BIG HUNT

Relazione progetto

Programmazione ad oggetti

A.A. 2018/2019

Consegna: 25/04/2019

Mattia Achilli

Yuri Bernardini

Giulio Dulja

Simone Romagnoli

Indice

Capitolo 1
Analisi

1.1 Requisiti

Il gruppo si pone come obiettivo quello di realizzare un gioco ispirato a Duck Hunt, un videogioco di genere sparatutto "light gun". Il giocatore controlla un mirino attraverso il puntatore del mouse cercando di colpire le anatre che appariranno all'orizzonte.

Requisiti funzionali:

- Creazione di diversi menù:
    - Menu principale di gioco.
    - Menù delle impostazioni di gioco.
- Creazione di entità e gestione dei relativi eventi:
    - Cane (movimento, animazioni varie, raccolta dei bersagli).
    - Anatre (movimento, scomparsa dallo schermo di gioco).
- Grafica minimale di gioco con animazioni e effetti sonori (volo delle anatre, caduta delle anatre dopo essere state colpite, animazioni del cane).
- Ci saranno diverse tipologie di anatre che differiranno dal colore, velocità di movimento e tempo di permanenza nello schermo.
- Sarà possibile per l'utente ricevere dei potenziamenti dopo l'uccisione di particolari anatre.
- Il giocatore avrà un punteggio, mostrato a video, dato dalle anatre uccise.
- Si avranno a disposizione dei caricatori, ognuno dei quali avrà un certo numero di munizioni.
- Durante il corso della partita sarà visibile il punteggio e il numero di proiettili rimanenti per ogni caricatore.

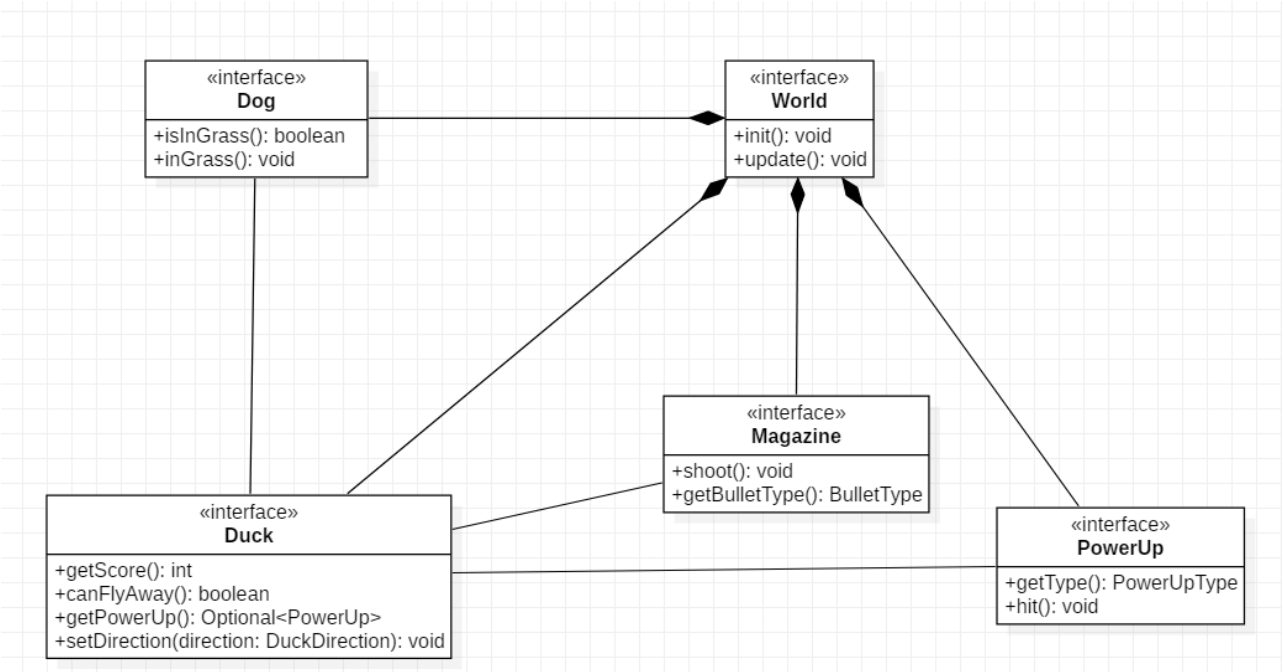
Requisiti non funzionali:

- Gestione di più round di gioco (superando i quali la difficoltà di gioco viene incrementata).

- Più modalità di gioco disponibili.
- Sarà aggiunto un ulteriore menù in cui saranno visualizzate le informazioni relative alle partite dell'utente.
- Salvataggio su file di informazioni di gioco, sfide, obiettivi e migliori punteggi.
- Possibilità di avere più account di gioco (salvataggio delle credenziali su file).

1.2 Analisi e modello del dominio

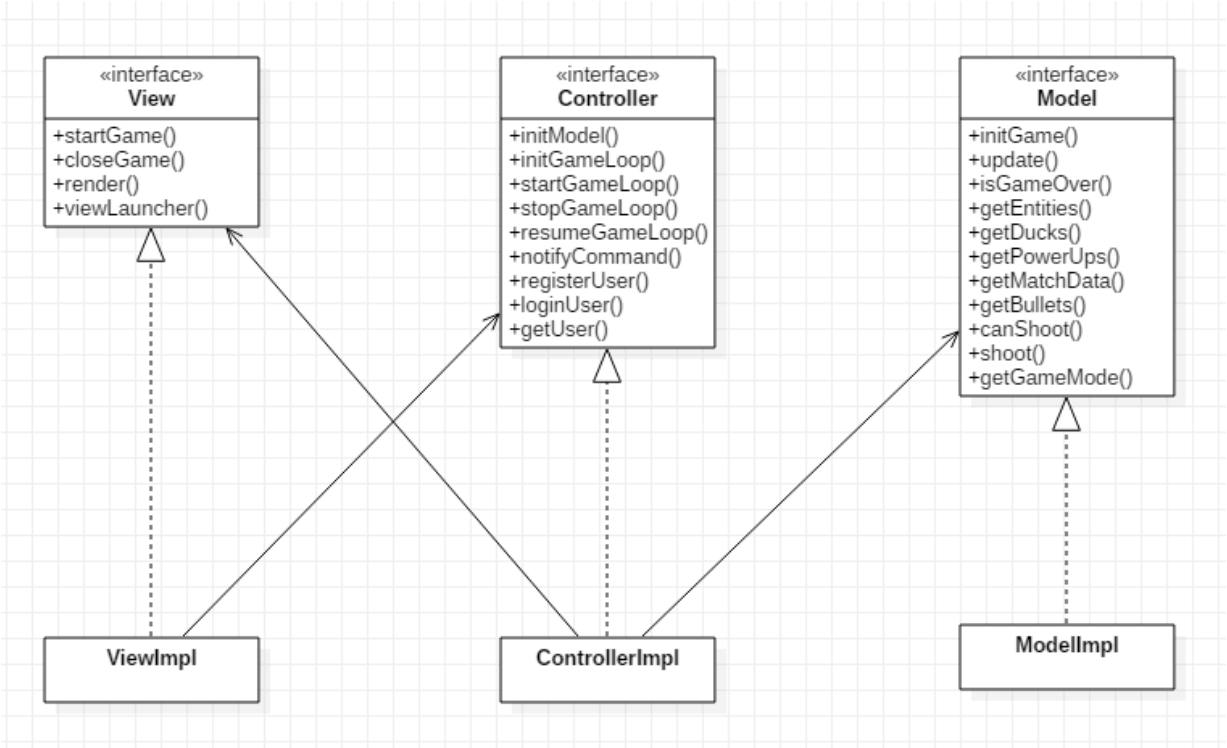
BigHunt è un gioco sparatutto di genere "light gun". Il mondo di gioco si presenta con un cane, il nostro fidato compagno che ci aiuterà nella nostra caccia. Il giocatore controlla un mirino e ha in suo possesso un numero finito o infinito di proiettili, a seconda della modalità di caccia, attraverso il quale colpisce le anatre che si presentano man mano nella schermata di gioco. Sparare al cane è considerato errore, infatti tale azione verrà punita con una penalizzazione del punteggio. Uccidendo le anatre si otterranno dei punteggi: esistono anatre di diverso tipo, facilmente distinguibili dai colori delle loro piume; colpendo le anatre più esperte e veloci, il giocatore otterrà un punteggio notevolmente più alto. Le anatre uccise possono rilasciare un potenziamento di vario tipo il quale, una volta colpito, darà dei vantaggi al giocatore. Il gioco termina una volta che il giocatore ha finito i proiettili o non ha raggiunto il minimo punteggio stabilito per un'ondata di anatre, oppure in caso di vittoria.



