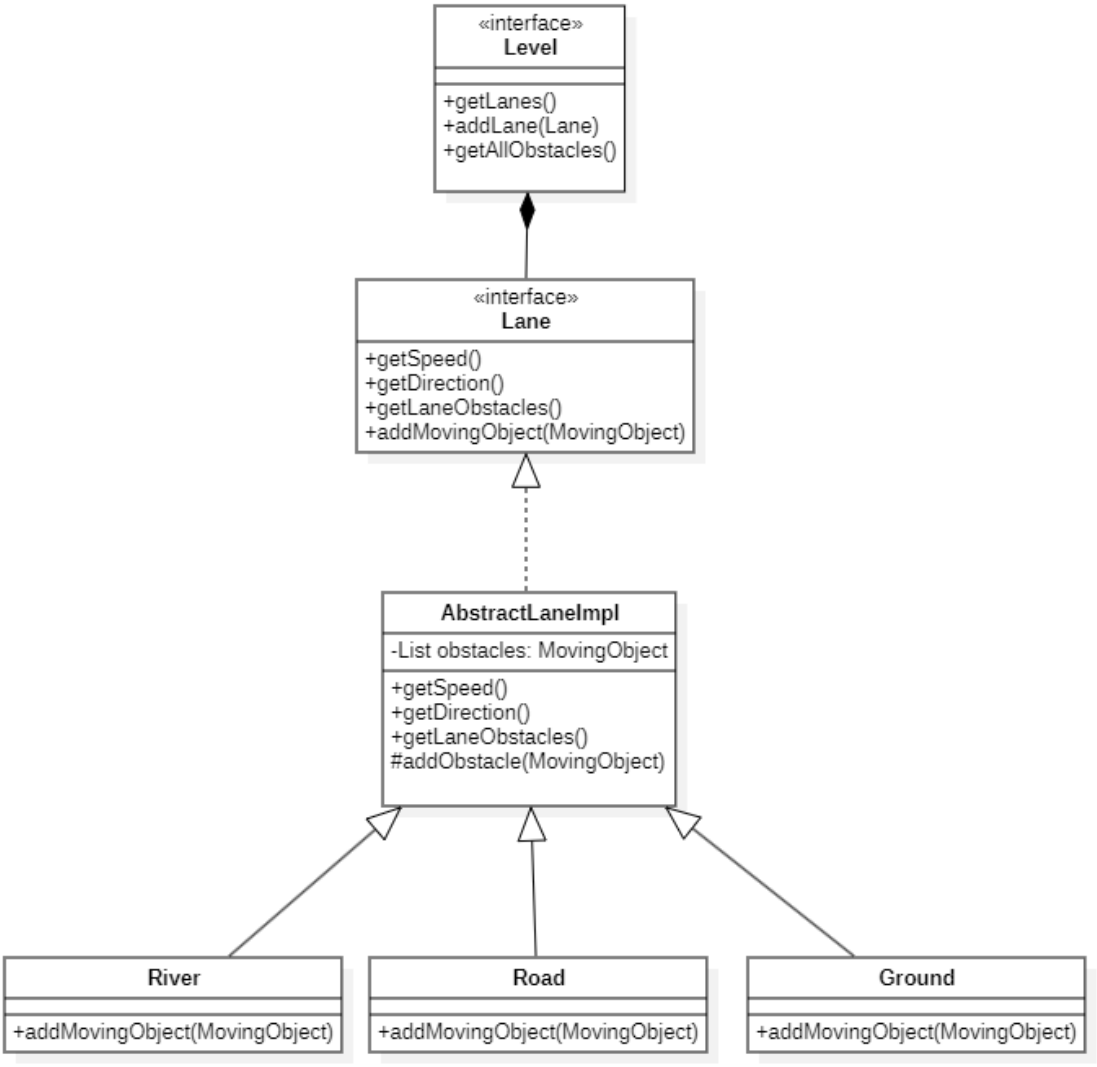

Figura 5: UML della modellizzazione per Lane e Level.

Problema Creare un livello con diverse tipologie di creazione (randomica, a difficoltà progressiva, ecc.)

Soluzione Ho pensato che calzasse bene l'utilizzo del pattern *Factory Method*, ho quindi creato un'interfaccia *LevelFactory* che contiene il factory

method *createLevel()* e lascia alle sottoclassi che la implementano la scelta del tipo di livello da generare (cioè i *concrete creator*). Nel caso specifico abbiamo solo l'implementazione randomica, *RandomLevelFactory*, ma la struttura è pensata per essere facilmente estendibile senza dover modificare codice esistente (principio OCP). Per altri tipi di livello (come quello a difficoltà progressiva) sarà infatti sufficiente implementare LevelFactory.

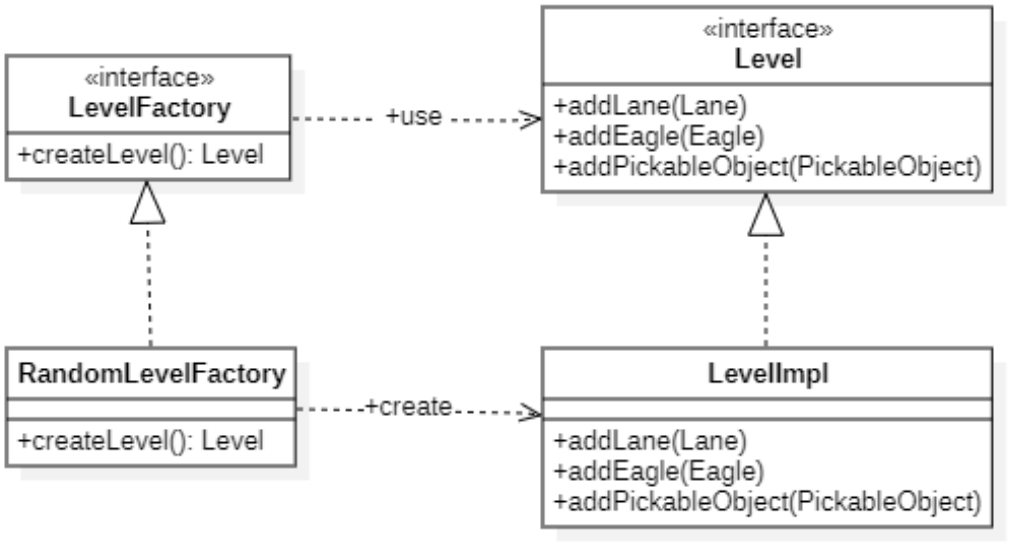

Figura 6: UML del pattern Factory Method.

Problema Implementare una logica di creazione degli ostacoli randomica.

