Relazione OOP

Gym-Life

Andrea De Carli, Luca Camillini, Mattia Morri, Lucio Baiocchi

24 Giugno 2024

Sommario

Questo documento è una relazione che mira a spiegare accuratamente e su più livelli di dettaglio il funzionamento dell'applicativo Gym-Life. La relazione sarà divisa in più sezioni, ognuna delle quali si occuperà di spiegare il funzionamento del software su livelli diversi, per consentire una comprensione completa e accurata. Alcune sezioni saranno divise in quattro parti distinte, una per ognuno dei componenti del gruppo, in modo da avere spiegazioni dettagliate sulla contribuzione dei singoli membri del team. Oltre a descrizioni a parole, ci saranno anche schemi UML per mostrare visivamente la struttura del programma. Alla fine del documento si avranno istruzioni su come operare il software nelle sue diverse parti.

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il software "Gym-Life" è un videogioco 2D in cui l'obiettivo del giocatore è quello di portare il proprio personaggio ad essere il più muscoloso possibile entro un limite di giorni. Per raggiungere questo obiettivo il giocatore dovrà andare in palestra e allenarsi. Si potrà dormire e mangiare cibi per aiutarlo a raggiungere tale obiettivo.

Obiettivi funzionali:

- Si potrà scegliere una difficoltà del gioco, che influenza il numero di giorni disponibili per arrivare alla vittoria.
- Ci saranno 3 location principali: Palestra, Casa e Negozio. il giocatore potrà spostarsi da un luogo all'altro interagendo con la porta di uscita di ogni location.
- All'interno delle location il giocatore si potrà muovere liberamente per interagire con i diversi oggetti.
- Quando il giocatore si sposta da un luogo ad un altro può accadere uno di tanti incontri casuali che gli offrono una scelta.
- Il personaggio ha 3 statistiche principali: Felicità, Stamina e Massa.
- Il gioco sarà vinto nel momento in cui la statistica Massa raggiungerà il suo massimo.
- Il gioco sarà perso nel momento in cui una qualunque statistica principale raggiunga 0.

- Per aumentare il livello di Massa il giocatore deve completare dei mini-giochi in Palestra o facendo determinati incontri casuali durante gli spostamenti.
- Nel Negozio il giocatore può comprare cibi e usare il bancomat per ricevere soldi (con il completamento di un mini-gioco)
- In Casa il giocatore può mangiare i cibi comprati oppure dormire.
- Il giocatore può scegliere la difficoltà per ciascun mini-gioco, la massa muscolare progredirà maggiormente aumentando la difficoltà.
- Il giocatore potrà tenere traccia dei propri risultati negli allenamenti.

Obiettivi non funzionali:

- L'interfaccia del software potrà essere ingrandita o rimpicciolita a preferenza del giocatore.
- Il gioco deve essere multipiattaforma.

1.2 Analisi del Dominio

Le location del gioco sono rappresentate da Mappe, ognuna delle quali è composta da una moltitudine di Celle, le quali contengono le informazioni necessarie come le collisioni o le interazioni. Queste ultime dovranno poter apportare modifiche ad altre componenti del gioco. Ogni Mappa dovrà associare una Cella ad una Posizione, formando una griglia sulla quale il Personaggio si muoverà. Ci sarà bisogno di sapere qual è la mappa attuale e di poterla cambiare, così come per lo scenario di gioco. Servirà tener conto delle singole Statistiche del personaggio. Per i mini-giochi sarà necessario sapere quale far partire e con quale difficoltà. Il mini-gioco della banca sarà gestito diversamente in quanto non necessita la scelta di difficoltà. Servirà inoltre un inventario per sapere la quantità di ogni cibo in possesso del personaggio. L'aggiunta e il consumo di cibi dovranno poter essere attuati tramite una interazione con le celle. Durante il cambio di mappa ci deve essere la possibilità che un incontro casuale possa accadere.
In Figura 1.1 si trova un UML dettagliato degli elementi del dominio del gioco.

1.2.1 Inizio del gioco

Il gioco inizia con una schermata che permette al giocatore di selezionare il livello di difficoltà dell'intera partita. Maggiore la difficoltà, minori saranno i giorni disponibili per completare il gioco; in questo modo il giocatore è costretto a giocare i vari mini-giochi ad una difficoltà maggiore per vincere.

1.2.2 Mappe e Movimento

Le location del gioco sono rappresentate da tre Mappe, ognuna delle quali è composta da una moltitudine di Celle, le quali contengono le informazioni necessarie come le collisioni o le interazioni. Il giocatore potrà muoversi all'interno delle mappe in quattro direzioni: UP, DOWN, RIGHT, LEFT, e sarà limitato esclusivamente dagli oggetti su cui non è possibile camminare.

1.2.3 Statistiche

Per vincere il personaggio deve raggiungere il livello massimo di massa, e questo avviene principalmente attraverso l'allenamento o secondariamente con incontri casuali durante il cambio di scenario. Per evitare che le altre statistiche raggiungano lo zero, e che quindi il giocatore perda, è indispensabile mangiare e dormire per mantenere alti anche i livelli di stamina e umore.

1.2.4 Cibi e soldi

All'interno del gioco ci sono cibi di tre diverse tipologie. Possono essere acquistati al supermercato con i soldi in possesso del personaggio. Cibi diversi modificano in maniera diversa le statistiche, rendendo fondamentale un giusto equilibrio tra di loro. Dopo l'acquisto sono visibili nell'inventario, e per essere effettivamente mangiati è necessario cucinarli in casa. Se il giocatore non ha abbastanza soldi può iniziare un minigioco nell'ATM al supermercato per guadagnarne altri.

1.2.5 Mini-giochi palestra

All'interno della palestra, il giocatore avrà l'opportunità di cimentarsi in tre mini-giochi principali: panca piana, Lat Machine e Squat. Ognuno di questi mini-giochi è progettato per migliorare specificamente la massa muscolare associata al relativo macchinario utilizzato. Il giocatore potrà scegliere tra diversi livelli di difficoltà per ogni allenamento. Optare per un

livello di difficoltà maggiore permetterà di ottenere una crescita muscolare più significativa, ma aumenterà anche il rischio di fallimento. In caso di fallimento, il giocatore non solo non guadagnerà massa muscolare, ma la perderà. Ci sarà anche la possibilità di monitorare i propri migliori tempi per ogni esercizio, registrandoli per ciascun livello di difficoltà selezionato.