Capitolo 2

Design

2.1 Architettura



Per la realizzazione del gioco abbiamo utilizzato il pattern architetturale **MVC (model-view-controller)**. Il pattern ci consente di isolare le tre parti di Model, View e Controller rendendole indipendenti. Questo aspetto legato all'indipendenza tra le varie parti ci consente di modificare una di esse come la grafica senza compromettere le altre. La componente denominata Model è stata progettata in modo che fosse all'oscuro dell'esistenza di View e Controller, lasciando a quest'ultime la responsabilità di chiederle le informazioni necessarie periodicamente. La parte di View si occupa di prendere l'input dall'utente per poi notificarli al Controller ed infine saranno comunicati al Model il quale si aggiornerà costantemente considerando gli input. Il Controller mantiene la gestione dell'accesso ai dati persistenti (Users e

Podium) e ha come compito quello di aggiornare periodicamente il Model tramite un loop che viene creato ad ogni sessione di gioco assieme ad un'istanza ModelImpl: tale loop è rappresentato dalla classe interna all'implementazione del Controller, chiamata GameLoop. Infine, la View si occuperà di aggiornare tutta la grafica di gioco; anch'essa è dotata di un loop rappresentato da una inner-class, chiamato Render, che la aiuta nel suo ruolo.

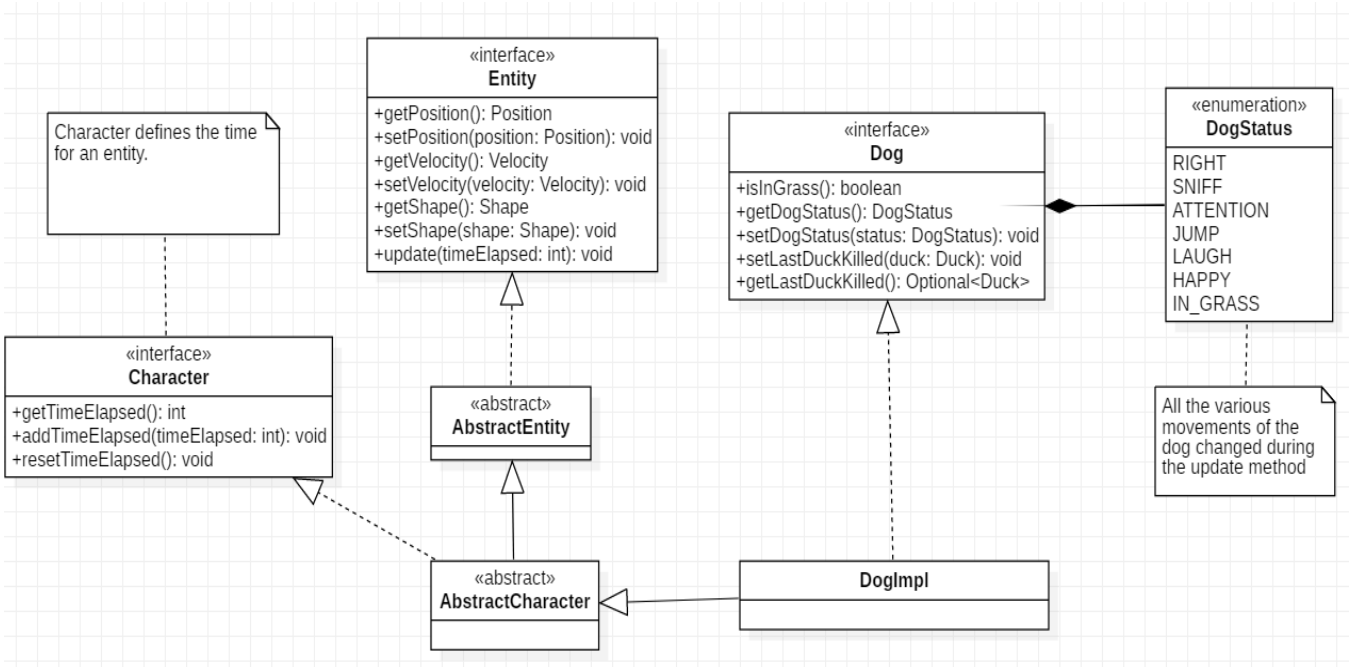
2.2 Design Dettagliato

Mattia Achilli

Nel progetto la mia parte è incentrata sulla creazione delle entità di gioco come le anatre (e la relativa generazione) e il cane, qui di seguito spiegherò in maniera dettagliata il design e gli eventuali pattern di questi.

Cane

Il cane è un'entità di gioco puramente visiva più che logica, infatti le uniche cose che il cane dovrà fare sono quelle di: muoversi, uscire fuori dall'erba quando le anatre scappano e mostrare le anatre colpite una volta cadute. Non ho utilizzato alcun pattern dato che non ne necessitavo però ho voluto comunque definire il cane come un'entità di modello, salvando i suoi diversi "stati" durante il movimento relativo alla posizione in una enum chiamata: "DogStatus".

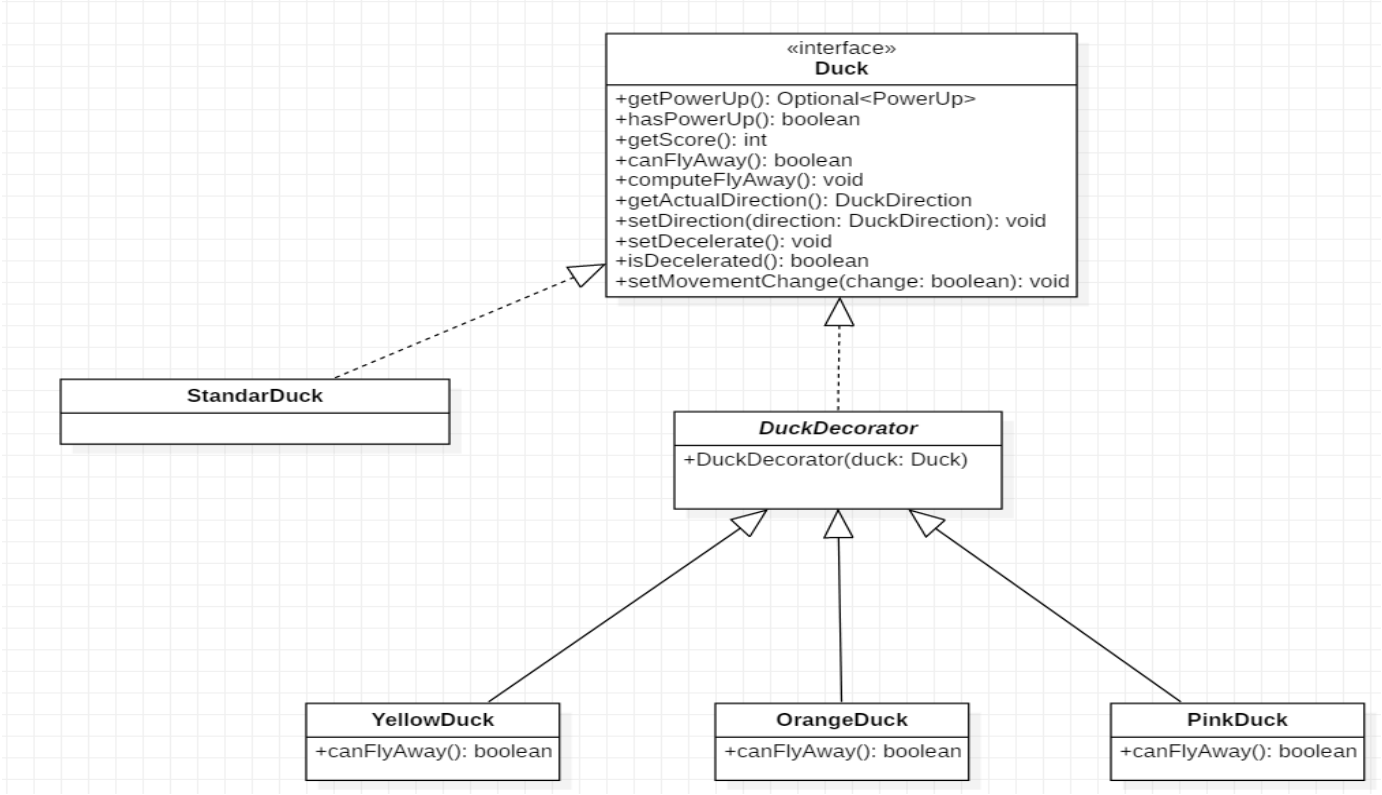


Ogni tot millisecondi il cane aggiorna il proprio movimento, ogni movimento corrisponderà nella View ad immagini differenti. Nonostante non abbia utilizzato pattern il cane come tutte le entità di gioco è facilmente estendibile.

Nota: Ho inserito nello schema solo le classi e le interfacce di maggiore interesse.