Soluzione In prima battuta ho implementato lo "spawn" di ciascun entità direttamente in *RandomLevelFactory* tramite metodi privati, ma questo rompeva il principio SRP e generava una classe inutilmente lunga e ridondante. Dopo vari refactoring sono arrivato alla soluzione attuale, utilizzando il pattern *Template Method*. Ho creato un'interfaccia generica *EntitySpawner* che modella il concetto di "spawner", ovvero un oggetto con l'unico obiettivo di restituire una lista di entità di un certo tipo. Aderendo quindi al pattern ho

una classe astratta generica che implementa il metodo *spawn()*, ma dipende dall'implementazione di vari metodi astratti (quindi spawn() è il *template method*). Ogni sottoclasse è adibita alla creazione di una specifica entità e implementa i metodi astratti in base ai suoi bisogni specifici. Ho trovato buona questa soluzione sia per il Single Responsibility Principle, infatti ora ogni classe coinvolta è strettamente responsabile di una cosa, sia per la notevole riduzione di ridondanze in *RandomLevelFactory* (anche se ancora non era perfetta, come si noterà nella prossima sezione).

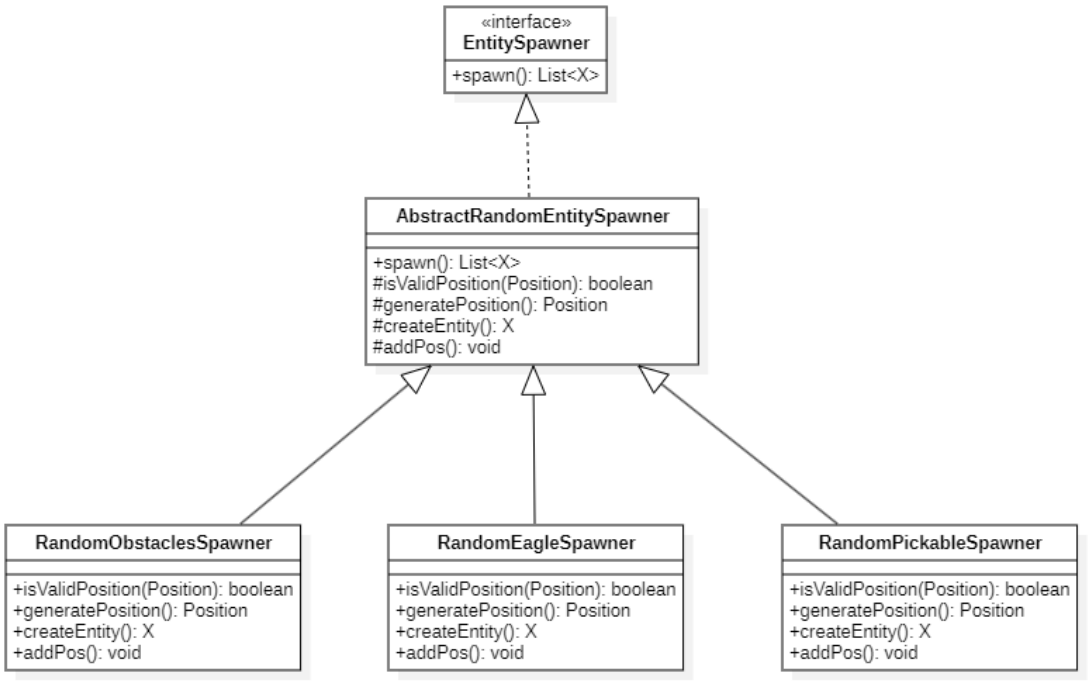

Figura 7: UML del pattern Template Method per modellare gli spawner.

Problema Creare un modo centralizzato per accedere ad ogni spawner.

Soluzione In *RandomLevelFactory* procedendo a sostituire ai metodi privati gli spawner appena creati, ho notato una superflua e pesante ridondanza

di codice dovendone istanziare molteplici. Per questo ho pensato fosse necessario un modo per centralizzare le varie creazioni, a questo scopo ho utilizzato il pattern *Abstract Factory*. L'interfaccia *SpawnerFactory* è l'abstract factory, rappresenta una "fabbrica" per tutte le classi della "famiglia" degli spawner, lasciando alle sottoclassi che la implementano la libertà di decidere quale specializzazione (randomica, ecc.) di spawner istanziare. Nel caso specifico utilizzo solo l'approccio randomico, ho quindi *RandomSpawnerFactory* che implementa la factory e istanzia gli spawner randomici visti nella sezione precedente. In questo modo riduco considerevolmente le ripetizioni, ad esempio in RandomLevelFactory mi è sufficiente istanziare la factory per poter richiamare ogni tipo di spawner tramite metodi. Inoltre questo "passaggio intermedio" tra gli spawner e il client ha anche i vantaggi di: semplificare al client la creazione di spawner più specifici che vengono gestiti internamente, ad esempio vengono forniti i metodi per istanziare uno spawner di Car e uno di Trunk nonostante in realtà ne esista solo uno generico, e di non mostrare al client la parte di random injection (utile in fase di testing), che viene gestista internamente passando un "vero" oggetto random. Ho cercato di creare queste ultime tre sezioni con l'obiettivo di rendere il più semplice possibile una concreta possibiltà di estensione del progetto, cioè l'aggiunta di nuove tipologie di livello visto che nelle funzionalità opzionali ne compariva una (a difficoltà progressiva). Con questo tipo di design sarebbe sufficiente creare una classe che implementi *LevelFactory* nel quale utilizzare degli spawner appositi implementando *EntitySpawner* e centralizzandone l'accesso con una factory che implementi *SpawnerFactory*.

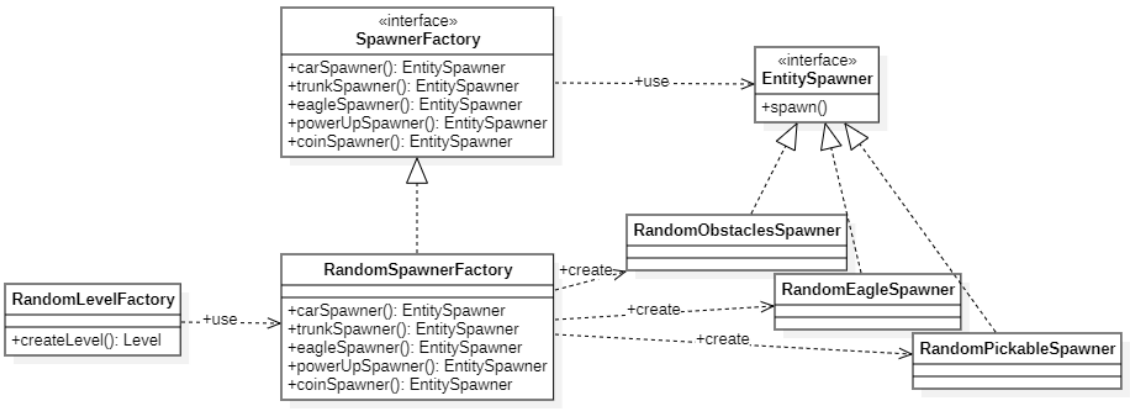

Figura 8: UML del pattern Abstract Factory.

2.2.2 Mariani Federico

Problema Il sistema necessita di diversi tipi di oggetti che si muovono da soli, che chiameremo *MovingObject*, essi hanno come principio fondamentale il muoversi ma possono sviluppare altre caratteristiche (es: i tronchi devono "trasportare" altri oggetti, le aquile devono partire in momenti prestabiliti), ciò ci costringe a creare classi differenti e perciò porta ad una pesante duplicazione del codice per le funzionalità base.