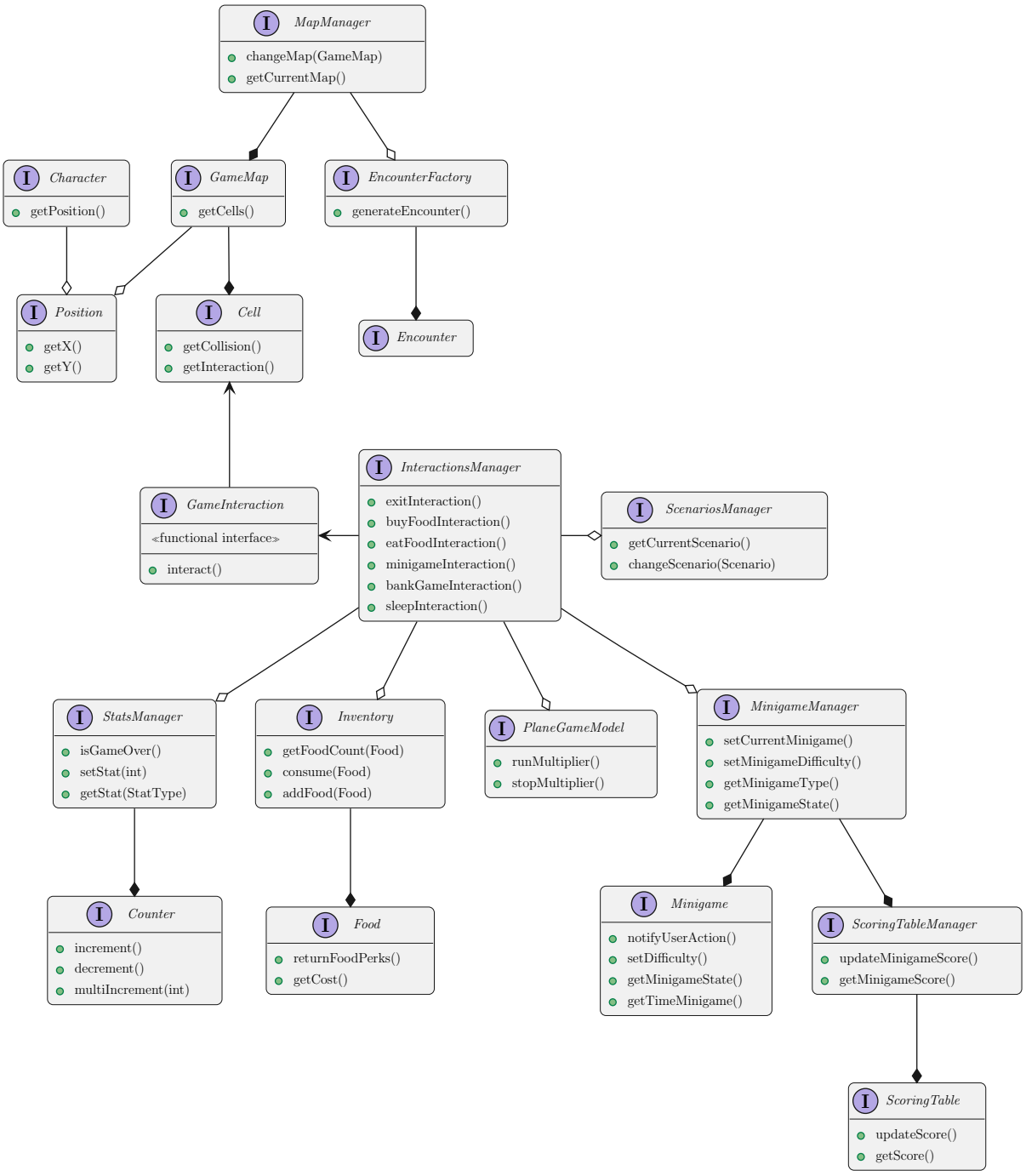

Figura 1.1: Schema UML del dominio del Software

Capitolo 2

Design

2.1 Architettura

Per il progetto *"Gym-Life"* si è adottata una architettura di tipo *MVC*, consistente in una parte di Model, che contiene tutta la parte puramente logica del nostro software, una *View* che costituisce la parte di interfaccia grafica ed un *Controller* adibito a far comunicare le due altre parti. Le componenti di *View* ricevono gli input dell'utente che vengono elaborati in istruzioni per il *Controller*, il quale modifica il *Model*. Per aggiornarsi, la *View* aspetta uno specifico input, che gli comunica di "chiedere" al *Controller* in che modo deve aggiornarsi e lo fa di conseguenza. Il *Controller* ha il compito di contenere tutte le istanze degli elementi e i metodi necessari per apportare modifiche al Model, fungendo da collegamento tra le altre due componenti architetturali. *"Gym-Life"* è progettato in maniera che le Parti di *View* non siano influenzate da come è fatto il *Controller*, né tanto meno dal *Model*, anch'esso non dipendente dal *Controller*, rendendo così possibile cambiare libreria grafica mantendendo senza modificare *Model* e *Controller*. Inoltre è presente una sezione chiamata *Utilities* che contiene diverse componenti utili al funzionamento del software, contiene per esempio costanti che possono servire in qualunque sezione architetturale. La Figura 2.1 presenta una visione dettagliata con metodi e interfacce dell'architettura del progetto.

2.1.1 Model

Il *Model* contiene tutte le parti di "Entità" del progetto e i metodi necessari a modificarla. Questa parte è indipendente da come è sviluppato il *Controller* in quanto mette a disposizione tutti i metodi che servono per apportare modifiche alla logica del gioco accessibili esternamente.

2.1.2 Controller

Il *Controller* è la parte architetturale che contiene tutte le effettive istanze degli oggetti del *Model* e si occupa di applicare modifiche su di loro utilizzando i loro specifici metodi. In questo modo il *Model* non è influenzato in alcun modo da come è fatto il *Controller* e una implementazione differente è facilmente integrabile.

2.1.3 View

La *View* è strutturata in modo da avere una interfaccia principale che coordina lo switching delle altre sub-interfacce. Ogni Sub-Interfaccia, quando deve essere sostituita da un'altra, comunica all'interfaccia principale la necessità di switching. A questo punto l'interfaccia principale chiede al *Controller* qual è la sub-interfaccia da mostrare e lo fa. Perciò se si dovesse sostituire l'implementazione della *View*, lo si dovrebbe fare utilizzando questo sistema di richiesta al *Controller*, dato che il gioco funziona a "scenari" e lo scenario attuale è mantenuto a livello di *Model*.

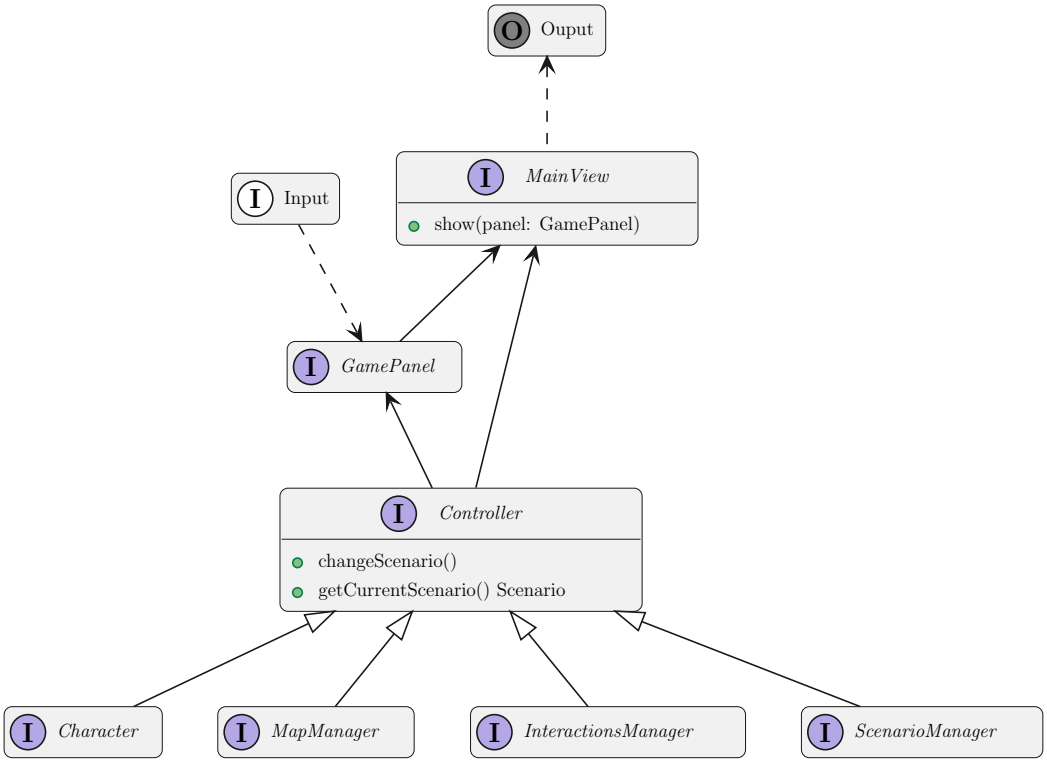

Figura 2.1: Schema dettagliato di come l'architettura MVC gestisce gli input e output.

2.2 Design dettagliato - Andrea De Carli

Nel progetto il mio compito era quello di occuparmi delle mappe, celle ed interazioni per quanto riguarda il model. Ho inoltre contribuito abbastanza alla gestione delle interfacce grafiche nella parte di View come per la MainView e GamePanel. In particolare in questa sezione parlerò di:

- Mappe e celle.
- Oggetti interagibili.
- Pannelli di scelta nella view.

2.2.1 Mappe e Celle

Descrizione del Problema.

Una delle parti del gioco che dovevo sviluppare riguarda il caricamento delle mappe e la loro successiva gestione: il passaggio da una mappa ad un'altra, mantenere il layout di celle in ogni mappa costante e memorizzare per ogni cella informazioni utili. Per come è stata pensata la realizzazione del gioco si avranno un totale di 3 mappe, perciò si deve trovare una soluzione che consenta di creare diverse mappe nello stesso modo ma con celle diverse. Parlando di Celle anch'esse dovranno essere molteplici e ognuna dovrà contenere gli stessi tipi di informazioni riguardo le collisioni e interazioni.
Soluzione.
Per risolvere questi problemi ho deciso di sfruttare ampiamente gli *Enum* di Java, che consentono di definire la struttura di un oggetto e di poi crearne le istanze direttamente nella classe stessa. Le celle sono un *Enum* con 3 campi: *id*, *collision* e *interaction*. Questo mi permette di aggiungere o togliere una cella con estrema semplicità creando l'istanza nell'*Enum* stesso e definendo i valori dei campi. Il campo *id* serve per il caricamento della mappa da file, infatti per fare in modo che il layout delle celle in ciascuna mappa rimanga tale e che sia facilmente modificabile esternamente al codice, ho deciso di creare dei file .txt che contengono una matrice di interi, ognuno dei quali rappresenta l'*id* di una cella. La classe *MapLoader*, nelle *utility*, dispone un metodo *load()* che dato il nome della mappa, legge il file corrispondente e ritorna una *Map* di posizioni e celle. Ho deciso di tenere separati gli oggetti Cella e Posizione per fare in modo che una non dipendesse dall'altra: se ogni cella avesse una posizione predefinita sarebbe molto laborioso andare a modificare le mappe e inoltre non potrei avere celle ripetute. Anche le mappe sono create dentro un *Enum*, il che presenta gli stessi vantaggi delle celle. In Figura 2.2 si trova uno schema UML di come sono strutturate le mappe.