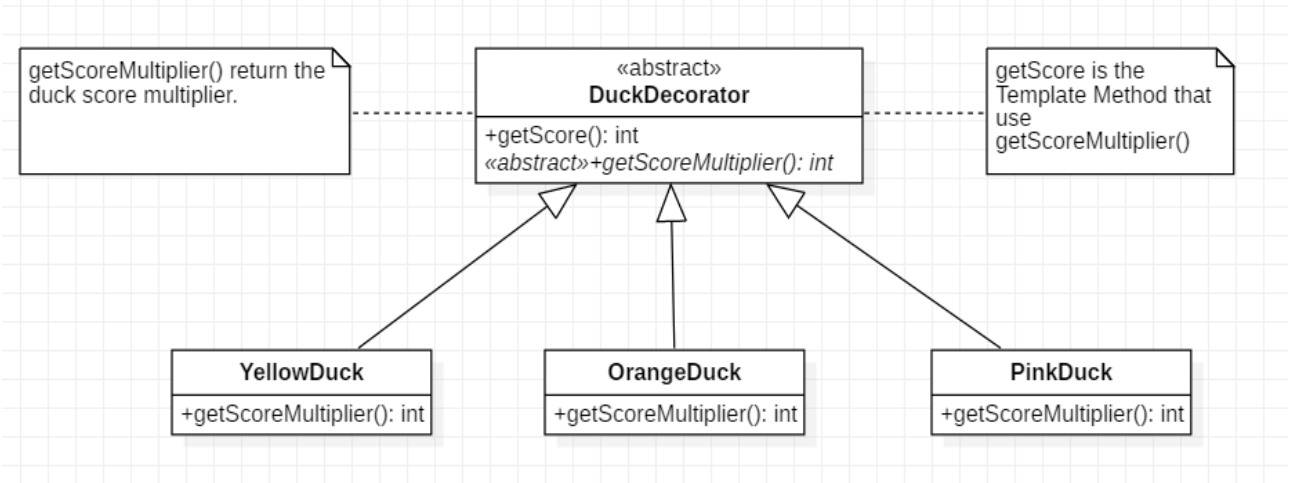
Anatre

Le anatre sono entità di gioco che compaiono durante la partita partendo leggermente al di fuori dello schermo in posizioni random, esse una volta comparse nello schermo volano cambiando la loro direzione ogni secondo. Le anatre hanno una velocità, un tempo di permanenza nello schermo (superato ciò voleranno via), e un punteggio ottenibile sparandogli, queste caratteristiche differiscono in base al colore dell'anatra: nero (o standard), giallo, arancione e rosa. Ho utilizzato il pattern **Decorator** per evitare di duplicare il codice e per possibili aggiunte future modellando il comportamento di ogni anatra a mio piacimento.

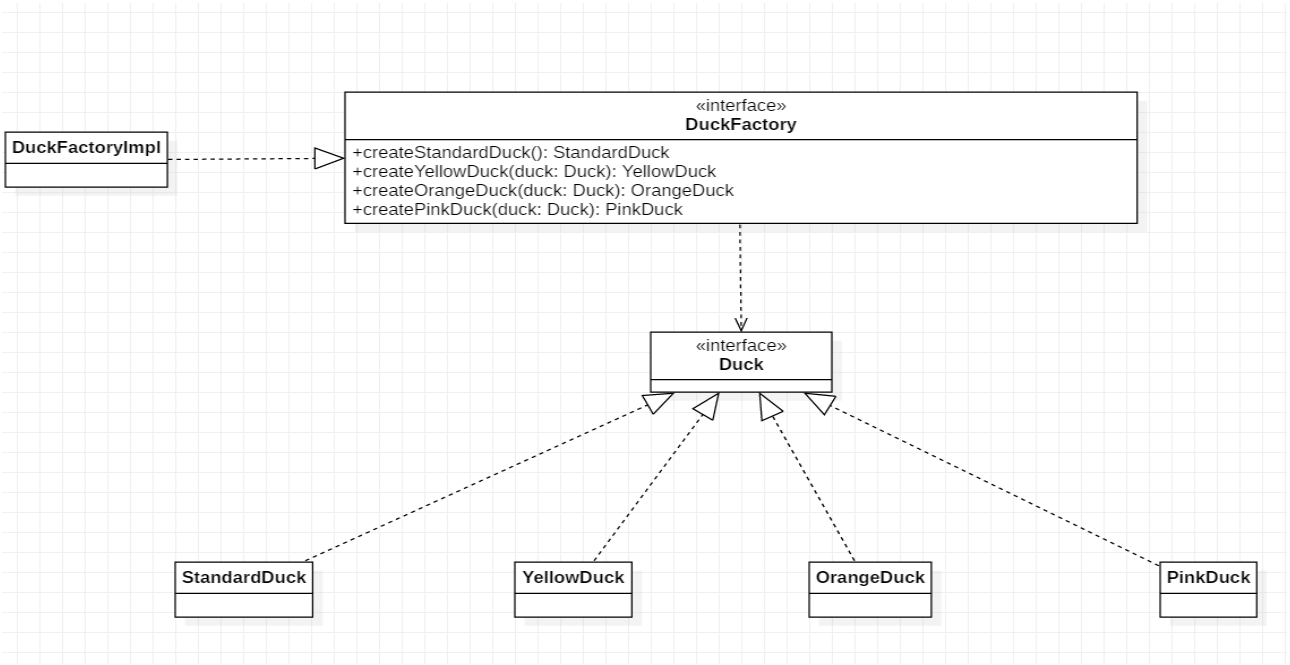


Nota: inserisco i metodi più importanti per spazio o solamente per mostrare il concetto principale dell'utilizzo del pattern.

Per la gestione relativa ai punteggi delle diverse anatre ho deciso di usare il pattern **Template Method** in quanto mi permette di non duplicare il codice e di ottenere moltiplicatori di punteggio diversi per tipo di anatra.

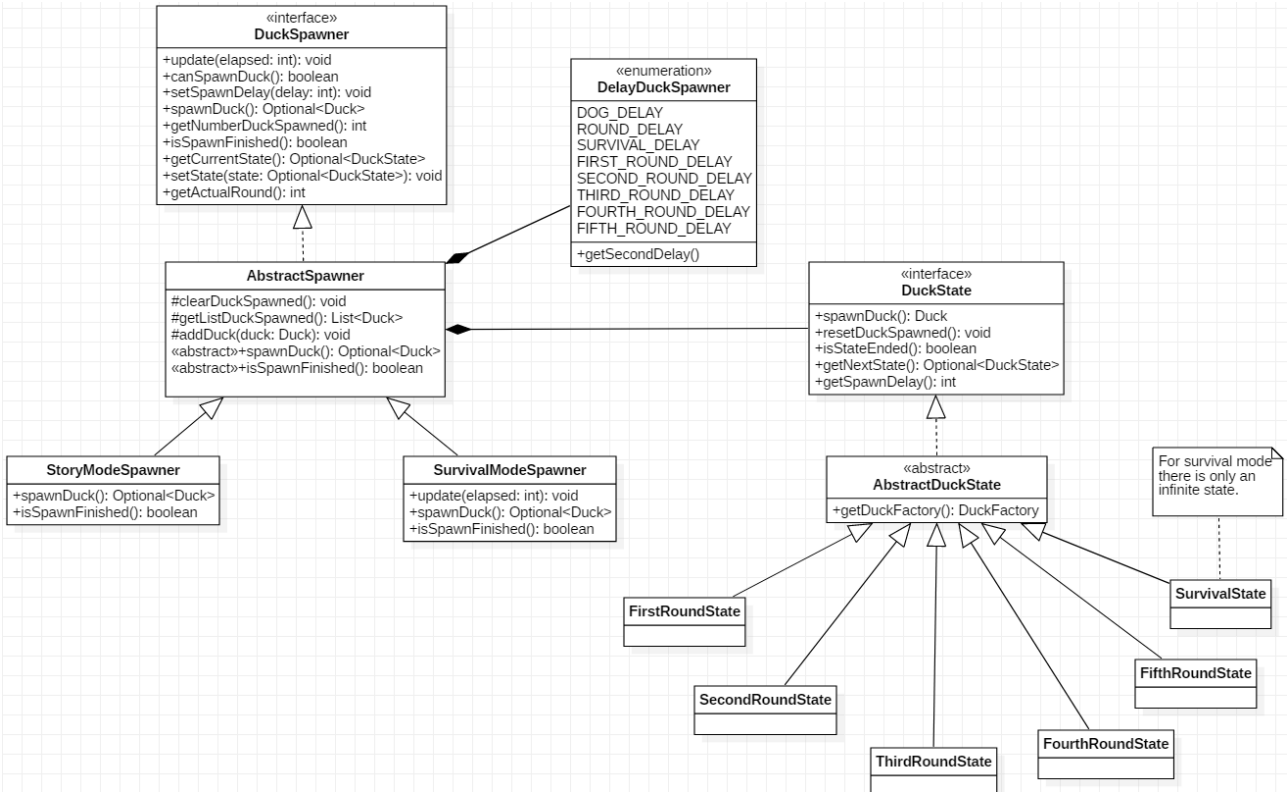


Per quanto riguarda la creazione delle anatre e quindi tutti i relativi tipi, ho utilizzato il pattern **Factory Method**, così da poterle creare facilmente in una sola classe.