Soluzione Siccome l'unica differenza nel comportamento da *MovingObject* era nel modo di muoversi, per ovviare a tale problema ho utilizzato il pattern *Template Method*, con una piccola variazione. il *Template method* all'interno della classe MovingObjectImpl è *move()* che al suo interno usa *step()*. La variazione sta nel fatto che *step()* non è astratto ma è solo protetto e contiene l'implementazione base del passo, ovvero lo spostamento della posizione. In questo modo è utilizzabile in ogni sottoclasse senza doverla riscrivere, cosa che sarebbe invece successa usando il pattern Template Method base, lo rende molto comodo e riduce al minimo la duplicazione del codice.

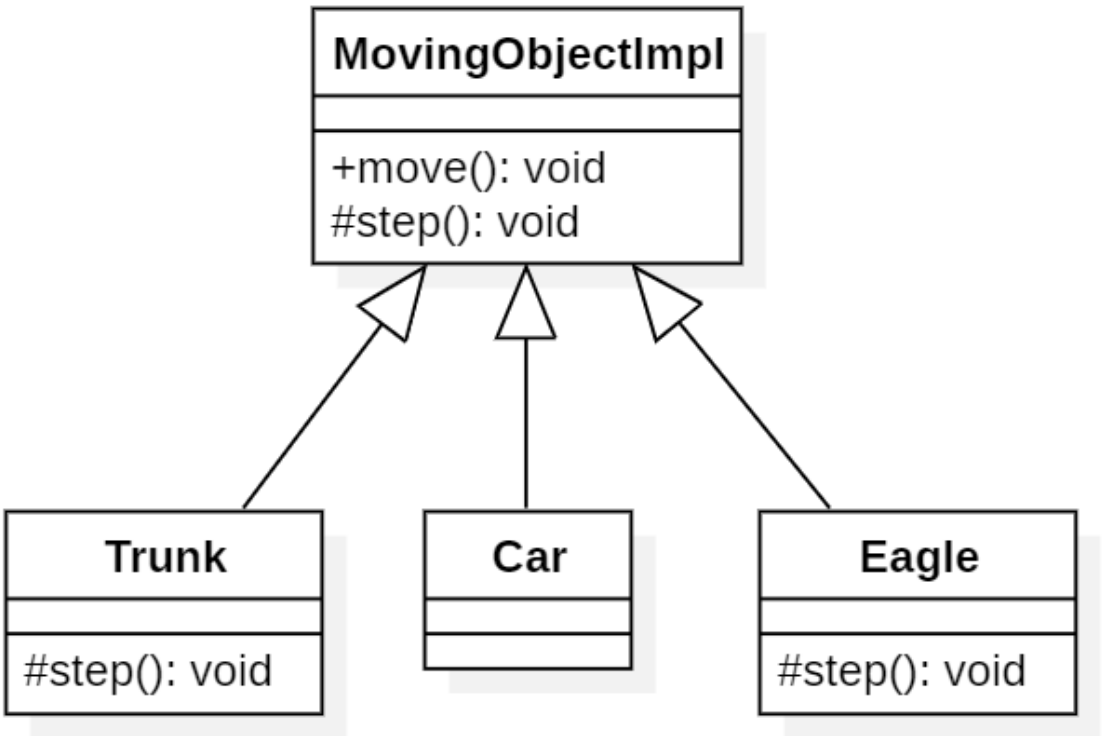

Figura 9: UML del Template Method dei MovingObject.

Problema Il programma deve controllare i diversi stati del gioco ognuno con un controller ed ognuno di essi possiede un loop con sempre la stessa struttura. Questo crea duplicazione del codice.

Soluzione Siccome le cose che differenziavano tra un loop e l'altro all'interno dei diversi controller erano poche righe di codice ho deciso di utilizzare anche in questo caso il pattern *Template Method*. Ho creato nella classe padre *AbstractController* il template method *loop()* che al suo interno usa 2 metodi protetti e astratti: *core()* e *loopCondition()*, in modo da coprire le possibili parti che necessitano codice nel loop ed evitare la duplicazione di codice.

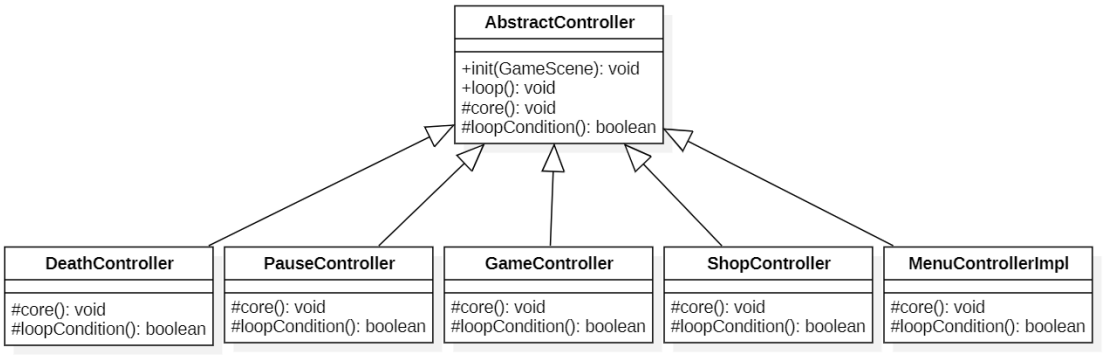

Figura 10: UML del Template Method dei Controller.

Problema Il sistema deve poter creare ogni tipo di *MovingObject* e per sviluppi futuri si può pensare a nuovi oggetti di questo tipo, perciò si vuole puntare alla centralizzazione della creazione degli oggetti, al riutilizzo migliore del codice, e all'estendibilità di esso.

Soluzione Per migliorare la crezione dei *MovingObject* ho creato una *Factory*, utilizzando il pattern *Simple Factory*, ovvero creando una classe pubblica che istanziandola dà la possibilità di creare un qualsiasi *MovingObject*, i vantaggi di questa factory è che centralizza totalmente la creazione degli oggetti, e in questo modo aiuta anche il SOLID principle "Open/Close" dando la possibilità di aggiungere nuove classi dovendo estendere solamente la factory e nient'altro. In questo caso si poteva utilizzare in modo simile il pattern *Static Factory* ma siccome personalmente preferisco stare il più attento possibile al numero di classi statiche che creo ho pensato fosse ugualmente conveniente usare una classe pubblica.