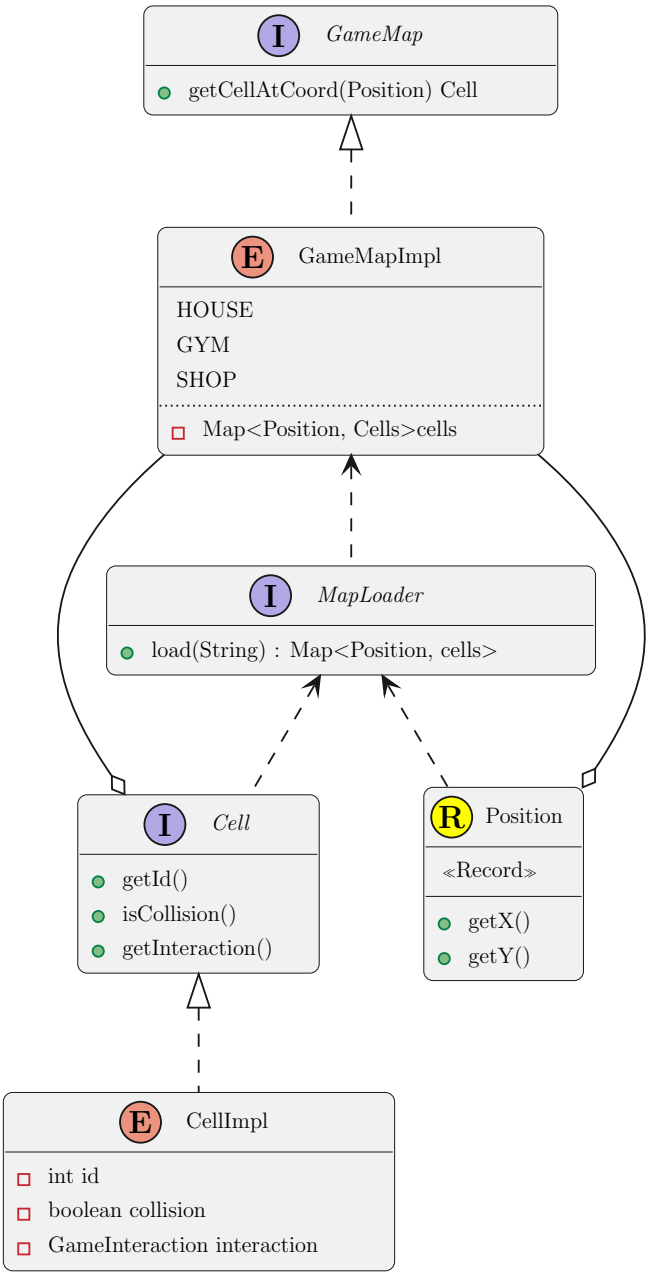

Figura 2.2: Schema di come sono strutturate le classi che gestiscono le mappe

2.2.2 Interazioni

Descrizione del Problema.
Alcune celle delle mappe, quando il personaggio decide di interagirci, devono eseguire diverse azioni o modifiche al gioco. Ciò che rende difficile questo problema è l'eterogeneità delle azioni che devono essere eseguite e anche il riconoscere quale cella deve fare quale azione.
Soluzione.
Ho utilizzato una combinazione di due design pattern per la risoluzione di questo problema: *Strategy* e *Mediator*. Inizialmente il piano era di mettere dei method-reference nel campo *interaction* di *Cell*, che facevano riferimento a metodi delle classi interessate a tale interazione. Per esempio per la cella di consumo del cibo ci sarebbe stato il riferimento al metodo *consume()* della classe *Inventory*. Ma dato che ogni method-reference sarebbe stata di tipo diverso, non era possibile, data l'uniformità dei tipi negli *Enum*. Allora la soluzione a cui sono arrivato è quella di usare una classe che funga da "filtro" chiamata *InteractionsManager*. Questa classe prende da costruttore i riferimenti agli oggetti che saranno modificati dalle interazioni, le quali sono rappresentate da metodi pubblici nella classe (Esempio: *buyFoodInteraction()* che apporta modifiche all'inventario). In questo modo quando si crea una method-reference di questi metodi sono tutte dello stesso tipo, nonostante lavorino su oggetti diversi. Facendo così si applica con successo il Mediator pattern con la classe *InteractionsManager* come mediatore. L'utilizzo di Lambda come valori del campo *interaction* di *Cell* è in sé l'applicazione di un altro pattern: Strategy, grazie anche all'interfaccia funzionale *GameInteraction* che è essenzialmente un Consumer ma di una classe specifica (*InteractionsManager*). Avere la gestione delle interazione strutturata in questo modo mi consente di farle eseguire con una singola chiamata a interaction nel Controller, senza l'utilizzo di if o switch. In Figura 2.3 si trova uno schema UML dettagliato di come è strutturata la parte delle interazioni.

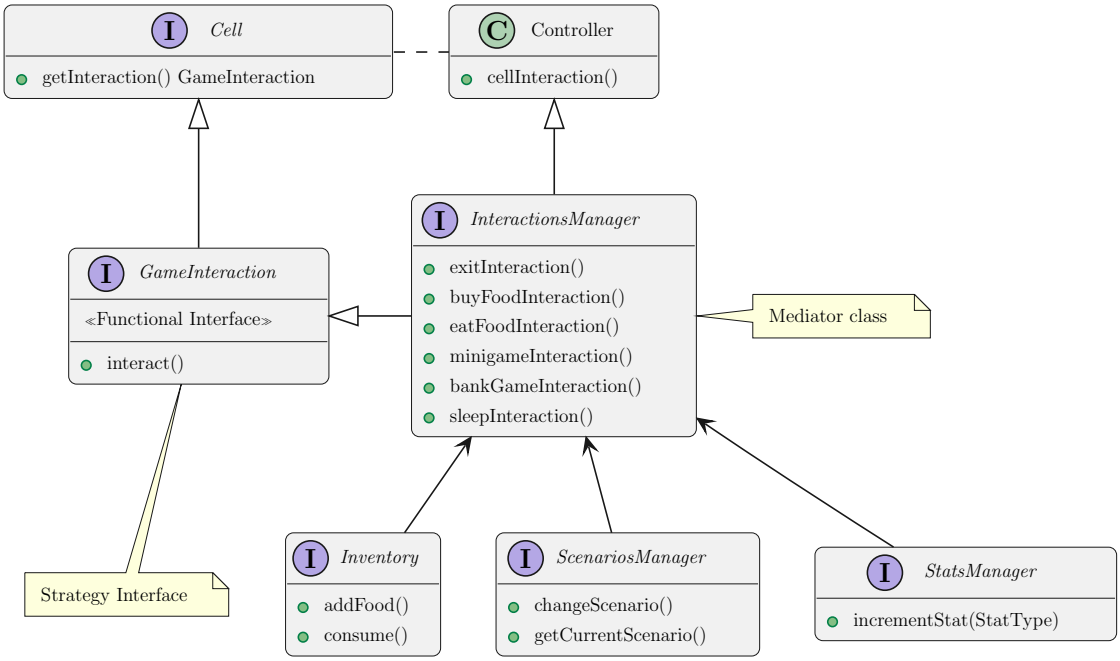

Figura 2.3: Schema UML del funzionamento delle interazioni

2.2.3 Pannelli di Scelta

Descrizione del Problema.
All'interno della *View* ci sono più interfacce grafiche che presentano la stessa struttura: una scelta binaria. Un semplice pannello con una breve descrizione, una immagine e due bottoni. Il problema è che questo tipo di pannello dovrà comparire più volte ma con leggere modifiche: cosa devono fare i bottoni, cosa deve essere scritto sui bottoni e nella descrizione e quale immagine far apparire.
Soluzione.
Per risolvere questo problema ho pensato di creare una classe astratta chiamata *ChoicePanel* e utilizzare dei metodi astratti per ottenere le stringhe di descrizione, messaggi dei bottoni e l'immagine. *ChoicePanel* ha un metodo non astratto *resizeComponents()* che carica tutti i componenti grafici utilizzando i metodi astratti. Facendo così ogni classe che estende *ChoicePanel* può implementare i metodi astratti come vuole, e funziona sempre. Le azioni eseguite dai bottoni invece non possono essere definite in metodi astratti perché gli *ActionListener* devono essere definiti nel costruttore. Strutturare *ChoicePanel* in questo modo è una applicazione del Design Pattern Template, con il metodo *resizeComponets()* che funge da metodo

template. In Figura 2.4 si ha uno schema UML che illustra il sistema di *ChoicePanel*.