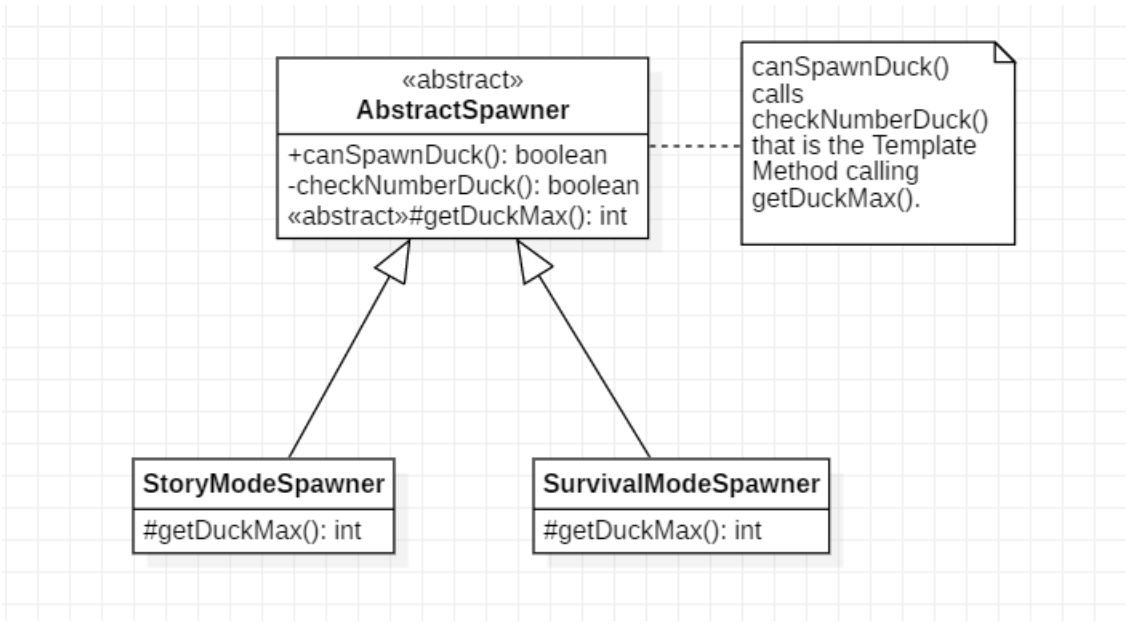


Generazione nelle modalità di gioco

Dato che il nostro gioco ha due modalità, una a round e l'altra a sopravvivenza ho voluto progettare le due diverse modalità in modo che siano facilmente estendibili (esempio: aggiungere nuovi round). Ho creato un oggetto principale denominato spawner il quale si suddivide in uno spawner per la storia e per la sopravvivenza, per entrambi gli spawner ho utilizzato il pattern **State** potendo così sviluppare diversi tipi di round e di complessità nella generazione delle anatre durante il corso del gioco. Per la modalità a round i vari stati dei round sono collegati utilizzando il pattern **Chain of Responsibility** e quindi finito un round si lascia la responsabilità del round successivo ad un altro oggetto. Per la modalità survival o sopravvivenza c'è un solo e infinito round con difficoltà incrementale proporzionale al tempo.



Con il fatto di avere due diverse modalità ho voluto diversificare un po' le cose cambiando il numero massimo di anatre nello schermo, anche qui ho utilizzato il pattern **Template Method** ottenendo il numero massimo di anatre a seconda della modalità.



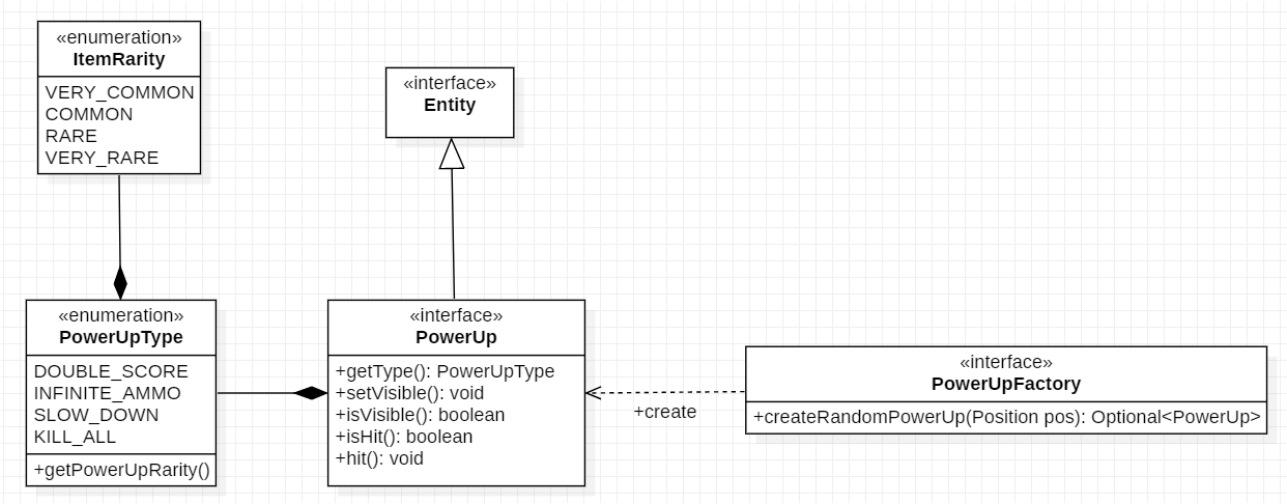
Yuri Bernardini

Il mio compito nel progetto è incentrata nella creazione e generazione dei potenziamenti di gioco, nelle impostazioni di gioco e nella creazione, disposizione, generazione e navigazione di tutte le schermate o scene di gioco.

Potenziamenti

I *PowerUp* sono i potenziamenti di gioco che compaiono dopo la morte di alcune anatre, se colpiti, quindi attivati danno diversi tipi di vantaggi, che durano per un certo intervallo di tempo.

Per la creazione di queste entità ho scelto di utilizzare il pattern **Factory Method** poiché è un pattern creazionale molto utile per la creazione di oggetti, in questo caso, di tipo PowerUp; inoltre così facendo si rende il sistema indipendente dal metodo di creazione e nascondiamo questa modalità. La generazione di questi potenziamenti viene fatta tramite un semplice algoritmo da me implementato.



L'interfaccia PowerUp estende l'interfaccia *Entity* così come altre entità di gioco. Si basa su una enumerazione PowerUpType dove vengono definite tutte le varie tipologie, questa enumerazione fa uso di una ulteriore enumerazione chiamata *Rarità.*

Rarità

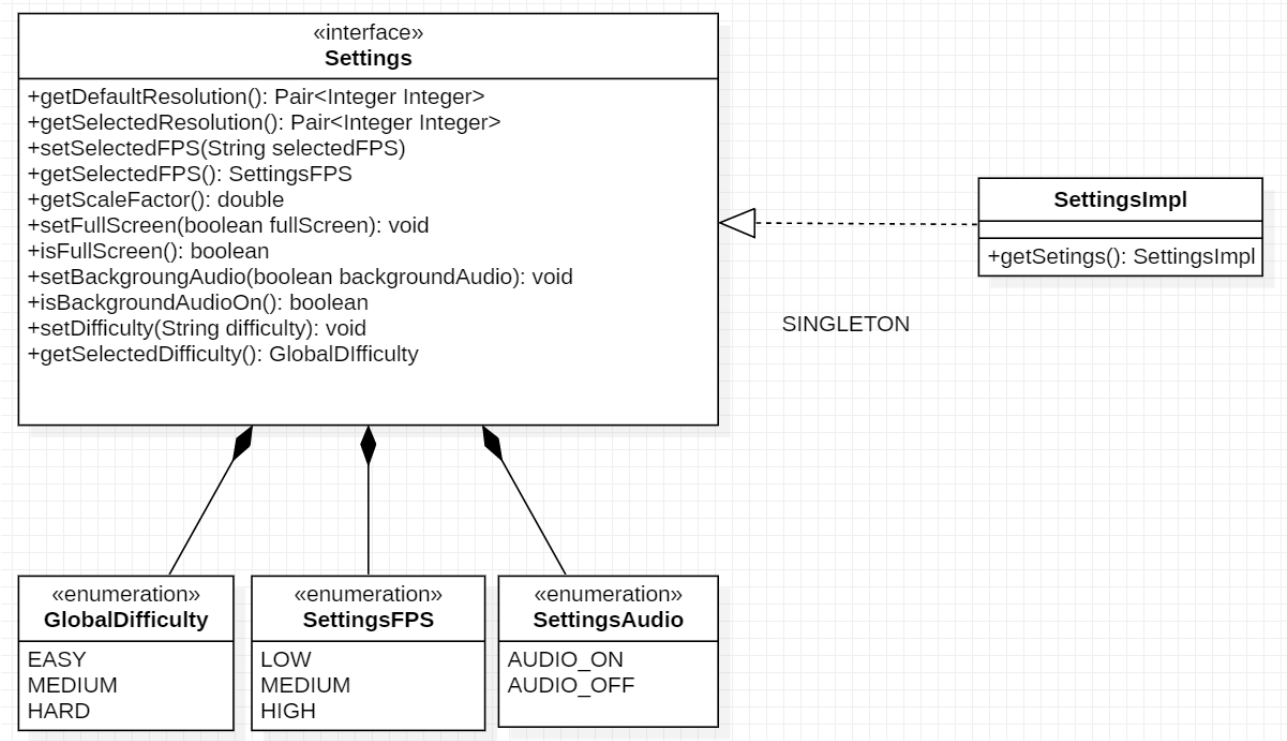
Ho inoltre implementato questa enum poiché ritenevo comoda da usare per la creazione di tipi PowerUp in base ai diversi gradi di rarità e la possibilità che sia usata anche da altre entità di gioco quali ad esempio le anatre che possono avere diversa rarità.