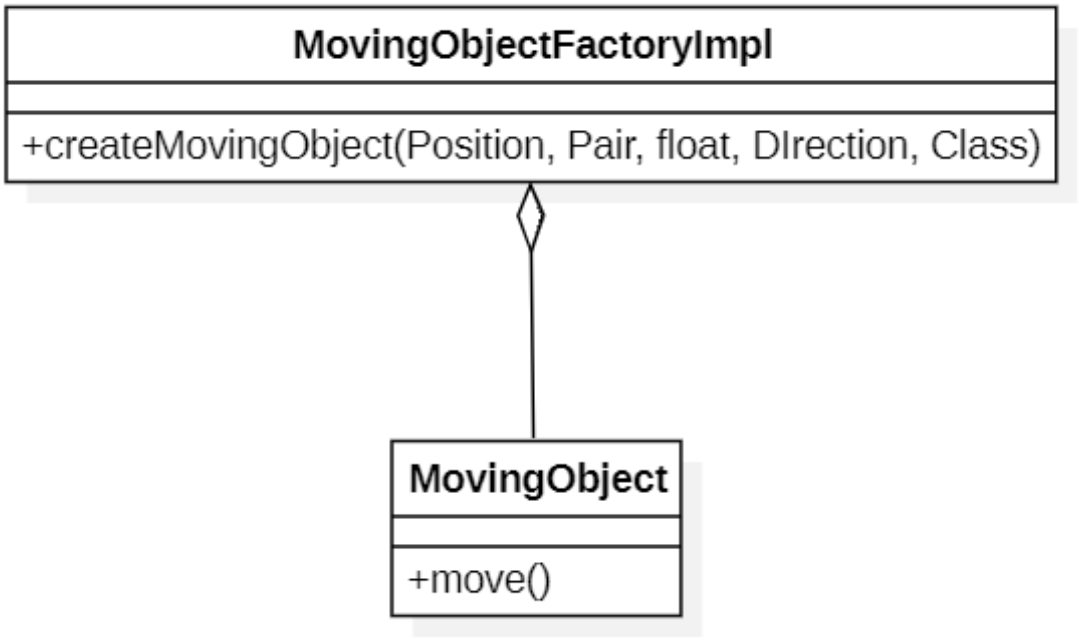

Figura 11: UML della Factory dei MovingObject.

Problema All'interno del programma sono molteplici i menù che vanno mostrai, ad esempio quello principale all'avvio del gioco, il menu di pausa e il menù di "morte" una volta terminata la partita. Ogni menù possiede dei bottoni che hanno come solo compito quello di cambiare stato al gioco ovvero indicare al *MainController* che è successo qualcosa e che è il momento di dare ad un altro *Controller* il "posseso" del programma.

Soluzione Per risolvere questo problema ho prima di tutto generalizzato la classe *Menu*, dandogli la possibilità di avere un qualsiasi numero di bottoni che portassero ad un qualsiasi stato del gioco. E per utilizzare efficientemente questa "generalizzazione" ho usato il pattern *Simple Factory* in modo da poter creare direttamente da quest'ultima i principali tipi di Menu che differiscono per tipi di bottone e ordine. Centralizzando ciò si possono aggiungere altri tipi di menu, o riutilizzare altrove quelli già creati in altri.

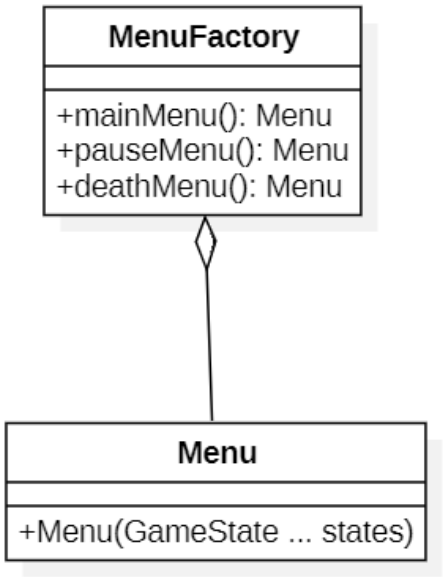

Figura 12: UML della Factory dei Menu.

2.2.3 Mondardini Marco

Problema La crezione dello shop è stato l'aspetto più complicato per me. Volevo trovare un metodo per creare gli oggetti dello shop in modo "automatico", per fare in modo anche che nel momento in cui il giocatore compra un oggetto quell'oggetto venga segnato come "Acquistato", anche nella View.

Soluzione Gli oggetti, *PurchasableObject*, sono salvati in un file, se un oggetto viene acquistato o equipaggiato, un nuovo file verrà scritto con le informazioni aggiornate. Avendo pensato che si possano vendere più tipologie di oggetti da vendere, l'interfaccia *PurchasableObject* viene implementata da una classe astratta *AbstractPurchasableObject*, che viene poi implementata dalle sottoclassi, per ora solo la sottoclasse *Skin*. Il processo di lettura e aggiornamento dei dati avviene attraverso la classe *Shop*, che esegue anche l'inizializzazione con i dati iniziali. Per per il discorso che in futuro si potrebbero aggiungere nuove tipologie di oggetti vendibili o già creato una classe Factory per le future implementazioni, *PurchasableObjectFactory*.

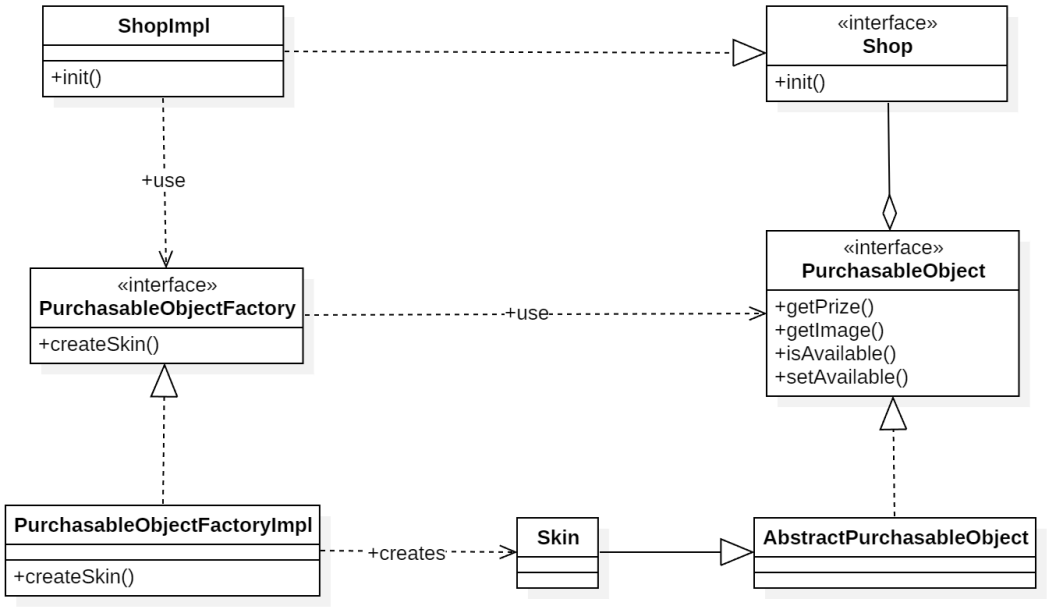

Figura 13: UML dello shop con pattern Simple Factory.

Problema Lettura ed esecuzione dell'input dei comandi.