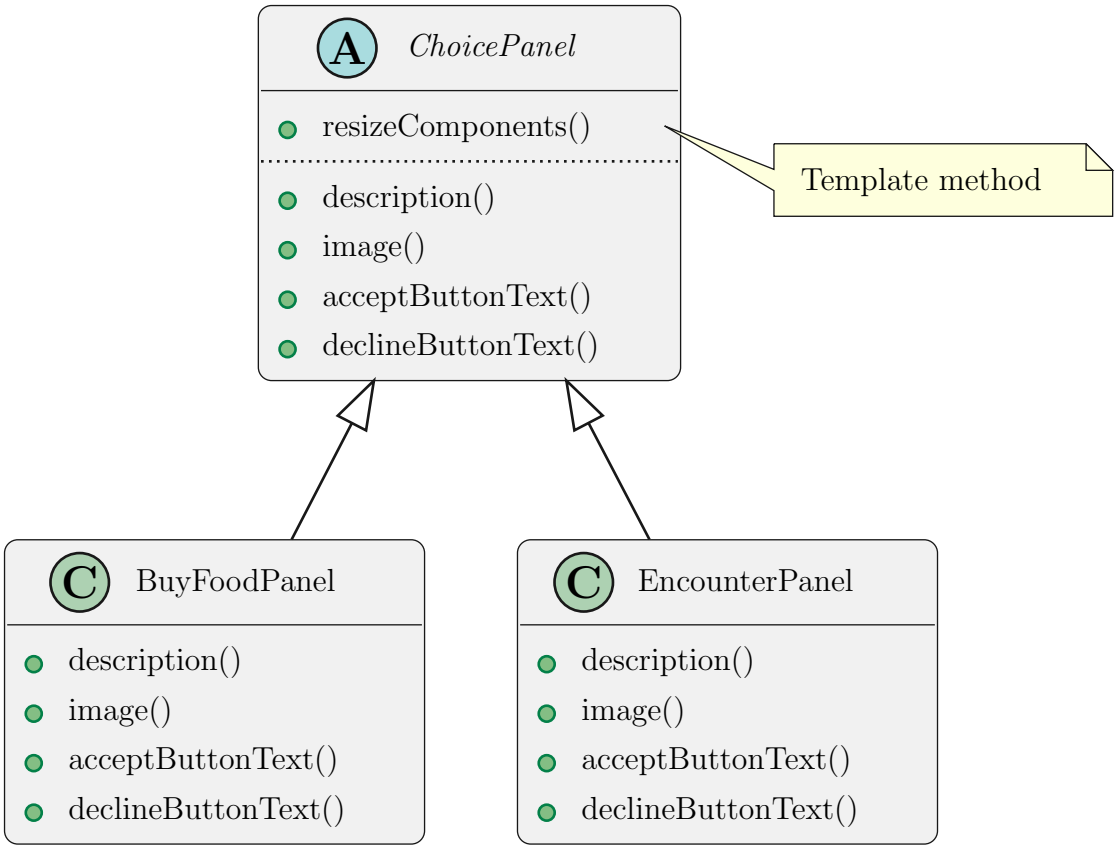

Figura 2.4: Schema UML di come è strutturata la parte di view dei pannelli a scelta binaria

2.3 Design dettagliato - Lucio Baiocchi

All'interno del progetto il mio compito era quello di gestire tutte le statistiche all'interno del gioco, e quindi anche la logica che sta dietro al game over o alla vittoria. Mi sono anche occupato della gestione del movimento del personaggio all'interno della mappa. Infine ho dovuto anche gestire gli incontri casuali e la loro creazione.

2.3.1 Statistiche

Descrizione del Problema. Capire come ottimizzare la classificazione delle varie statistiche in maniera tale da evitare il più possibile ripetizioni di codice e riutilizzare porzioni di codice comune.

Soluzione. Per fare questo ho utilizzato un approccio bottom-up, classificando in maniera generale tutte le statistiche e identificando gli aspetti comuni di quest'ultime, rendendole dei *GameCounter* che non assumono mai un valore minore di zero e partono tutte da un valore variabile. Il numero dei giorni e la quantità dei soldi sono incaspulate in modo ottimale in un semplice contatore, ma le restanti statistiche *(LEG MASS, CHEST MASS, BACK MASS, MASS, HAPPINESS, STAMINA)* necessitano di alcuni metodi aggiuntivi, hanno infatti tutte la caratteristica di avere un limite prestabilito. Il giocatore può scegliere di continuare ad allenarsi nonostante abbia raggiunto il valore massimo di petto e sta a lui sfruttare nella maniera ottimale tutti i giorni di gioco per ottimizzare la crescita muscolare. Per questo motivo tutte queste statistiche sono dei *LimitedGameCounter* che estendono i metodi di base di un *GameCounter*, ma non possono mai superare il limite prestabilito.

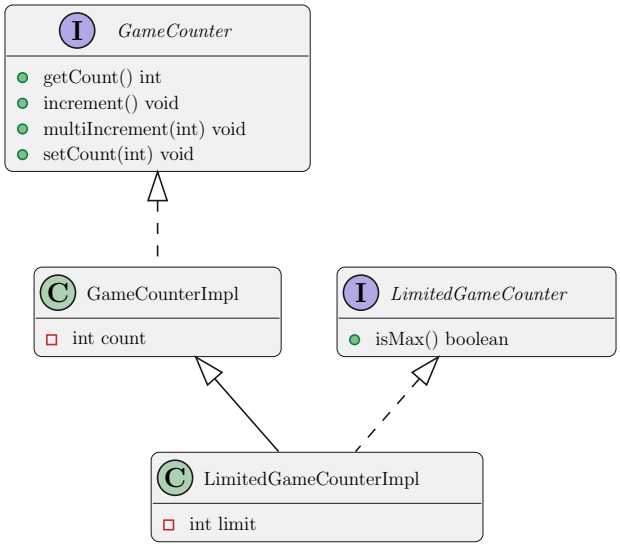

Figura 2.5: Schema UML della classe *Counter* e *LimitedCounter*

2.3.2 Logica Statistiche

Problema: Una volta identificate le due tipologie di statistiche si è posto il problema di capire come gestire la logica dietro al loro cambiamento durante il corso del gioco. Il loro valore infatti determina l'esito della partita, e inoltre devono essere modificate in seguito a derminati eventi, come la fine di un minigioco, un incontro casuale o dopo aver mangiato un cibo.

Soluzione: Ho pensato di raggruppare tutte le statistiche comuni (per comuni sono intese tutte quelle statistiche che sono *LimitedCounter*, per esempio *LEG MASS* o *HAPPYNESS*) all'interno della classe *CommonStats*. Questa classe funge sostanzialmente da container, contendendo al suo interno tutte le statistiche comuni associate al loro *LimitedGameCounter*. Inoltre ho reso la massa totale composta dalla somma delle tre masse (*LEG MASS, CHEST MASS e BACK MASS*), attraverso l'uso di una classe anonima.
Arrivato a questo punto manca la gestione di tutte le statistiche e la loro logica in un'unica classe. Per risolvere questo problema ho utilizzato il pattern manager, nello specifico *StatsManager*, che si occupa di contenere *CommonStats*, il numero di giorni e di soldi.
Il manager ha i getter e setter di ogni statistica che può essere modificata, e i metodi per capire se il gioco è vinto o perso.
In questo modo le statistiche all'interno del gioco sono modificate solo attraverso lo *StatManager*, limitando anche determinate operazioni per ogni

statistica, per esempio i giorni sono un contatore, ma essendo incapsulati all'interno del manager possono essere solo decrementati di uno e non settati/incrementati, inoltre partono da un valore determinato in base alla difficoltà del gioco(passato da costruttore al manager).
Per associare il concetto di difficoltà a un numero di giorni ho utilizzato un *Enum*.

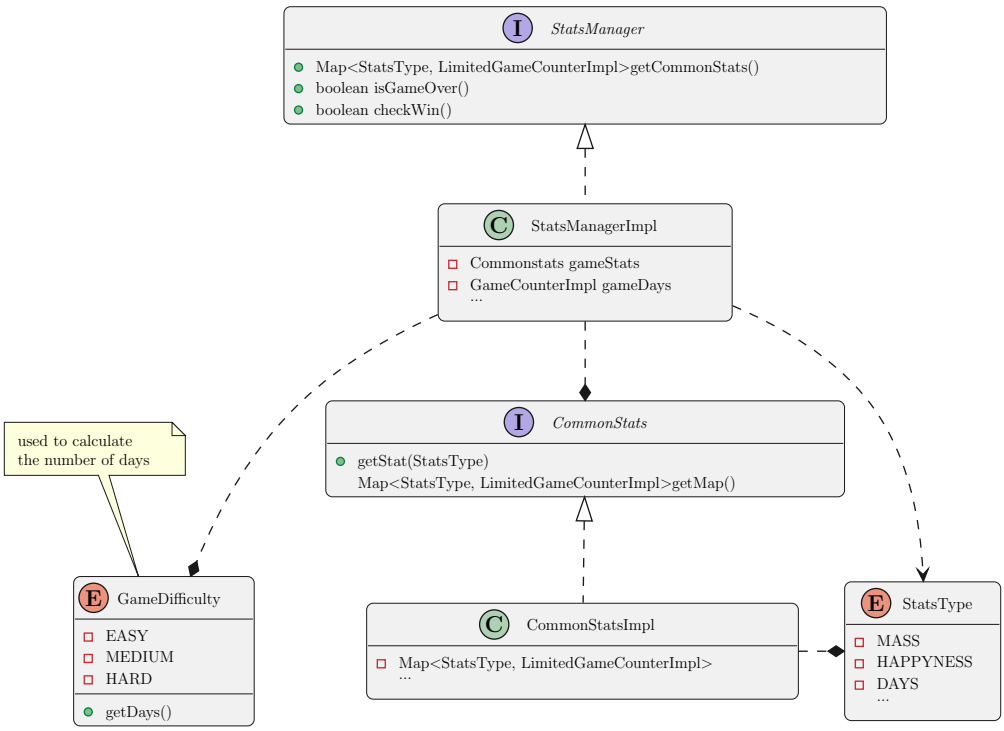

Figura 2.6: Schema UML della classe *StatsManager*

2.3.3 Incontri casuali

Descrizione del Problema. Fare avvenire la creazione dei vari incontri ognuno con probabilità diversa e come incapsulare le varie informazioni relative ad uno specifico incontro.