Impostazioni di Gioco

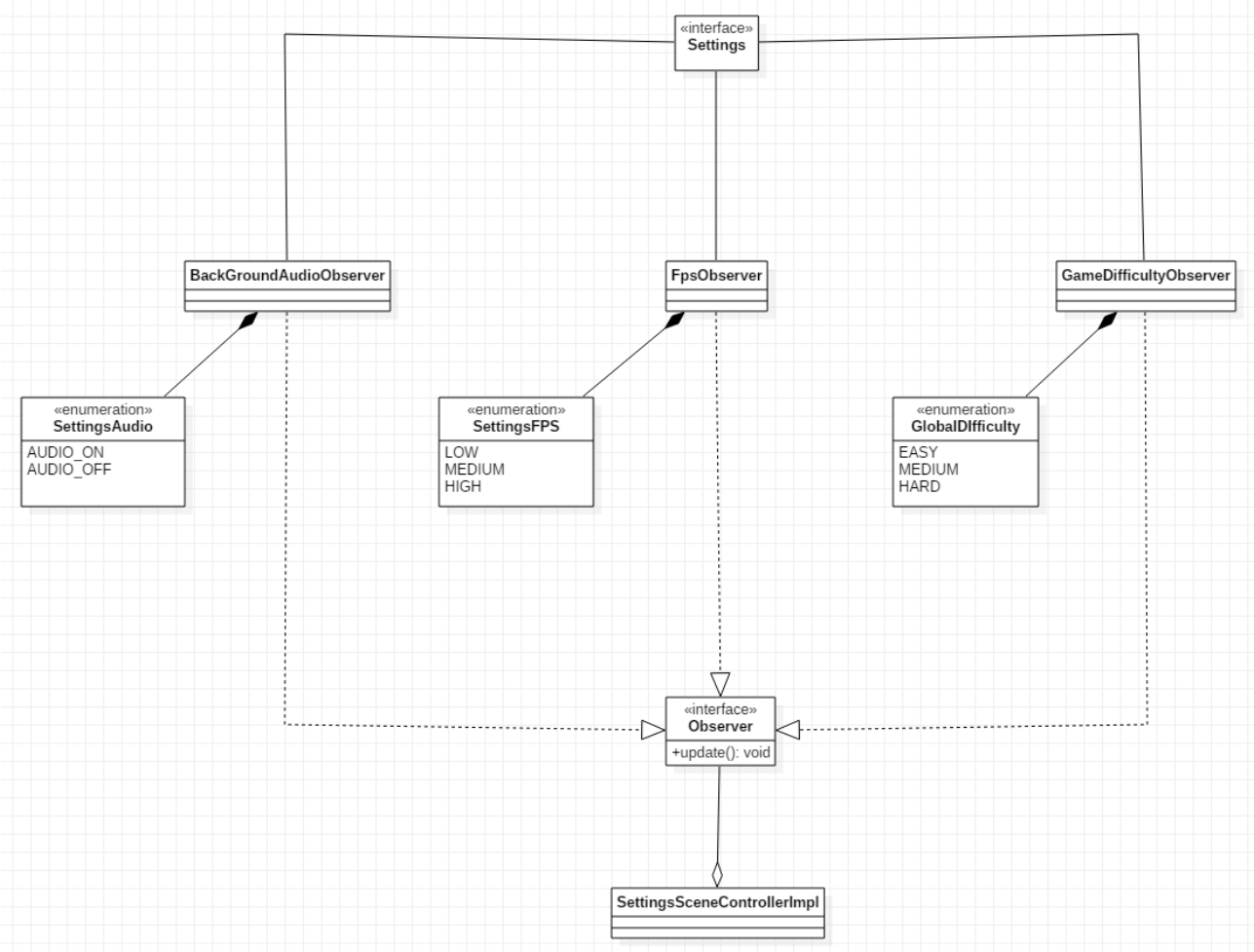
Altro aspetto di mia responsabilità sono le impostazioni di gioco, *Settings.*

In questa parte vengono messe a disposizione dell'utente una serie di opzioni per regolare alcune caratteristiche di gioco a proprio piacimento. Le varie impostazioni che si possono modificare sono la difficoltà, la modifica degli fps, la scelta di tenere attivo o meno la musica durante le partite.

La classe contenente le impostazioni *SettingsImpl* è stata creata usando il pattern **Singleton** poiché, anche se si tratta di un pattern da utilizzare con il massimo riguardo, lo ritenevo più adatto per avere una sola istanza di questa classe, così da non compromettere il funzionamento dell'applicazione ed essere facilmente reperibile dove richiesta.



Inoltre per gestire le varie modifiche apportate alle impostazioni durante l'esecuzione ho scelto il pattern **Observer** creando diverse gestioni per ogni tipologia di impostazione; ho così collegato una classe contenente il componente grafico, con metodi per generazione, selezione e riempimento (ad esempio utilizzanti checkbox, o combobox), alla sua equivalente ObserverClass. Così ogni volta che l'utente applica una scelta, le impostazioni di gioco vengono aggiornate.



View

Altro mio importante compito era implementare le classi di view del gioco.

Ho scelto una struttura che rendesse facile la generazione, e la navigazione delle scene di gioco in base alla scelta dell'utente.

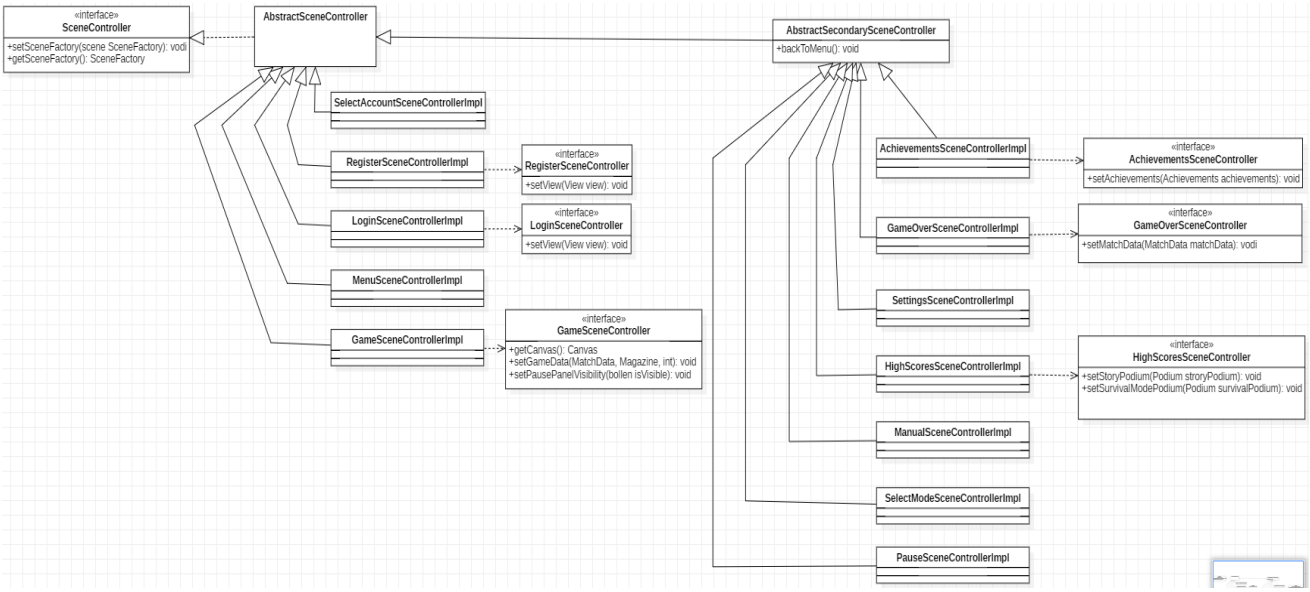
Ho creato le varie scene di gioco tramite l'aiuto di un programma **JavaFx** sceneBuilder ogni singola classe *fxml*: Achievements, Game, GameOver, HighScores, Login, Manual, Menu, Pause, Register, SelectAccount, SelectMode, Settings.

Ad ogni classe *fxml* ci ho collegato la relativa classe *Controller* che ne implementa le azioni e le caratteristiche per ottenere una completa indipendenza le une dalle altre.