Soluzione Ho creato una classe, *KeyInput*, che implementa l'interfaccia *KeyListener* per la lettura dell'input dei tasti. Ho deciso di usare il pattern *Command* per la logica dell'input creando per ognuna delle quattro direzioni una classe che implementi l'interfaccia *Command*. Questi "Comandi" possono essere notificati o processati da un *InputController*. La classe *GameController* visto che deve gestire il core del gioco deve poter dire quando processare un input, quindi ha in dotazione un InputController.

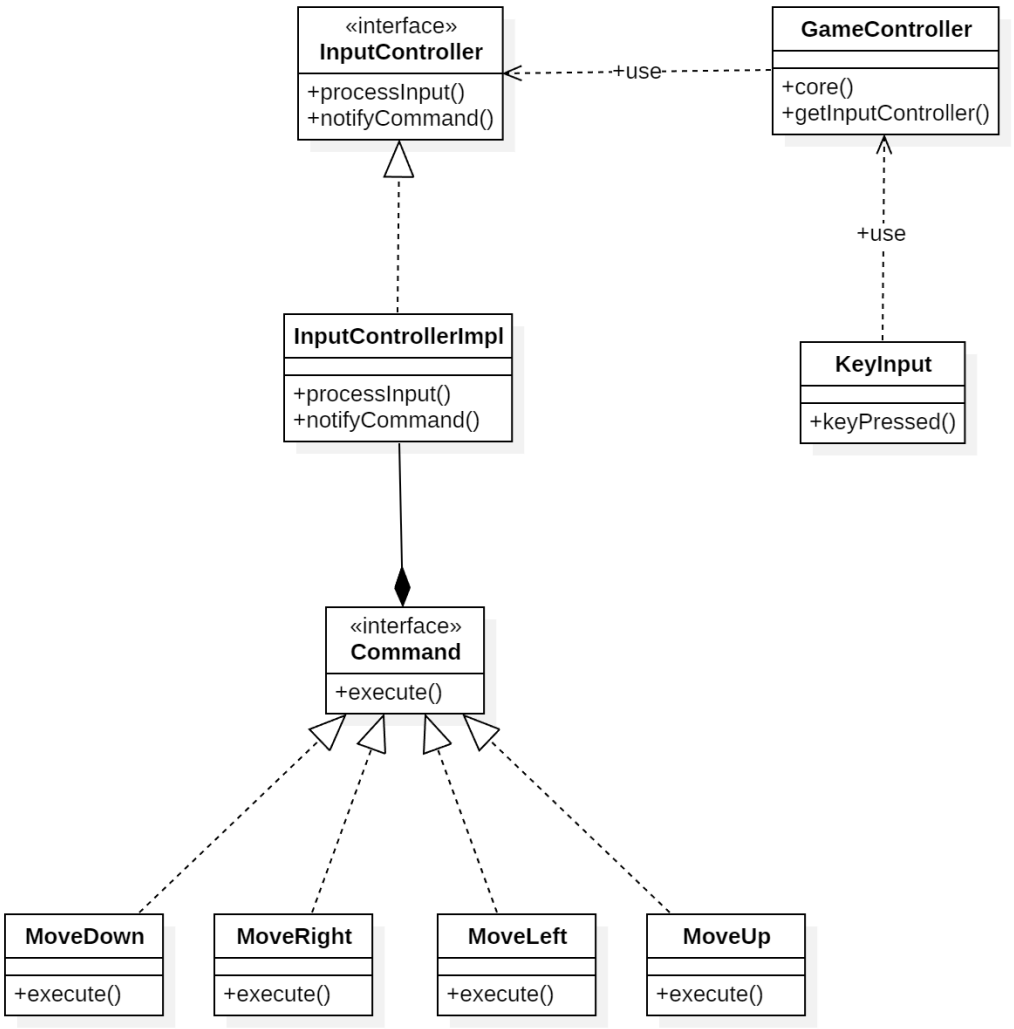

Figura 14: UML dell'input con pattern Command.

Problema Gestione del Player e creazione di un sistema per controllare la "partita", con lo scopo di gestire le collisioni con ostacoli e power up.

Soluzione Ho creato una classe *PlayerObject* che implementa l'interfaccia *GameObject*. Ho poi creato la classe *Game*, che ha a disposizione il Player, gli ostacoli, i power up e l'intero livello, in modo da poter gestire ogni situazione e componente della partita.

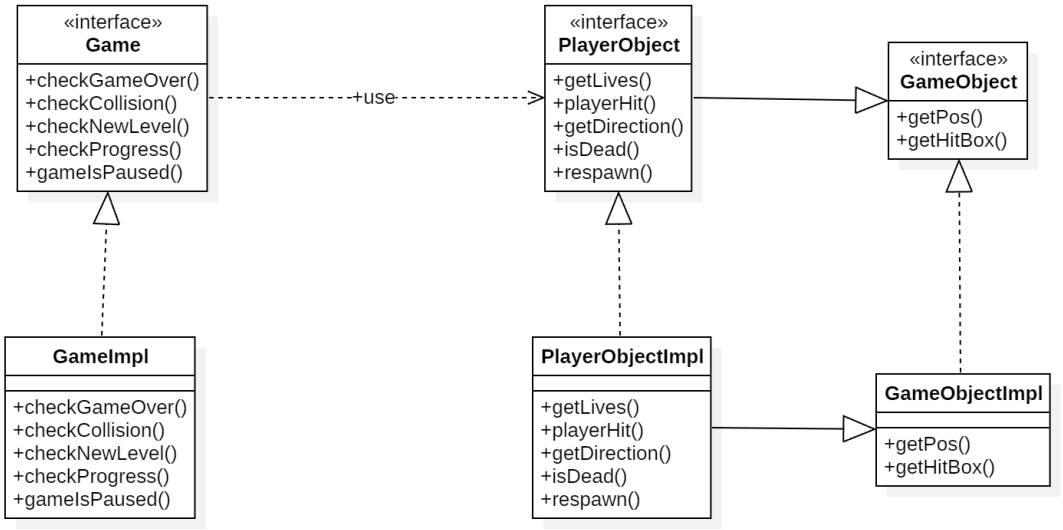

Figura 15: UML del Player con gestione partita.

2.2.4 Ferri Lorenzo

Problema Gestire diversi tipi di power-up raccoglibili, ciascuno con un effetto specifico, evitando duplicazioni di codice e garantendo estendibilità per future aggiunte.

Soluzione Ho modellato i power-up tramite l'interfaccia *PowerUp*, e una classe astratta *PowerUpImpl*, che estende *PickableObjectImpl* e fornisce il comportamento comune. Per evitare ridondanze ho usato il pattern *Template Method*: ho implementato *onPick()* in *PowerUpImpl*, che richiama *activate()*, che a sua volta invoca *applyEffect()*, un metodo astratto da implementare all'interno dei singoli power-up. La disattivazione invece è gestita in *isActive()*: se la durata è terminata, invoca *deactivate()*, che richiama *removeEffect()*,

metodo astratto da implementare all'interno dei singoli power-up. Questo approccio permette di gestire il ciclo di vita del power-up in modo centralizzato, lasciando alle sottoclassi solo la definizione dell'effetto specifico. Per rappresentare i diversi tipi ho usato l'enum *PowerUpType*, che consente una classificazione esplicita, utile sia in fase di creazione che di visualizzazione. Il design rispetta l'OCP, permettendo l'aggiunta di nuovi power-up senza modificare il codice esistente.