Soluzione. Come per le statistiche, sono partito analizzando il problema dalle basi. Ho creato quindi il record *Encounter* che rappresenta uno dei cinque incontri casuale del gioco; Contiene due mappe che sono composte da una coppia chiave valore di *StatsType* e un *Integer* che servono per modificare le statistiche in due modi diversi a seconda che l'incontro sia stato accettato o meno. L'incontro ha anche un nome e una breve descrizione di quest'ultimo.
Per risolvere il problema della creazione di tali incontri ho fatto ricorso all'uso del pattern *Factory*, all'interno di *EncountersFactory*. Questa classe si occupa di creare un *Optional* composto da un qualsiasi incontro casuale. Ogni volta che viene chiamato il metodo per generare incontri si ha una determinata probabilità di farne uno, se l'incontro avviene verrà poi costruito uno tra i 5 disponibili. Gli incontri hanno ognuno una probabilità diversa di avvenire. Tutte le varie probabilità sono salvate in constanti all'interno di *EncountersCostants* con anche tutti gli altri campi necessari alla creazione di un incontro. In sintesi, la combinazione della classe *Encounter* per rappresentare gli incontri e della classe *EncountersFactory* per generarli, insieme all'uso delle costanti definite in *EncountersConstants*, fornisce una soluzione strutturata e flessibile per gestire gli incontri casuali nel gioco.

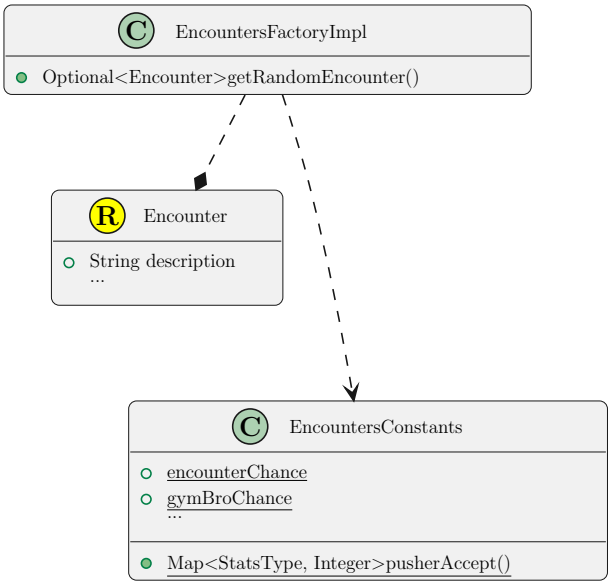

Figura 2.7: Schema UML della gestione degli *Encounter*

2.3.4 Movimento Personaggio

Problema: Gestire il movimento del personaggio all'interno della mappa, e tradurre i tasti premuti nelle 4 direzioni di spostamento *(UP, DOWN, RIGHT, LEFT)*. Quando il personaggio esce da uno scenario ed entra in un altro la posizione in cui viene caricato è diversa (questo è dovuto alla struttura delle varie mappe) perciò non è sufficiente poter solo muovere il personaggio. Inoltre un altro obbiettivo è quello di limitare il più possibile la dipendenza tra personaggio e mappa.

Soluzione: Sono partito dal creare il record *CharacterImpl* che incapsula l'idea del personaggio associato ad una posizione. La posizione verrà poi associata ad una cella della mappa, ma solo nella parte di *View*, garantendo l'indipendenza funzionale tra mappa e personaggio. La classe *CharacterImpl* permette di muovere il personaggio con il metodo *moveCharacter* tramite il passaggio di una delle 4 direzioni.
Le direzioni sono salvate all'interno dell'enum *Direction*. Ad ogni direzione è anche associato il tasto della tastiera corrispondente e un offset dovuto allo spostamento nella data direzione.
Per risolvere il problema dovuto al cambiamento degli scenari ho dovuto aggiungere un metodo per "settare" la posizione del personaggio ogni volta che viene modificato lo scenario.

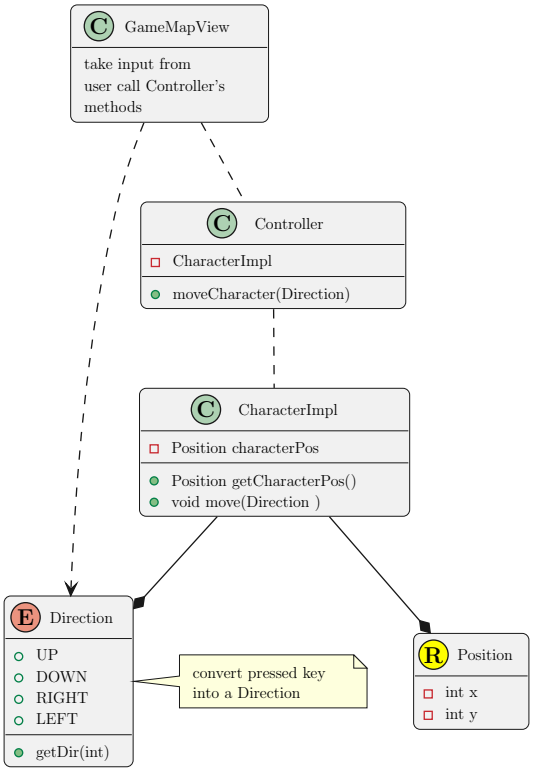

Figura 2.8: Schema UML della gestione del movimento del *Character*

2.4 Design dettagliato - Luca Camillini

Personalmente, all'interno del progetto, mi sono occupato della completa realizzazione dei tre mini-giochi. Questo ha comportato la gestione di ogni azione del giocatore una volta che ha interagito con uno di essi. In particolare, ho curato:

- La scelta della difficoltà dell'esercizio;
- Lo svolgimento dell'esercizio stesso;
- Esito dell'esercizio con annesso aggiornamento del risultato e della tabella dei tempi.

2.4.1 Scelta dei mini-giochi

Descrizione del problema.
Il problema principale che ho dovuto affrontare è stato creare un sistema flessibile e mantenibile per gestire diversi mini-giochi. Ogni minigioco ha

logiche differenti e il sistema doveva essere progettato in modo tale da permettere l'aggiunta di nuovi mini-giochi senza dover apportare modifiche significative al codice esistente. Inoltre, volevo garantire che i mini-giochi potessero essere sostituibili o rimovibili facilmente.

Soluzione.

La soluzione proposta utilizza il pattern *Strategy* per incapsulare le logiche dei mini-giochi in classi separate, ciascuna delle quali estende una classe astratta comune, *Minigame*. Questo approccio permette di passare una strategia, ovvero uno dei diversi mini-giochi, e creare un'istanza appropriata.

Per implementare questa soluzione, ho utilizzato un *Enum MinigameType*, che contiene i tre mini-giochi e come attributi le relative strategie. Questo *Enum* serve a identificare i diversi tipi di mini-giochi disponibili.

Ho inoltre sviluppato una classe di supporto chiamata *MinigameManager*, che è responsabile della gestione delle istanze dei mini-giochi. Questa classe è stata poi instanziata nell'*InteractionManager*.

Il flusso operativo è il seguente:

- L'utente interagisce con un minigioco;
- L'*InteractionManager* riceve il tipo di minigioco selezionato e lo passa al *MinigameManager*;
- Il *MinigameManager* crea un'istanza del minigioco corrispondente;
- Il minigioco viene avviato e gestito tramite il *MinigameManager*.

Questo sistema permette una facile aggiunta, sostituzione o rimozione di mini-giochi senza modificare significativamente il codice esistente. Inoltre, mantiene il codice organizzato e facilmente manutenibile. In questo modo, in caso si voglia aggiungere un nuovo minigioco basterà creare un *Enum* in *MinigameType* e creare la classe.

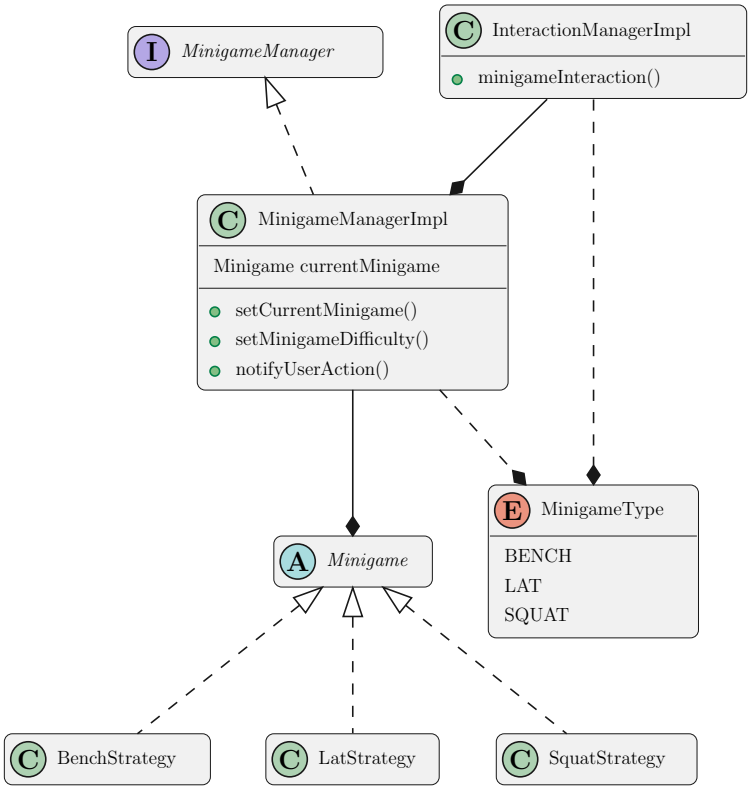

Figura 2.9: Schema UML della gestione mini-giochi

2.4.2 Implementazione mini-giochi

Descrizione del problema.
L'implemetazione vera e propria dei mini-giochi è stata la parte che più mi ha messo in difficoltà perchè notavo che tra di loro c'erano elementi di logica comuni e altri diametralmente opposti.
Soluzione
Per risolvere questo problema ho utilizzato la combinazione di due patter: *Strategy* e *State*. Il pattern *Strategy*, come già spiegato nel problema di scelta dei mini-giochi, ha il compito di creare l'istanza relativa al minigioco. La classe *Minigame* (estesa da tutte le *strategy*) implementa un'interfaccia *MinigameStateHandler*: come dice il nome mi ha aiutato a gestire i vari stati dei mini-giochi. Ogni stato è un *Enum MinigameState*. Quindi, ad ogni interazione dell'utente il minigioco viene notificato tramite il metodo *notifyUserAction()*. Questo metodo utilizza una mappa con chiavi gli *Enum* e come *Value* dei *Consumer* che, data come chiave lo stato attuale del minigioco eseguono il metodo corrispondente ad essi. Questa implemen-