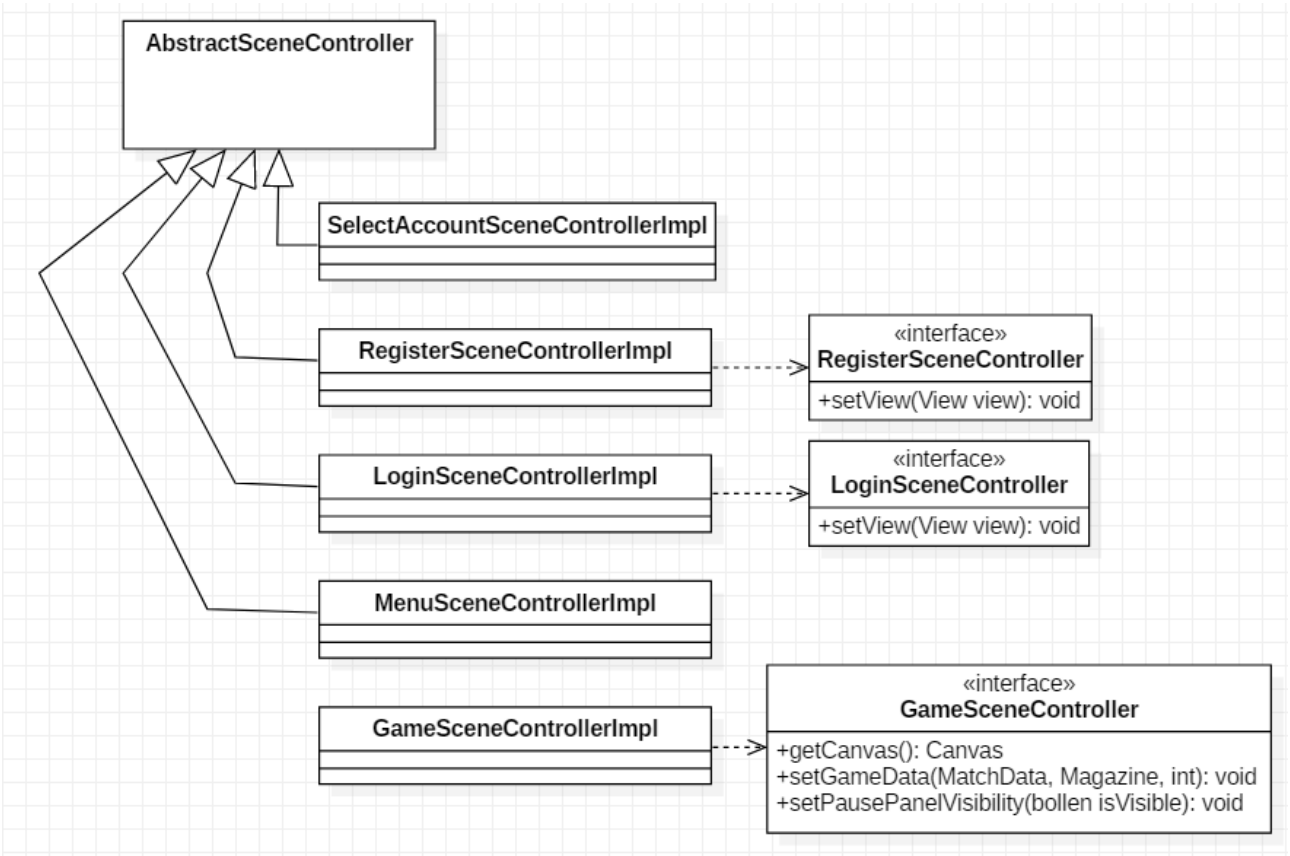
Se una Scena richiedeva particolari funzionalità o semplicemente di mostrare in output dei dati, ho fatto in modo che implementasse una sua specifica *interfaccia* oltre alla *classe astratta*.

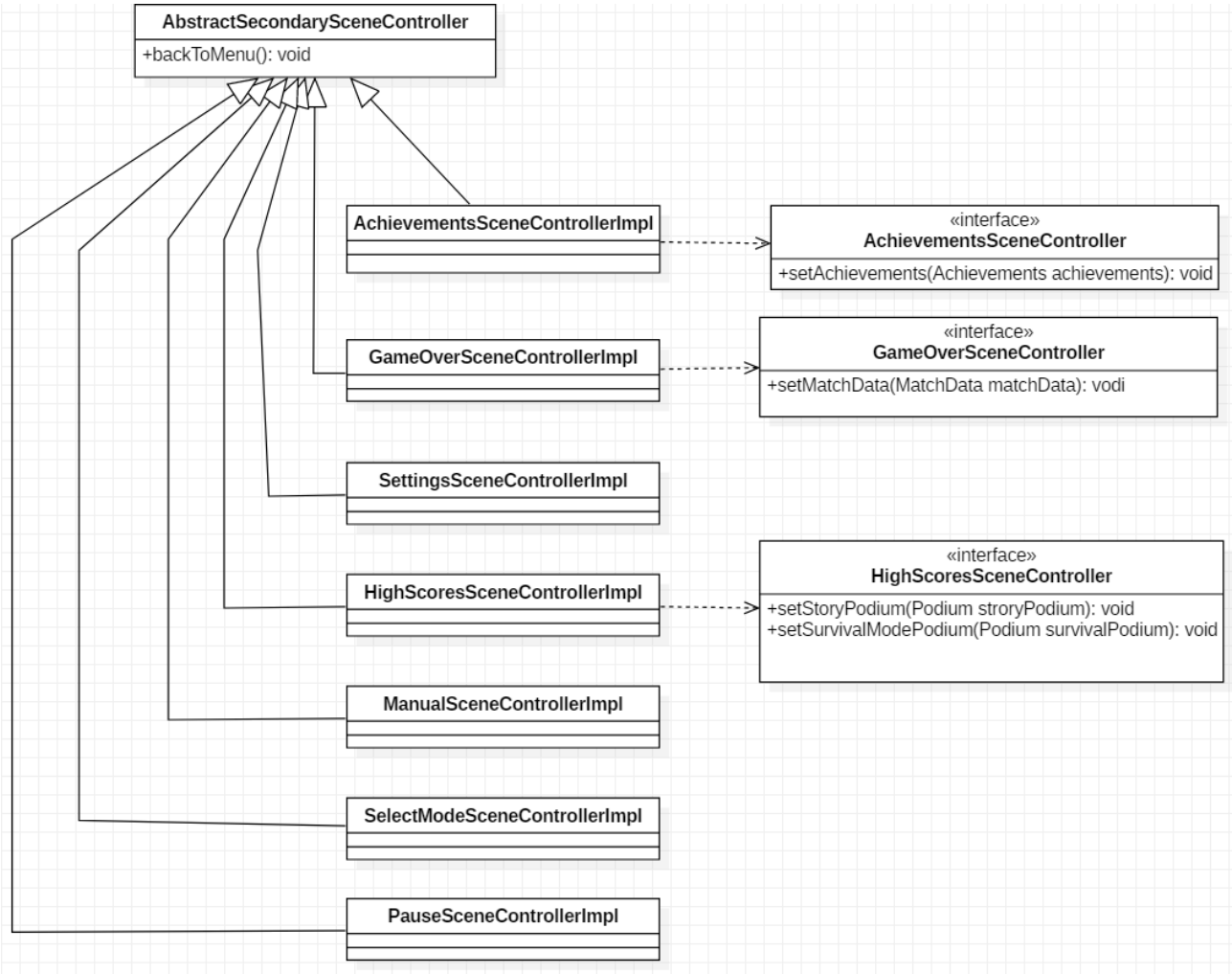
Ci sono due tipi di classi astratte associate alle varie scene del gioco: AbstractSceneController (che contiene i metodi per caricare la scena) e

AbstractSecondarySceneController (usata dalle pagine secondarie, che alla prima aggiunge un metodo per tornare indietro alla pagina di menu principale).



Di seguito mostro in modo più chiaro le due tipologie di classi astratte e quelle che ne ereditano le caratteristiche.

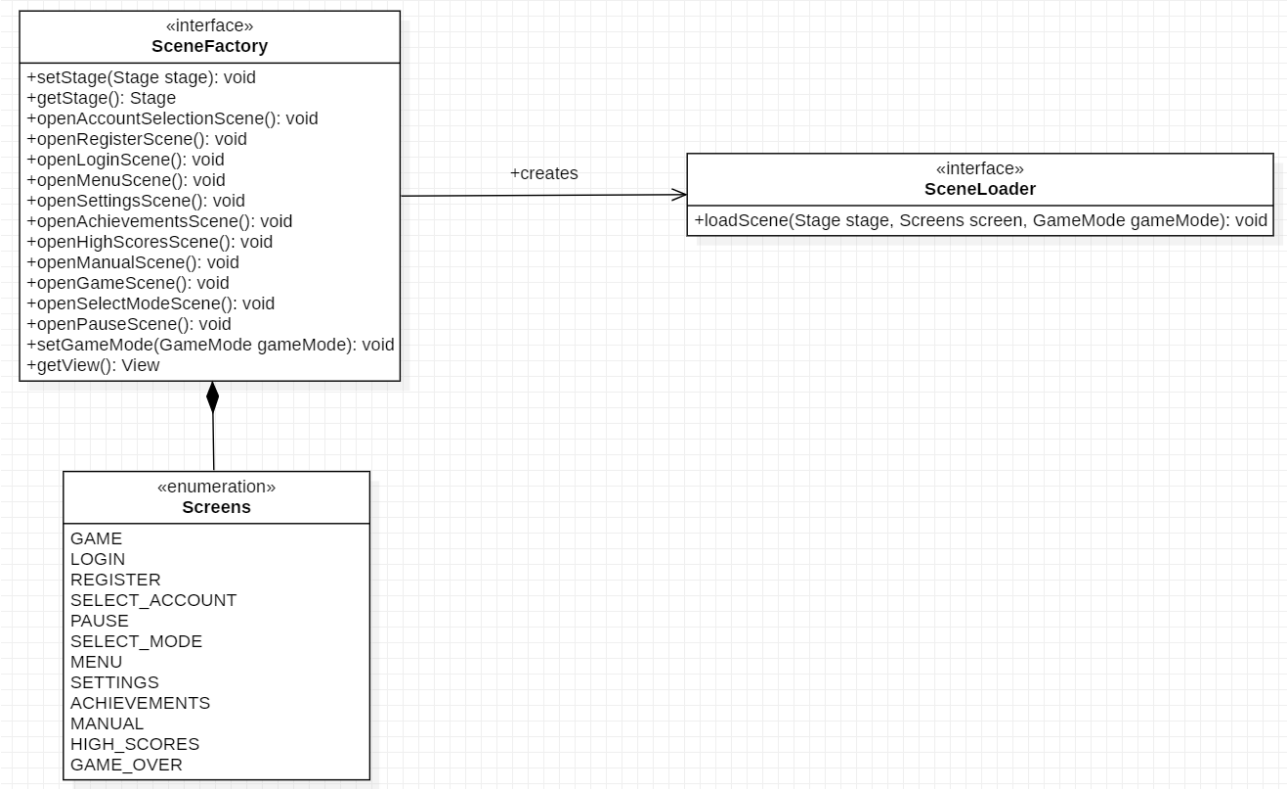




Per ogni file fxml ho deciso di implementare tramite un **foglio di stile Style.css** le parti principali del suo aspetto.