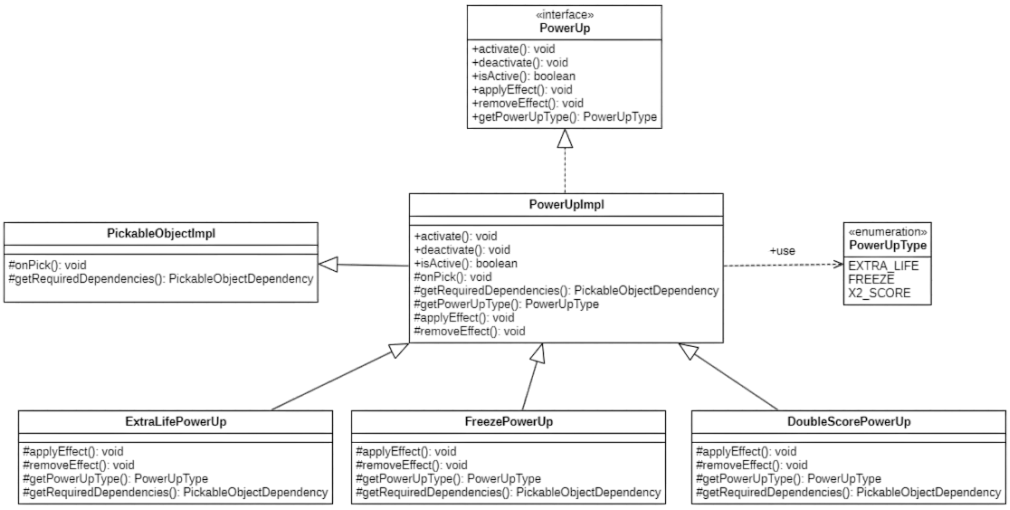

Figura 16: UML del pattern Template Method per modellare i power-up.

Problema Gestire la creazione e il comportamento di oggetti raccoglibili (come monete e power-up), mantenendo il codice modulare ed estendibile.

Soluzione Ho definito l'interfaccia *PickableObject* come contratto per ogni oggetto raccoglibile. La classe astratta *PickableObjectImpl* centralizza la logica comune e viene estesa dalla classe specifica *Coin* e utilizzata da *PowerUpImpl*. La creazione degli oggetti è centralizzata in una *Simple Factory*, chiamata *PickableObjectFactory*, che restituisce un entità coin o un power-up random in base all'input ricevuto. Mentre la gestione del ciclo di vita degli oggetti è delegata a *PickableObjectManager* e alla sua implementazione *PickableObjectManagerImpl*, che ne garantisce l'attivamento e la rimozione. Per

associare a ciascun oggetto l'entità su cui deve agire (es. player), uso l'enum *PickableObjectDependency*, che fornisce un modo esplicito e sicuro per gestire le dipendenze. Questo design è estensibile: per aggiungere un nuovo oggetto, basta estendere *PickableObjectImpl*, aggiungere un valore all'enum, se serve, e aggiornare la factory.

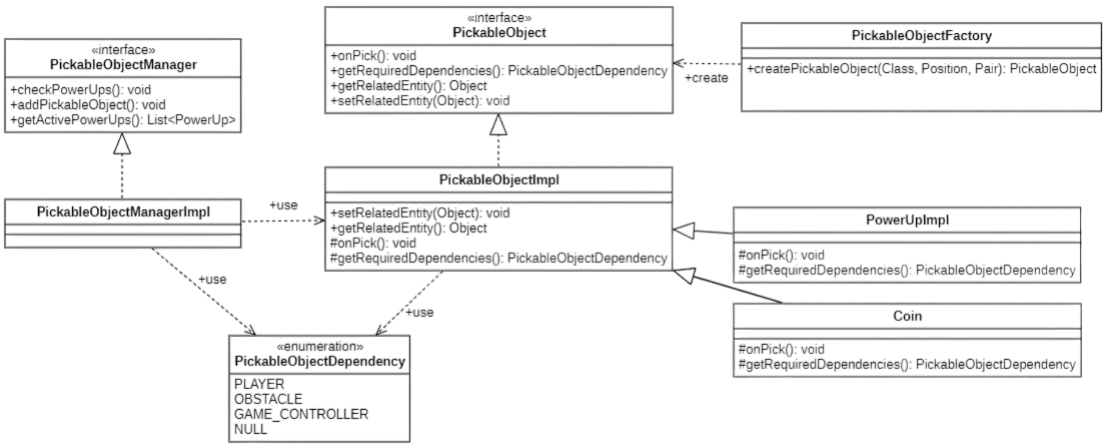

Figura 17: UML della gestione degli oggetti e della factory.

Problema Gestire un singolo menu con un insieme di bottoni, evitando duplicazioni e semplificando la creazione e gestione dell'interfaccia.

Soluzione Ho modellato il menu tramite l'interfaccia *Menu* e la sua implementazione *MenuImp*, che costruisce e controlla un insieme di bottoni basati sugli stati di gioco passati al costruttore. Passando un numero variabile di *GameState*, *MenuImpl* crea i bottoni corrispondenti, posizionandoli e gestendone lo stato visivo (hover, pressed) tramite immagini. Gli eventi del mouse sono catturati da *MouseInput*, che implementa *MouseListener* e *MouseMotionListener* e delega al *MenuController*, responsabile di passare il MouseEvent al menu attualmente attivo, che aggiornerà lo stato dei bottoni controllando la posizione del cursore e attivando le azioni associate.