tazione mi ha permesso di poter mettere nella classe astratta *Minigame* tutti gli stati che i mini-giochi avevano in comune tra di loro e di gestire separatamente nelle rispettive sottoclassi gli stati diversi. Per le statistiche all'interno dei mini-giochi ho raccolto quelle in comune in un record *MinigameStatistics*. Mentre il resto le ho istanziate all'interno delle classi *Strategy*.

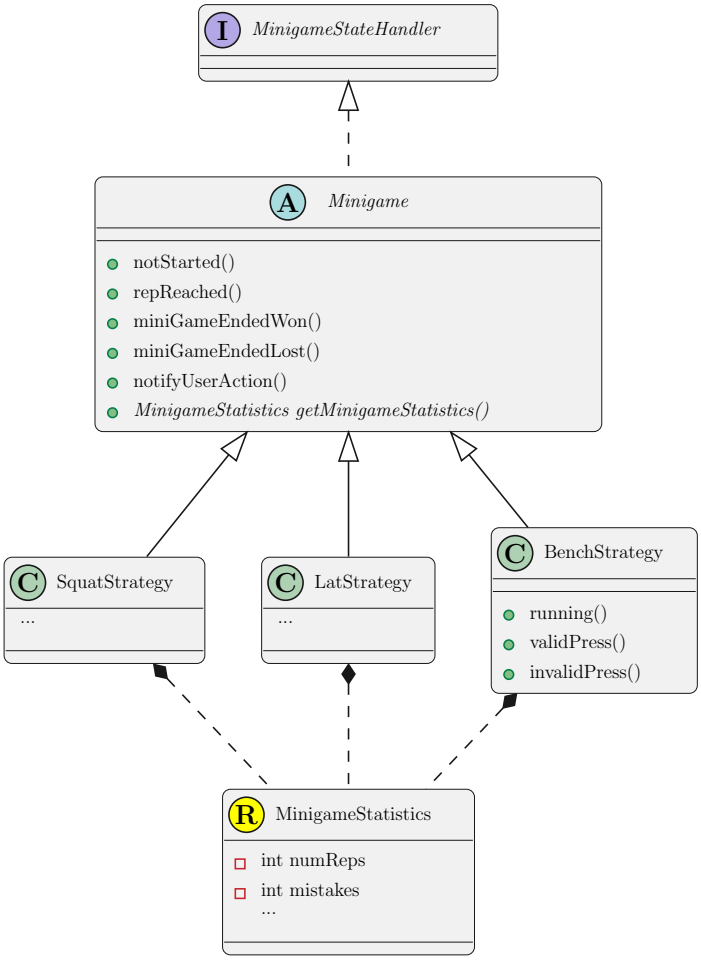

Figura 2.10: Schema UML implementazione mini-giochi

2.4.3 Gestione view dei mini-giochi

Descrizione del problema.
Una volta avviato il minigioco, il giocatore visualizzerà una schermata per la selezione della difficoltà. Dopo aver scelto una delle opzioni disponibili, il gioco procederà con la schermata del minigioco vero e proprio. Il proble-

ma che ho dovuto affrontare è in che modo cambiare i pannelli tra di loro. Al termine dell'esercizio, verrà mostrata una schermata di fine minigioco che indicherà l'esito, specificando se il giocatore ha vinto *(WIN)* o perso *(LOSE)*.

Soluzione.

Per cambiare tra di loro le *View*, utilizzo un *CardLayout*. Appena *MinigameSwitchView* acquisisce il focus dalla *MainView*, mette in primo piano la schermata di scelta delle difficoltà. Questa schermata, come tutte le altre *View*, estende *GamePanel* e quindi *JPanel*. Alla creazione di *DifficultyMenu*, nel costruttore gli vengono passati i tre *ActionListener* da assegnare ai bottoni di scelta della difficoltà. In questo modo, premendo uno dei bottoni delle difficoltà, l'*ActionListener* (tramite il *CardLayout*) mette in primo piano la *View* del minigioco.

Anche per decidere il *Panel* di quale minigioco creare ho utilizzato lo *Strategy* come nel *Model*, con l'unica differenza che il *MinigameType* non gli viene passato direttamente dal *Model* (altrimenti si avrebbe una violazione del paradigma *MVC*), ma quando viene richiamato il metodo *startMinigameView()* la *View* chiederà al *Controller* quale minigioco è stato creato nel *Model* e provvederà a creare la *Strategy* corrispondente.

Non potevo applicare la stessa soluzione anche nello scambio tra il minigioco e la schermata di fine perché non potevo assegnare *ActionListener* a tre tipologie di mini-giochi differenti. Per questo motivo, ho lavorato con la visibilità e il pattern *State* spiegato nel problema precedente: ho utilizzato un *ComponentAdapter* con un metodo *componentHidden()* che comunicava al *CardLayout* di switchare alla *View* di fine minigioco quando il *MinigameState* è ENDED_WIN o ENDED_LOST. In questo modo, il pannello del minigioco veniva settato a invisibile. Infine, per tornare alla *MainView*, ho aggiunto un *ComponentAdapter* alla *MinigameEndView*, che trasferiva il focus di nuovo alla *MainView*.

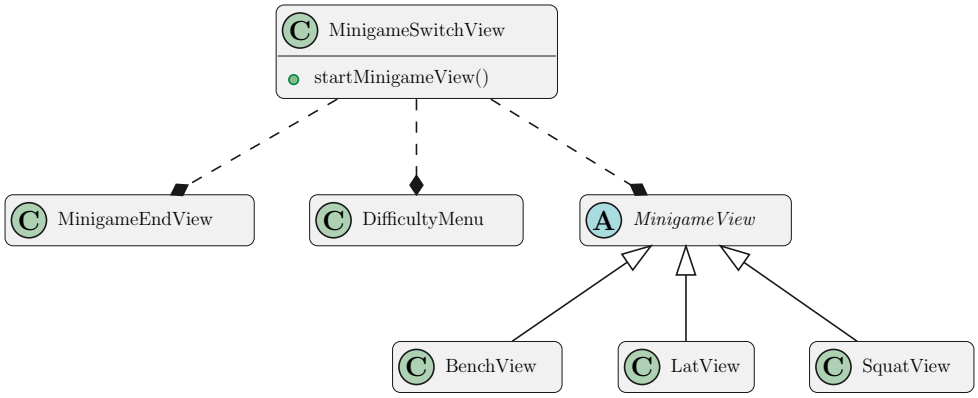

Figura 2.11: Schema UML gestioni schermate all'interno dei mini-giochi

2.4.4 ScoreBoard

Descrizione del problema.
Tenere traccia dei cinque migliori tempi del giocatore nei vari mini-giochi e nelle varie difficoltà.
Soluzione.
Per prima cosa ho creato la *ScoringTable* corrispondente ad un singolo mini-gioco. I tempi sono tenuti all'interno di una mappa in modo da distinguerle per difficoltà. Quando si va ad aggiungere un tempo la tabella controlla che ce ne siano non più di cinque, ovviamente in caso eliminerebbe il tempo maggiore. Una volta creata la *ScoringTable* per un mini-gioco mi è bastato usare una classe di appoggio *ScoringTableManager* dove ho creato un oggetto per tipo di mini-gioco. Questa soluzione, proprio come il caso della scelta dei mini-giochi mi permette di creare *Scoreboard* solamente aggiungendo un *Enum* di mini-gioco. Il *Manager* è all'interno di *MinigameManager*. Lo score si aggiorna ogni volta che l'utente esce dal mini-gioco: nel metodo *setResult()* del *Controller* c'è anche una chiamata al *MinigameManager* per notificarlo di aggiornare i tempi.
La *View*, invece, ogni volta che si clicca su un tipo di mini-gioco e su una determinata difficoltà richiede al *Model* la tabella richiesta dall'utente attraverso *getScores()*.