Per la metodologia di gestione e la navigazione tra le varie schermate di gioco ho scelto di usare due classi: SceneLoader con relativa interfaccia contenente i metodi principali per la gestione della pagina del gioco sia graficamente (aspetto finestra e informazioni) sia a livello di creazione di vari momenti di gioco in base alla scena richiesta, ad esempio cosa fare in caso di caricamento di Login, di Registrazione, di Gioco, di GameOver, di visualizzazione Achievements o di HighScores.

Mentre per la creazione delle varie pagine di gioco ho usato il pattern **Factory** con la classe SceneFactoryImpl collegata alla relativa interfaccia rende una facile introduzione di nuove schermate/scene del gioco in base alle integrazioni dell'utente.

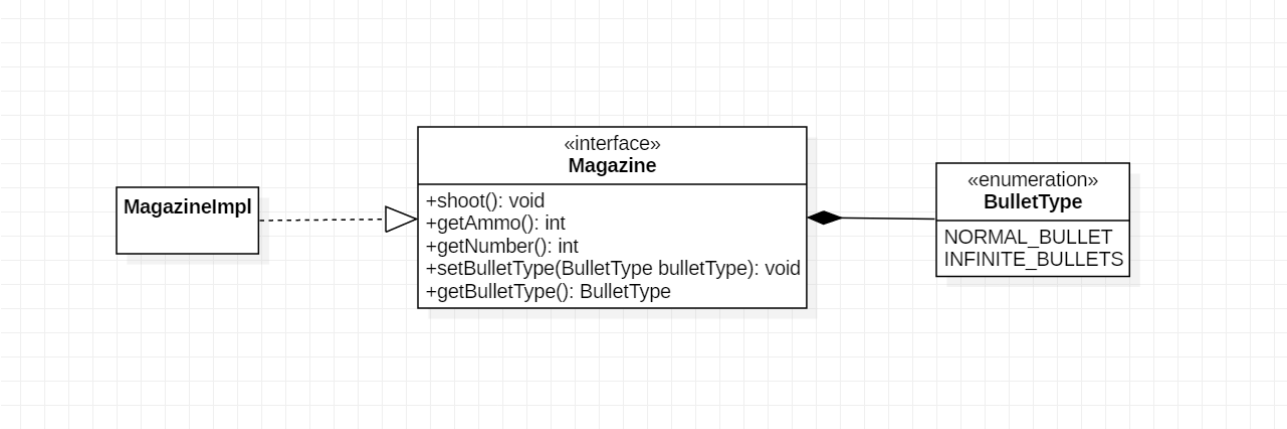

«interface»
SceneFactory
+setStage(Stage stage): void
+getStage(): Stage
+openAccountSelectionScene(): void
+openRegisterScene(): void
+openLoginScene(): void
+openMenuScene(): void
+openSettingsScene(): void
+openAchievementsScene(): void
+openHighScoresScene(): void
+openManualScene(): void
+openGameScene(): void
+openSelectModeScene(): void
+openPauseScene(): void
+setGameMode(GameMode gameMode): void
+getView(): View
+creates
«interface»
SceneLoader
+loadScene(Stage stage, Screens screen, GameMode gameMode): void
«enumeration»
Screens
GAME
LOGIN
REGISTER
SELECT_ACCOUNT
PAUSE
SELECT_MODE
MENU
SETTINGS
ACHIEVEMENTS
MANUAL
HIGH_SCORES
GAME_OVER

Giulio Dulja

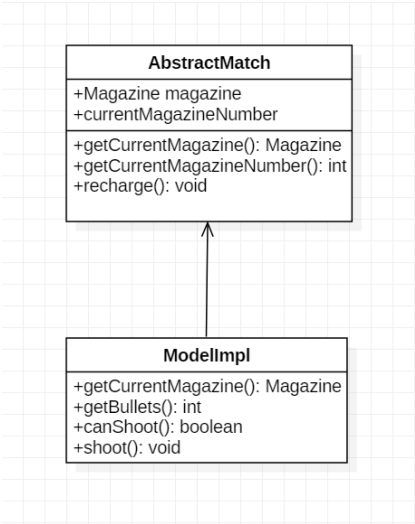
Il mio lavoro nel progetto riguarda vari aspetti del gioco, tra cui la gestione dell'input dell'utente, i caricatori con relative munizioni, l'audio del gioco e tutte le relative interazioni di questi con l'MVC.

Caricatori

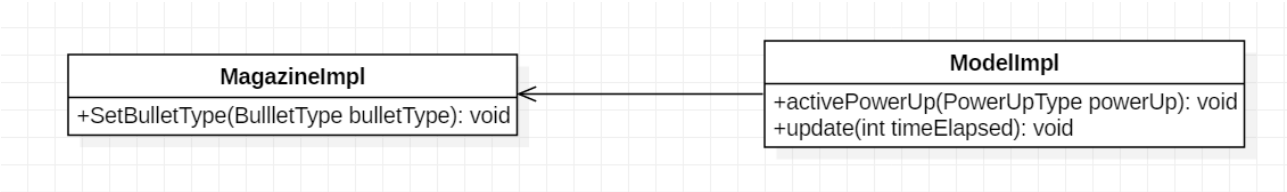
Abbiamo deciso di rendere la gestione delle munizioni nel gioco manuale, e per questo la gestione dei caricatori ha assunto una certa importanza. Essendo uno dei power-up all'interno del gioco quello delle munizioni infinite ho deciso di gestirlo grazie ad una enum "BulletType" che contiene i tipi di proiettile, che potrebbe inoltre lasciar spazio a maggiori personalizzazioni future.



I caricatori hanno un campo che tiene conto delle munizioni rimanenti e uno che indica il numero seriale del caricatore, per distinguerli l'uno dall'altro.



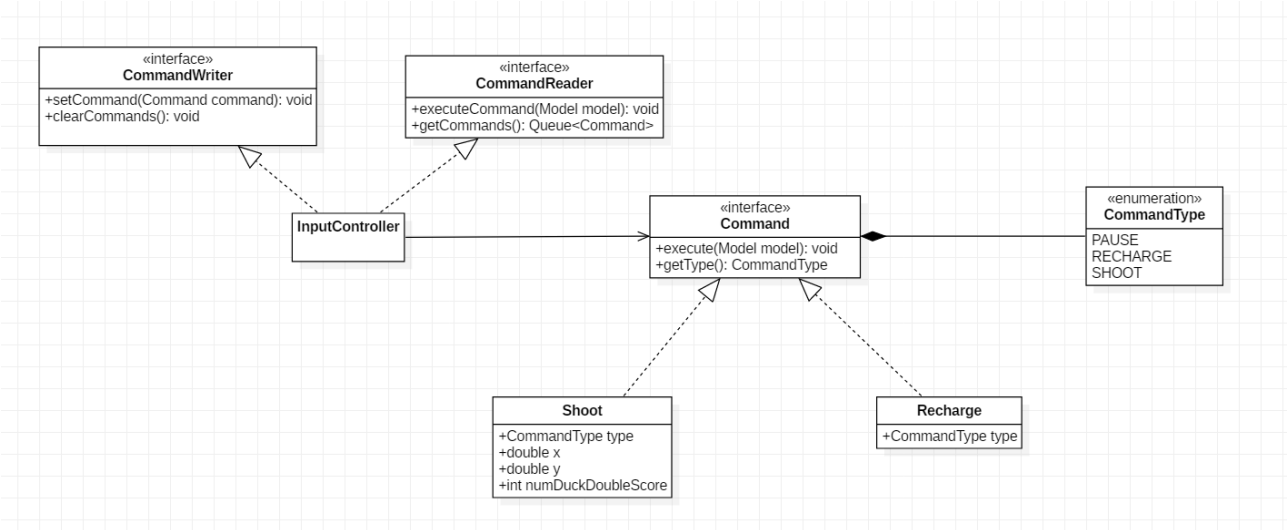
I caricatori sono memorizzati nella classe AbstractMatch e vengono lì gestiti, oltre che all'interno del Model. Gli aspetti cruciali sono la gestione del numero di munizioni e numero di caricatori. Uno dei modi in cui la partita si conclude è infatti per l'esaurimento di munizioni e caricatori.