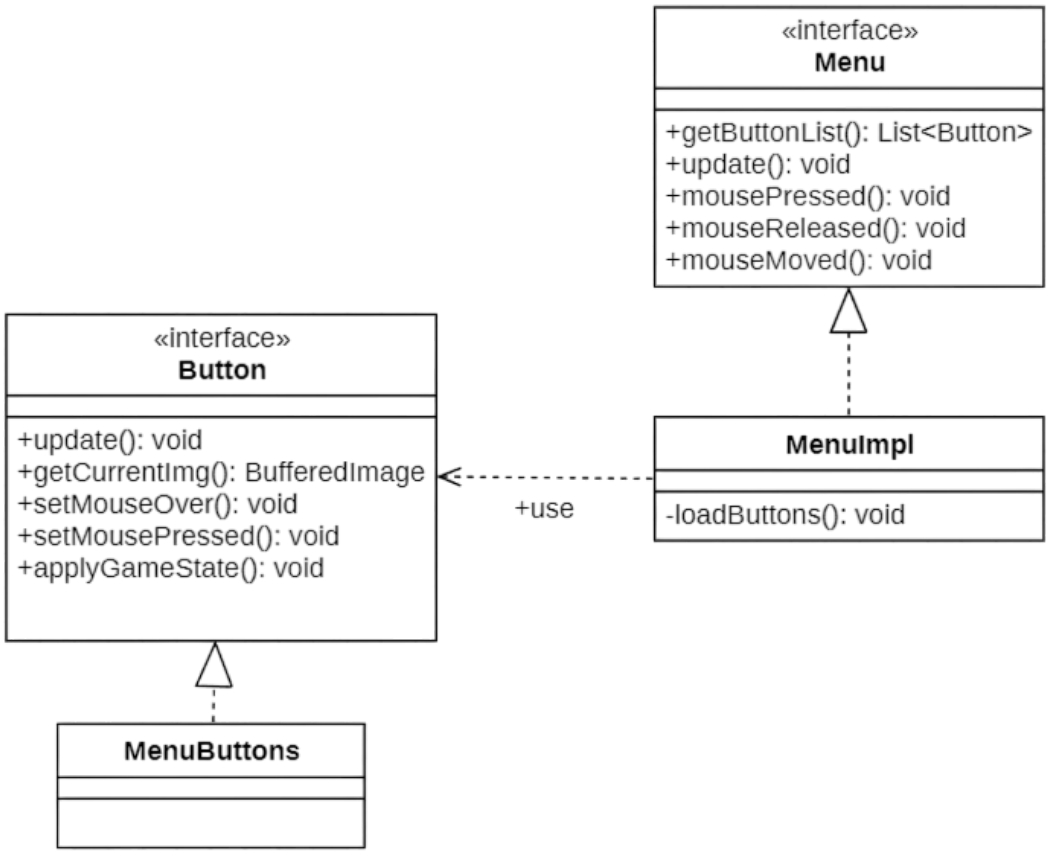

Figura 18: UML del menu.

3 SVILUPPO

3.1 Testing automatizzato

Per la parte degli **spawner randomici** i test effettuati (con l'utilizzo di Mockito) sono:

- **RandomEaglesSpawnerTest**, controlla la validità del numero di acquile generate, della loro posizione e del loro "Trigger row" (cioè al passaggio di quale Lane vengono attivate).
- **RandomObstaclesSpawnerTest**, controlla la validità del numero di ostacoli generato e della loro posizione, prestando particolare attenzione alla larghezza variabile degli ostacoli.
- **RandomPickableSpawnerTest**, controlla la validità del numero di pickable generati e della loro posizione, prestando attenzione agli overlaps tra tipi diversi di pickableObject (Coin, PowerUp).

Per la parte di creazione del **livello**:

- **RandomLevelFactoryTest**, controlla che la generazione del livello sia strutturalmente valida.
- **LevelTest**, controlla che la classe Level memorizzi correttamente le Lane e le entità che le vengono date.
- **LaneTest**, controlla che ogni tipologia di Lane memorizzi solo i tipi di ostacoli che le competono.

Per la parte di gestione degli **oggetti** e **power-up** raccoglibili:

- **PickableObjectManagerTest**: controlla che siano aggiunti, attivati correttamente e la rimozione di power-up terminati.
- **ExtraLifePowerUpTest**: controlla che sia aggiunta correttamente una vita, che le dipendenze e il tipo restituito sia giusto e che non ci sia un eccezzione nel momento che il powerUp non sia "collegato" al player

- **FreezePowerUpTest**: controlla che imposti le velocità a zero all'attivazione e termitato il power-up ripristini le corrette velocità a ciascun ostacolo, che le dipendenze e il tipo restituito sia giusto
- **CoinTest**: controlla che aggiunga le monete correttamente, che la dipendenza sia giusto e che non ci sia un eccezzione nel momento che il coin non sia "collegato" al player

Per la parte di creazione del **Menu**:

- **MenuImplTest**: controlla la creazione, il ripristino di stato dei bottoni (MouseOver e MousePressed) e che i bottoni cambiano lo stato del gioco.
- **MenuButtonTest**: controlla i get del bottone e la sua hitbox, i suoi stati (MouseOver e MousePressed) e che il bottone cambi lo stato del gioco correttamente.

Per la parte di gestione dei **MovingObject**:

- **MovingObjectTest**: controlla la correttezza del muovimento in ogni direzione e il respawn dopo essere uscito dallo schermo
- **MovingObjectFactoryTest**: controlla la corretta creazione di oggetti di ogni tipologia di MovingObject
- **EagleTest**: verifica il corretto funzionamento dell'interfaccia Startable attraverso l'utilizzo della classe Eagle, quindi controlla che non si muova una volta fermato e viceversa, e la corretta impostazione del trigger. Inoltre controlla che l'aquila abbia l'immagine giusta.
- **TrunkTest**: verifica il corretto funzionamento dell'interfaccia Carrier attraverso l'utilizzo di Trunk, quindi controlla l'adeguato settaggio dell'oggetto da trasportare, e che esso si muova con esso solo quando è sopra al Trunk. Inoltre controlla che il tronco abbia l'immagine giusta.

Per il controllo *dell'input* i test sono:

- **InputTest**, controlla che l'input venga ricevuto correttamente e che i controlli sui valori massimi della 'x' e della 'y' funzionino

Per il controllo dello *shop* i test sono:

- **ShopInitTest**, controlla che il file venga letto correttamente e il funzionamento della factory

Per il controllo del *player* i test sono:

- **PlayerObjectTest**, controlla il corretto funzionamento dei metodi di PlayerObjectImpl

3.2 Note di sviluppo

3.2.1 Barzanti Alan

Utilizzo della libreria Mockito: usata in tutti i test degli spawner randomici, qui un esempio permalink: https://github.com/Mouchix/OOP24-ranocchietta/blob/1484e6d54eda8afe7bc68767c6b63947b1ca327b/src/test/java/frogger/RandomEaglesSpawnerTest.java#L51

Utilizzo di Reflection permalink: https://github.com/Mouchix/OOP24-ranocchietta/blob/1484e6d54eda8afe7bc68767c6b63947b1ca327b/src/main/java/frogger/model/implementations/RandomLevelFactory.java#L78

Utilizzo di generici permalink: https://github.com/Mouchix/OOP24-ranocchietta/blob/e0943fec3891e63f3ef023dc3f4b62a65d6ac928/src/main/java/frogger/model/implementations/AbstractRandomEntitySpawner.java#L18

Utilizzo di lambda tramite method reference usati pervasivamente qui un esempio, permalink: https://github.com/Mouchix/OOP24-ranocchietta/blob/e0943fec3891e63f3ef023dc3f4b62a65d6ac928/src/main/java/frogger/model/implementations/RandomLevelFactory.java#L43

Utilizzo di Stream usati pervasivamente qui un esempio, permalink: https://github.com/Mouchix/OOP24-ranocchietta/blob/e0943fec3891e63f3ef023dc3f4b62a65d6ac
src/main/java/frogger/model/implementations/LevelImpl.java#L35

3.2.2 Mariani Federico

Utilizzo di Reflection permalink: https://github.com/Mouchix/OOP24-ranocchietta/blob/df8cf9296aba8ed064bde886d90ba4e8708b70f6/src/main/java/frogger/model/implementations/MovingObjectFactoryImpl.java#L29