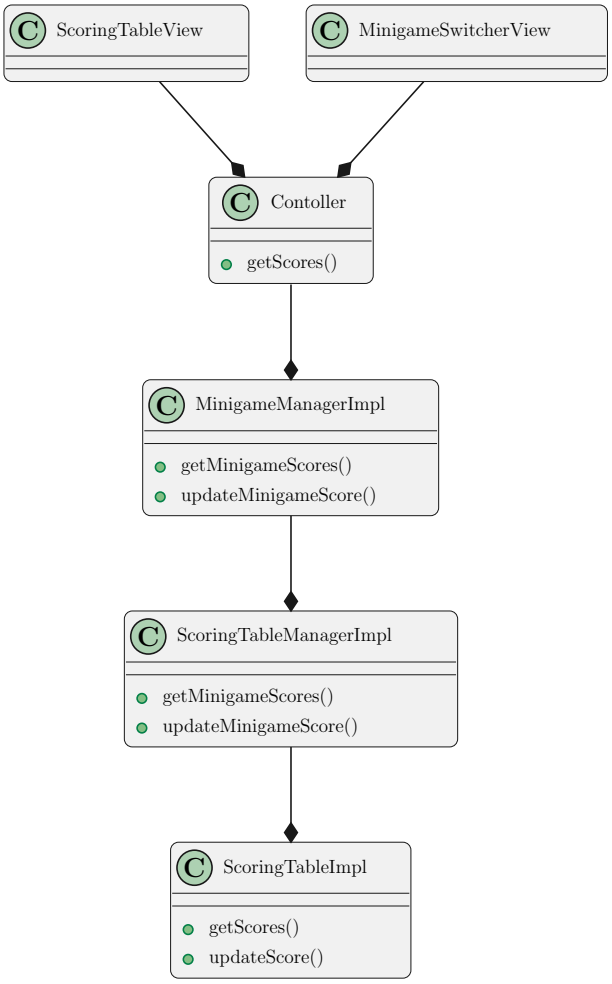

Figura 2.12: Schema UML della ScoreBoard

2.5 Design dettagliato - Mattia Morri

2.5.1 View di Bank Game

La vista del mio gioco è stata progettata pensando ad un'architettura modulare che suddividesse i vari componenti visivi. Ogni classe rappresenta un componente specifico della vista (eccetto i bottoni). Successivamente nella vista principale del mio gioco sono state create istanze delle classi, il che permette una gestione centralizzata. Ho deciso di utilizzare questo approccio per motivi di modularità, manutenibilità e chiarezza nel codice.

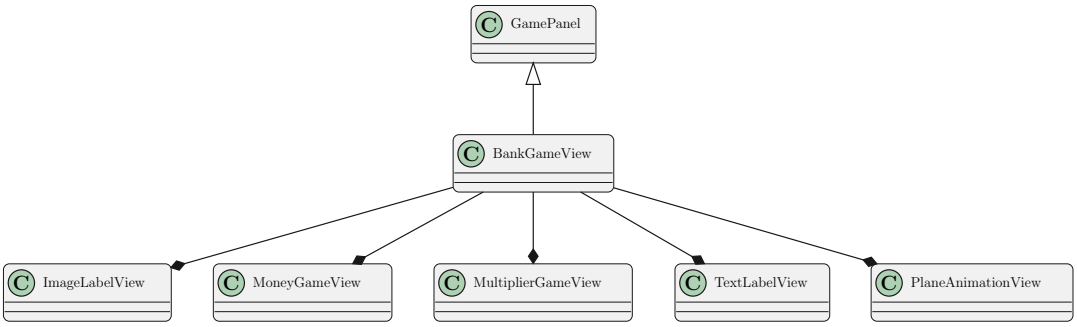

Figura 2.13: UML vista *BankGame*

2.5.2 Utilizzo thread

Descrizione del problema.
Quando il giocatore preme il tasto play viene avviato il moltiplicatore, ma se il *Main Thread* esegue un metodo del *Model* con un ciclo *while* all'interno, non può fare altro, quindi azioni come mostrare alla vista il moltiplicatore che prosegue e ascoltare gli input dell'utente, non possono essere eseguite contemporaneamente e quindi e ci si ritroverà con una schermata bloccata.
Soluzione.
Perciò, essendo un argomento che mi aveva particolarmente interessato a lezione e per sfida personale, ho deciso di risolvere questo problema tramite la creazione di due *thread*. In questo modo, infatti, una volta che il giocatore preme il tasto "play", si avrà:

- Un *thread* responsabile della gestione dell'interfaccia utente e degli eventi dell'utente (clic sui bottoni).
- Un *thread* che esegue la logica del moltiplicatore, aggiornandolo e segnalando al secondo *thread* creato quando può partire.

- Un ultimo *thread* che aggiorna l'interfaccia utente con i nuovi valori del moltiplicatore, dei soldi moltiplicati e segnala al primo *thread* quando può ripartire.

2.5.3 Sincronizzazione thread

Descrizione del problema
Personalmente la parte che mi è risultata più complessa è stata la sincronizzazione dei due *thread* nel mini-gioco. Come ho spiegato nelle righe precedenti, avendo creato 2 *thread* che operano su due variabili (un *thread* incrementa queste variabili mentre l'altro va a leggerle e stamparle), il problema era che senza una corretta sincronizzazione il tutto avveniva in maniera imprecisa, facendo di fatto "perdere" alcuni aggiornamenti di variabile al *thread*, che non arrivava in tempo a leggere la variabile, e quindi causando *race condition*.

Soluzione Per risolvere questo problema ho deciso di utilizzare il *monitor pattern*, visto a lezione con il professore Alessandro Ricci. Ho, quindi, creato una classe *"monitor"* con i metodi per sincronizzare i *thread* e istanziato due oggetti (*sync1* e *sync2*) nel *Controller* per poter richiamare i metodi nella *View*. Ho deciso di utilizzare questo pattern principalmente perché dedicare una singola classe alla logica della sincronizzazione trovo che renda il codice più pulito, semplice da comprendere e da apportare modifiche. In Figura 2.14 ho aggiunto un UML esplicativo.

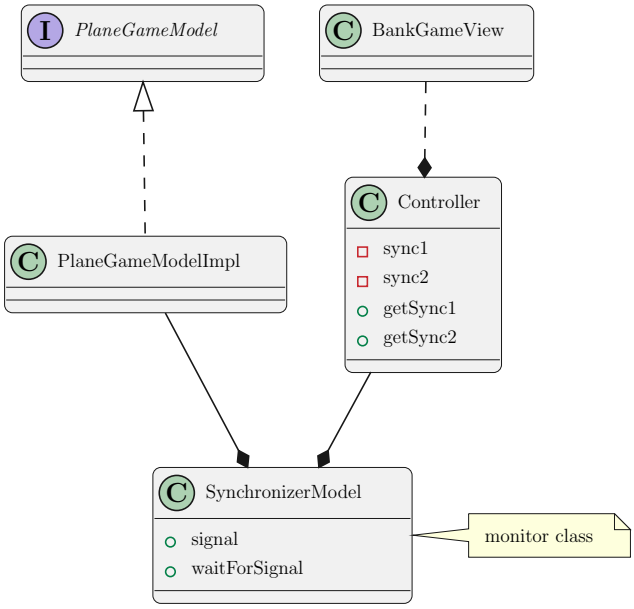

Figura 2.14: UML Model

Capitolo 3

Sviluppo

3.1 Testing Automatizzato

Durante la fase di sviluppo del software ogni qual volta si creava una nuova classe, la si accompagnava con dei test per controllare il suo corretto funzionamento, sia interno che esterno. Infatti molte delle classi sono composte da altre sottoclassi, quindi utilizzare il testing automatizzato per verificare la corretta composizione dell'insieme di classi era assolutamente necessario. Questo implica che le uniche classi per le quali abbiamo creato test sono classi del Model, in quanto è la logica di base del progetto, ed è stata la prima parte sviluppata. I test però sono anche stati utili in fasi successive quando si modificavano alcune classi del Model, i test scritti in precedenza mettevano in luce possibili problemi che tale modifica avrebbe generato. Per creare le classi di testing si è fatto uso della suite JUnit.

3.2 Note di sviluppo - Andrea De Carli

- Utilizzo di Lambda expression - Permalink: `https://github.com/Lucaaa31/OOP23-gym-life/blob/6cd2d428b9f84794ac0855ec5ca3f3da03d074ca/src/main/java/gymlife/model/statistics/StatsManagerImpl.java#L69`
- Utilizzo di Optional come oggetto di ritorno di un metodo - Permalink: `https://github.com/Lucaaa31/OOP23-gym-life/blob/c24892d01f8b9b064736166c6cb12e3413f8407e/src/main/java/gymlife/model/map/CellImpl.java#L283`

- Utilizzo di stream - Permalink: https://github.com/Lucaaa31/OOP23-gym-life/blob/0128ca792fa2949e104ec0d09be6001909e96391/src/main/java/gymlife/view/map/FastTravelView.java#L98
- Creazione di interfaccia funzionale personalizzata - Permalink: https://github.com/Lucaaa31/OOP23-gym-life/blob/0128ca792fa2949e104ec0d09be600190
src/main/java/gymlife/model/map/api/GameInteraction.java#L6
- Utilizzo di thread - Permalink: https://github.com/Lucaaa31/OOP23-gym-life/blame/0128ca792fa2949e104ec0d09be6001909e96391/src/main/java/gymlife/view/map/SleepView.java#L55

3.3 Note di sviluppo - Lucio Baiocchi

- Utilizzo di Optional - Permalink: https://github.com/Lucaaa31/OOP23-gym-life/blob/6cd2d428b9f84794ac0855ec5ca3f3da03d074ca/src/main/java/gymlife/model/encounter/EncountersFactory.java#L21
- Utilizzo di Stream e Lambda expression- Permalink: https://github.com/Lucaaa31/OOP23-gym-life/blob/6cd2d428b9f84794ac0855ec5ca3f3da03d074ca/src/main/java/gymlife/model/statistics/StatsManagerImpl.java#L69
- Classe per caricamento dei Font - Permalink: https://github.com/Lucaaa31/OOP23-gym-life/blob/6cd2d428b9f84794ac0855ec5ca3f3da03d074ca/src/main/java/gymlife/utility/FontLoader.java#L1
- Utilizzo di lambda expression - Permalink : https://github.com/Lucaaa31/OOP23-gym-life/blob/a14e3fbd346a5356044d7cf8bb486e78d3eda628/src/main/java/gymlife/model/character/CharacterImpl.java#L33
- Utilizzo di Stream - Permalink : https://github.com/Lucaaa31/OOP23-gym-life/blob/6cd2d428b9f84794ac0855ec5ca3f3da03d074ca/src/main/java/gymlife/model/statistics/StatsManagerImpl.java#L157

3.4 Note di sviluppo - Luca Camillini

- Utilizzo Consumer - Permalink: https://github.com/Lucaaa31/OOP23-gym-life/blob/a14e3fbd346a5356044d7cf8bb486e78d3eda628/src/main/java/gymlife/model/minigame/Minigame.java#L22
- Utilizzo Stream - Permalink: https://github.com/Lucaaa31/OOP23-gym-life/blob/a14e3fbd346a5356044d7cf8bb486e78d3eda628/src/main/java/gymlife/model/minigame/ScoringTableImpl.java#L48
- Utilizzo di Thread - Permalink: https://github.com/Lucaaa31/OOP23-gym-life/blob/a14e3fbd346a5356044d7cf8bb486e78d3eda628/src/main/java/gymlife/view/minigame/BenchView.java#L66
- Utilizzo Lambda - Permalink: https://github.com/Lucaaa31/OOP23-gym-life/blob/a14e3fbd346a5356044d7cf8bb486e78d3eda628/src/main/java/gymlife/view/minigame/LatView.java#L122
- Utilizzo Reflaction - Permalink: https://github.com/Lucaaa31/OOP23-gym-life/blob/a14e3fbd346a5356044d7cf8bb486e78d3eda628/src/main/java/gymlife/model/minigame/MinigameManagerImpl.java#L53

3.5 Note di sviluppo - Mattia Morri

- Utilizzo di Lambda expression e thread - Permalink https://github.com/Lucaaa31/OOP23-gym-life/blob/6cd2d428b9f84794ac0855ec5ca3f3da03d074ca/src/main/java/gymlife/view/bankgame/BankGameView.java#L191

Capitolo 4

Commenti Finali

4.1 Autovalutazione - Andrea De Carli

Della parte che ho sviluppato io trovo che il maggior punto di forza sia L'estensibilità, infatti grazie al vasto utilizzo di Enum, si possono creare nuove mappe, celle ed interazioni con l'aggiunta di pochissime righe. Questo solo a livello logico però, dato che ogni nuova cella avrebbe poi bisogno di una nuova immagine a livello di view. Punti deboli invece si trovano nella view, più nello specifico nella classe MainView. Infatti quest'ultima si occupa di cambiare i pannelli che vengono mostrati a schermo, lo fa utilizzando l'evento "perdita di focus" di ogni pannello. Questo implica però che ogni qualvolta si usano elementi di view che fanno perdere il focus al pannello corrente, la MainView erroneamente cambia pannello.

4.2 Autovalutazione - Lucio Baiocchi

Nel complesso credo che la parte che ci ha richiesto più tempo sia stata la progettazione iniziale. Il motivo principale è che nessun membro del gruppo avesse mai realizzato un progetto di questo tipo in team.
Superato l'ostacolo iniziale della progettazione il resto del processo è stato piuttosto lineare e semplice, grazie al fatto che sin dall'inizio ci siamo suddivisi i compiti per permettere lo sviluppo in parallelo del progetto, senza dover aspettare più di troppo il progresso degli altri compagni. Penso che tutti i membri del gruppo abbiano dato un buon contributo, e sono molto soddisfatto di come il gioco rispecchi pienamente le nostre idee iniziali. Ho anche notato un miglioramento progressivo delle nostre abilità, sia nel lato di programmazione che nell'organizzazione del team. Per quanto riguarda la mia parte, penso di aver dato un buon contributo, anche se credo che la

mia parte di lavoro fosse un po' meno complessa logicamente rispetto ad altre.

4.3 Autovalutazione - Luca Camillini

Sicuramente la parte di progettazione iniziale è stata fra le più impegnative perché, avendo scelto di sviluppare un videogioco "inedito" e non un clone, abbiamo dovuto pensare interamente alla progettazione e soprattutto alle interazioni tra i nostri domini. Sono molto contento di come sia venuto il progetto perché siamo riusciti a realizzare un'implementazione scalabile che ci permette di aggiungere e rimuovere feature a nostro piacimento.

4.4 Autovalutazione - Mattia Morri

Sono molto soddisfatto del gioco creato. Come lato negativo ritengo di aver fatto meno parte logica rispetto agli altri membri del gruppo, ma che nel suo complesso sia un buon contributo. Per quanto riguarda la view ritengo di aver fatto un buon lavoro considerando la partenza da zero su certi concetti. Sono molto soddisfatto, invece, del miglioramento personale fatto durante questo percorso e delle nozioni imparate, e ho visto un grande miglioramento anche negli altri componenti del gruppo. Ho voluto utilizzare i thread essendo stato un argomento che mi è interessato molto in Sistemi Operativi, e avendoli trattati anche con il professore Alessandro Ricci ho pensato che sarebbe stato stimolante lavorarci.

4.5 Difficoltà

Le maggiori difficoltà durante lo sviluppo del software si sono riscontrate nella prima fase di progettazione. Abbiamo infatti passato molto tempo a pensare alla migliore maniera per organizzare le diverse classi in una struttura che potesse rispecchiare la nostra visione, e fare ciò in maniera efficace (con riuso di codice e poche ripetizioni). Il fatto poi di trovarsi in gruppo significava anche trovare soluzioni che utilizzavano classi compatibili con quelle degli altri membri del gruppo. Una volta trovata una buona fondamenta per il progetto il resto è stato abbastanza semplice.

Appendice A

Guida Utente

A.1 Istruzioni Generali

A.1.1 Movimento

- W per muoversi su
- A per muoversi a sinistra
- D per muoversi a destra
- S per muoversi in basso

A.1.2 Generali

Dopo aver scelto la difficoltà si può uscire dal gioco premendo ESC, mentre se ci si posiziona sopra una cella con interazione (marcata da una icona colorata esemplificativa dell'azione che compie) si può premere E per interagire. Nella mappa della città ci si muove selezionando lo scenario nella barra in basso. E per ridimensionare l'interfaccia si possono usare i pulsanti '+' e '-'.

A.2 Istruzioni mini-giochi palestra

In tutti i tre i mini-giochi all'inizio verrà chiesto di selezionare una difficoltà. Dopo essere entrati nel mini-gioco premere un pulsante per iniziare.

- Panca piana → Premere il tasto più velocemente possibile per completare una ripetizione.

- Lat machine → Aspettare l'animazione della sequenza dei tasti che si illumineranno e poi ripeterli a memoria.
- Squat → Cliccare il tasto corrispondente al colore che ti verrà detto.

A.3 Istruzioni Minigioco banca

Una volta entrato nel mini-gioco si avrà a disposizione un box bianco dove all'interno si può selezionare la somma che si vuole giocare (l'importo deve essere inferiore o uguale al numero di soldi che si possiedono). Una volta digitata la somma si preme sul tasto Invio della tastiera e apparirà l'aeroplano. Se si ha un ripensamento sulla somma inserita si può ancora modificare inserendone una diversa e cliccando sempre Invio per confermare. Una volta sicuri, premendo sul bottone "PLAY", si potrà far partire un moltiplicatore che partirà da 0.90 fino ad un massimo di 5.90. Esso andrà a moltiplicare la somma inserita dal giocatore ma, attenzione, quando meno ce lo si aspetta l'aereo potrebbe volare via e il moltiplicatore si fermerà, facendo perdere al giocatore i soldi inseriti, perciò sta a discapito del giocatore decidere quando premere il pulsante stop e incassare i soldi moltiplicati. Una volta fatto fermare il moltiplicatore o in caso di perdita, cliccando il pulsante "RESTART", si resetterà il gioco e, effettuando i passaggi sopra descritti, si potrà rigiocare. L'importo guadagnato in caso di vincita verrà calcolato per eccesso o difetto in base alla prima cifra dopo la virgola, quindi sopra il 5, compreso, per eccesso altrimenti per difetto. Se si vuole uscire dal gioco, si clicca nel box bianco e si digita la lettere 'q' (sia in maiuscolo che minuscolo).