Utilizzo di Optional permalink: https://github.com/Mouchix/OOP24-ranocchietta/blob/df8cf9296aba8ed064bde886d90ba4e8708b70f6/src/main/java/frogger/model/implementations/Trunk.java#L16

Utilizzo di lambda tramite method reference permalink: https://github.com/Mouchix/OOP24-ranocchietta/blob/df8cf9296aba8ed064bde886d90ba4e8708b70f
src/main/java/frogger/controller/GameControllerImpl.java#L68

Utilizzo di generici permalink: https://github.com/Mouchix/OOP24-ranocchietta/blob/d2b1e8909aad2847126dfdecbfc1a7f984606aff/src/main/java/frogger/view/AbstractPanel.java#L18

3.2.3 Mondardini Marco

Utilizzo di lambda permalink: https://github.com/Mouchix/OOP24-ranocchietta/blob/1484e6d54eda8afe7bc68767c6b63947b1ca327b/src/main/java/frogger/model/implementations/GameImpl.java#L109

Utilizzo di Stream permalink: https://github.com/Mouchix/OOP24-ranocchietta/blob/df8cf9296aba8ed064bde886d90ba4e8708b70f6/src/main/java/frogger/model/implementations/GameImpl.java#L107

3.2.4 Ferri Lorenzo

Utilizzo di Stream permalink: https://github.com/Mouchix/OOP24-ranocchietta/blob/df8cf9296aba8ed064bde886d90ba4e8708b70f6/src/main/java/frogger/model/implementations/GameImpl.java#L83

4 COMMENTI FINALI

4.1 Autovalutazione e lavori futuri

4.1.1 Barzanti Alan

Per me, come per tutti gli altri, questa è stata la prima esperienza di lavoro in team e ad un progetto reale. In fase iniziale ci siamo trovati decisamente spaesati cosa che ha influito molto sulla scelta del progetto, che oggi ritengo non sia stata molto sfidante da un punto di vista di logica implementativa. Tuttavia questo mi ha dato la possibilità di concentrarmi più che sul codice in sè sul design delle mie classi, che ho potuto pensare e riorganizzare più volte e, seppur sono sicuro non sia perfetto mi rende abbastanza soddisfatto. Per quanto riguarda il lavoro in team credo che il mio apporto sia stato positivo e costante. Se dovesse capitare l'occasione di tornare a lavorare sul progetto mi piacerebbe aggiungere nuovi tipi di creazione del livello visto quanto impegno ho messo nel rendere più semplice possibile un'estensione in tal senso.

4.1.2 Mariani Federico

Nel mio piccolo a livello di sviluppo di software in sè penso che avrei potuto dare di più, ma ciò perché sono state molte altre le sfide che ci si sono presentate.
Penso di aver scritto codice di livello medio anche se il progetto era abbastanza semplice e non richiedeva parti di codici davvero complicate. Nonostante ciò non mi sono accontentato della prima soluzione che mi veniva in mente ma ho sempre provato a migliorare l'idea iniziale.
La sfida maggiore è stata quella di progettazione, che per me e il mio gruppo è risultata davvero ostica soprattutto all'inizio del progetto, infatti ci siamo trovati talmente spaesati sul da farsi da preferire un approccio diverso, più pragmatico e materiale, ovvero assottigliare quasi gravemente la parte di progettazione a favore di quella in cui ci si lancia sul codice un po' a tentoni. Questo però ci ha dato la possibilità di prendere domestichezza con i principi del mvc e della struttura del progetto, che poi col tempo è stata rafforzata e migliorata, a costo di qualche ora di lavoro in più per sistemare le connessioni tra vecchia progettazione e nuova.
Un altra problematica che ho riscontrato è stato il modo di affrontare il progetto del gruppo. Formando il gruppo per amicizia e non per obiettivo ognuno aveva un modo diverso di affrontare il progetto, tanto che qualcuno

ha deciso di lavorarci solo gli ultimi due mesi ritardando considerevolmente la consegna fino ad arrivare al periodo peggiore, ovvero quello degli esami, ancora a dover concludere le ultime cose, spezzando anche il ritmo di lavoro a tutti gli altri.
In futuro mi piacerebbe continuare a sviluppare il progetto, in prima istanza aggiungendo tipi di ostacoli o power up interessanti. Dopodichè vista la semplicità in sè del gioco sarebbe anche bello aggiungere altri "minigiochi" ad esempio per guadagnare monete.
In generale sono molto soddisfatto del lavoro che abbiamo svolto e a ripensare da dove siamo partiti il miglioramento a livello di progettazione, programmazione e lavoro in team è davvero notevole.

4.1.3 Mondardini Marco

Essendo il primo vero grande progetto a cui ho partecipato partendo da zero, all'inizio ero un po' "spaventato" e non riuscivo a immaginare come poter creare questo gioco. Andando avanti però, provando, sbagliando e riprovando, aiutato anche dai miei compagni sono riuscito a trovare una strada. Sono sicuro che quello che ho scritto potrebbe essere ottimizzato o addirittura trasformato per essere più bello e funzionale, ma sono comunque soddisfatto del codice che sono riuscito a scrivere. Realizzare questo progetto mi ha aiutato a capire com'è lavorare in un team, creare un progetto da zero, prima costruendo le basi e poi tutto il resto. Vedere una propria creazione che a mano a mano viene su e si forma è un qualcosa di veramente stimolante. Alla fine abbiamo raggiunto un risultato che non mi dispiace per nulla ed è un qualcosa di cui vado fiero; ovviamente non sarà perfetto essendo la prima esperienza, ma proprio perchè è la prima esperienza è qualcosa di speciale.

4.1.4 Ferri Lorenzo

Nel complesso, il mio contributo al progetto avrebbe potuto essere più efficace. All'inizio ho faticato a entrare pienamente nel vivo del lavoro; trattandosi della mia prima esperienza con un progetto di questo tipo, inizialmente ho avuto qualche difficoltà a orientarmi, non avendo ancora un'idea chiara di cosa dovessi realizzare concretamente. e questo ha avuto qualche ripercussione sull'andamento generale. Con un approccio più tempestivo, forse avrei potuto ottenere risultati migliori. Nonostante ciò, sono soddisfatto di come

si è sviluppato il progetto e ho avuto modo di comprendere con maggiore chiarezza quanto sia fondamentale una buona analisi iniziale.

5 GUIDA UTENTE

All'avvio ci si troverà nel menù principale, per iniziare una partita premere "play". Quando ci si trova in una partita per muoversi nelle 4 direzioni premere le freccette, per mettere in pausa premere esc. È possibile tornare al menù principale dal menù di pausa o da quello di game over. Tornati nel menù principale si può accedere al negozio premendo su "shop". L'indicazione di quante monete si ha a disposizione si trova in alto a destra, mentre per tornare al menù principale è sufficiente premere "menù" in alto a sinistra. All'interno del negozio è possibile acquistare una skin premendo il pulsante ad essa sottostante, se si posseggono più skin è possibile scegliere quale "equipaggiare" tramite le stesso pulsante, che ora dalla dicitura "buy" è passato a "equip".