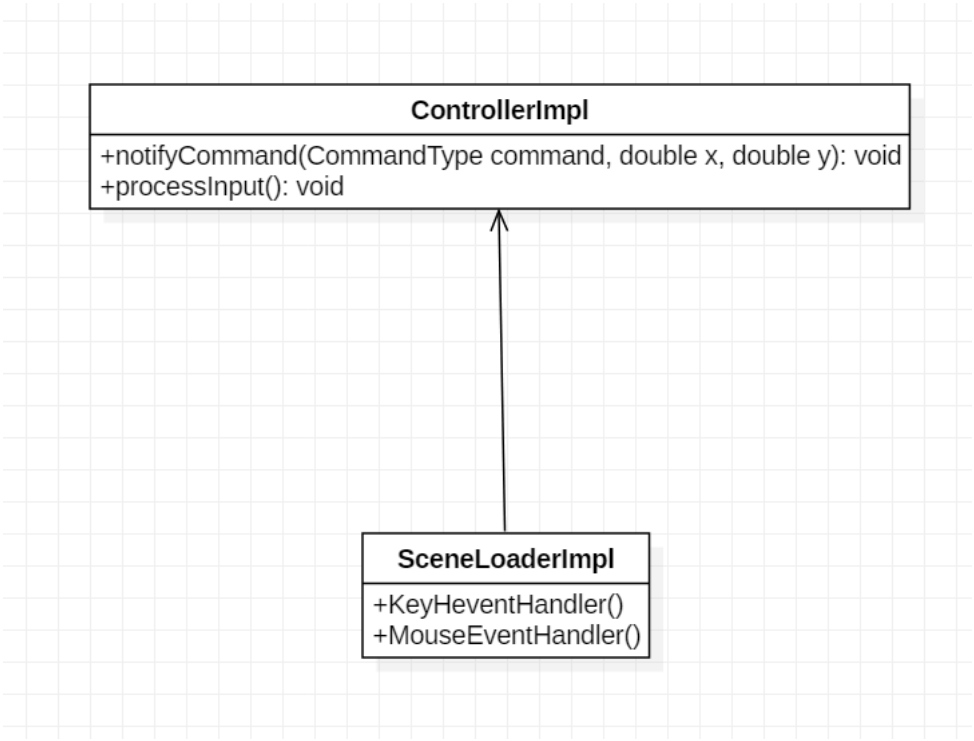
Il powerUp che garantisce munizioni infinite ha un'implementazione differente rispetto agli altri, viene infatti cambiato il tipo di proiettile. Avendo una durata massima, la sua disattivazione è gestita all'interno del metodo update del model, in cui viene controllato per quanto tempo è rimasto attivo il powerUp, e, quando necessario, lo disattiva come gli altri powerUp andando inoltre a cambiare nuovamente il tipo di proiettile a normale.

Gestione dell'input

Per la gestione dell'input dell'utente ho scelto di utilizzare il pattern **Command**. Ho quindi creato due interfacce, "CommandWriter" che comprende i metodi per la gestione dei comandi in entrata, e "CommandReader" per la gestione dei comandi in uscita. "InputController" implementa entrambe le interfacce sfruttando una coda come struttura dati per gestire i comandi.

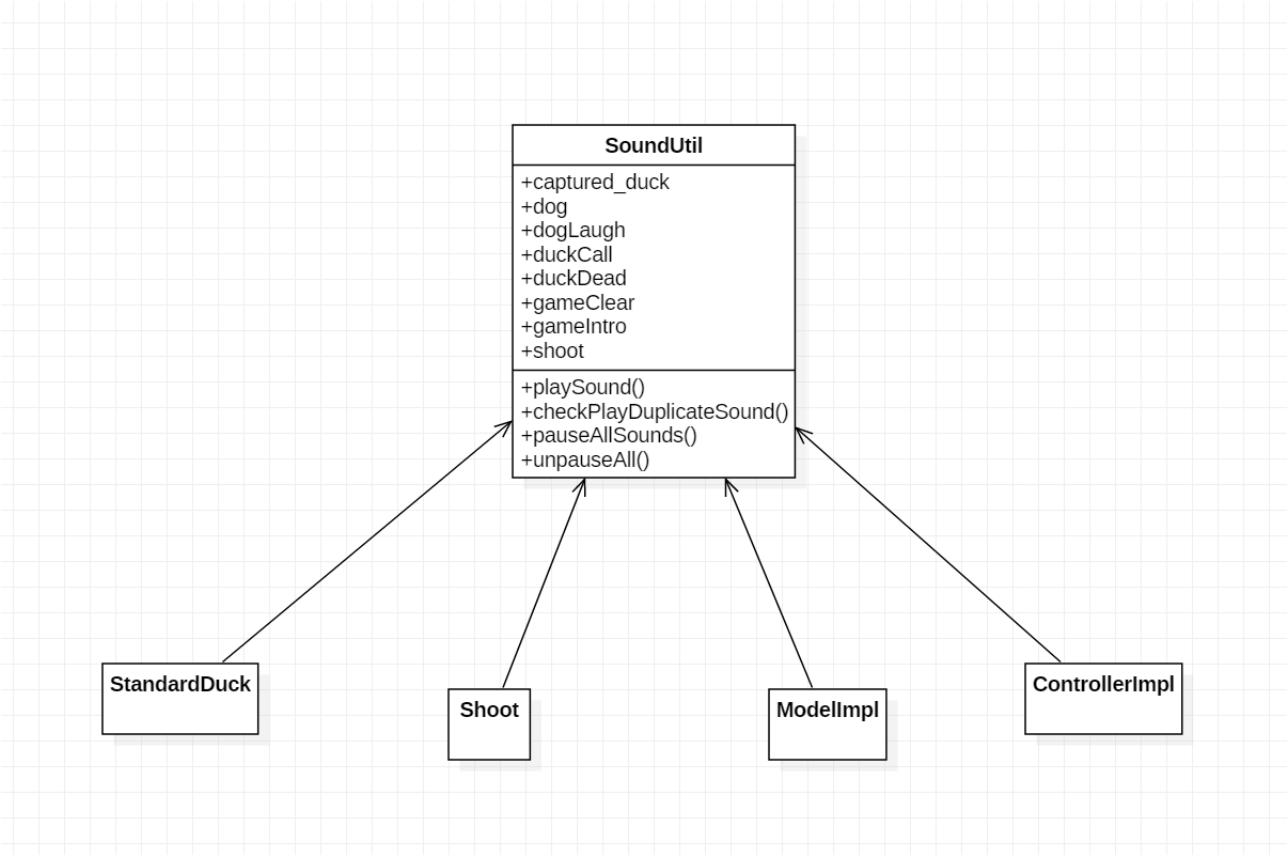


I comandi sono di tre tipi, elencati all'interno della enum "CommandType", due dei quali sono stati gestiti attraverso classi apposite.



Ho poi aggiunto degli "EventHandler" all'interno di "SceneLoader" per notificare il controller della pressione di tasti o mouse attraverso "notifyCommand" all'interno del quale viene anche eseguito il comando di pausa. Gli altri due comandi vengono invece eseguiti attraverso "processInput" che viene richiamato all'interno del GameLoop.

Audio



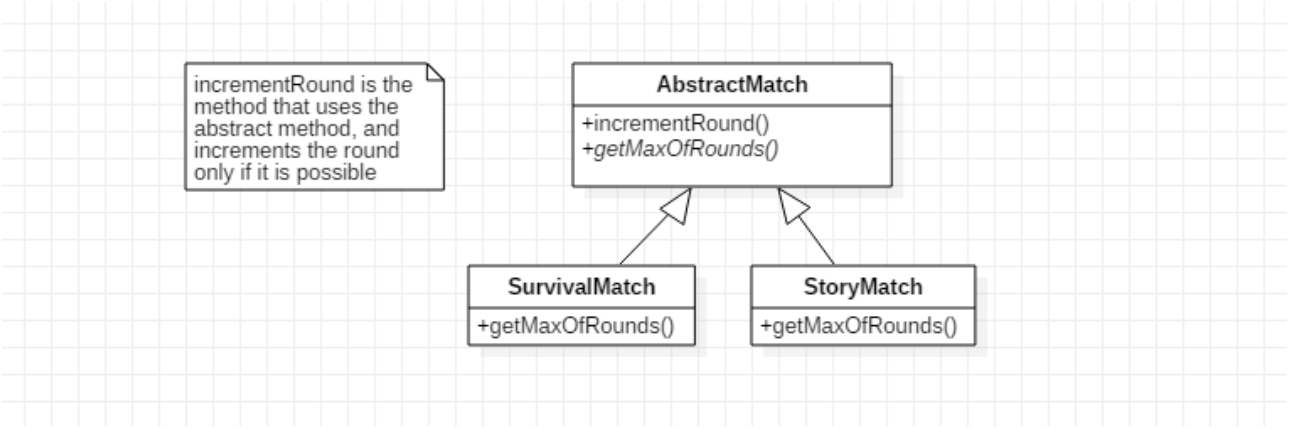
Per la riproduzione dei suoni è stata scelta la classe AudioInputStream. Abbiamo creato diversi campi statici rappresentanti i vari effetti audio del gioco con relativi getter richiamati in diverse parti del codice per la riproduzione. Questa avviene solo dopo aver controllato lo stato delle impostazioni dell'audio.

Simone Romagnoli

Il mio lavoro nel progetto è relativo ai dati impliciti degli utenti, quali gestione dei singoli account con salvataggio di username, password e obbiettivi raggiungibili; ho gestito anche i dati delle partite per ogni modalità di gioco, come punteggio e regole di game over.

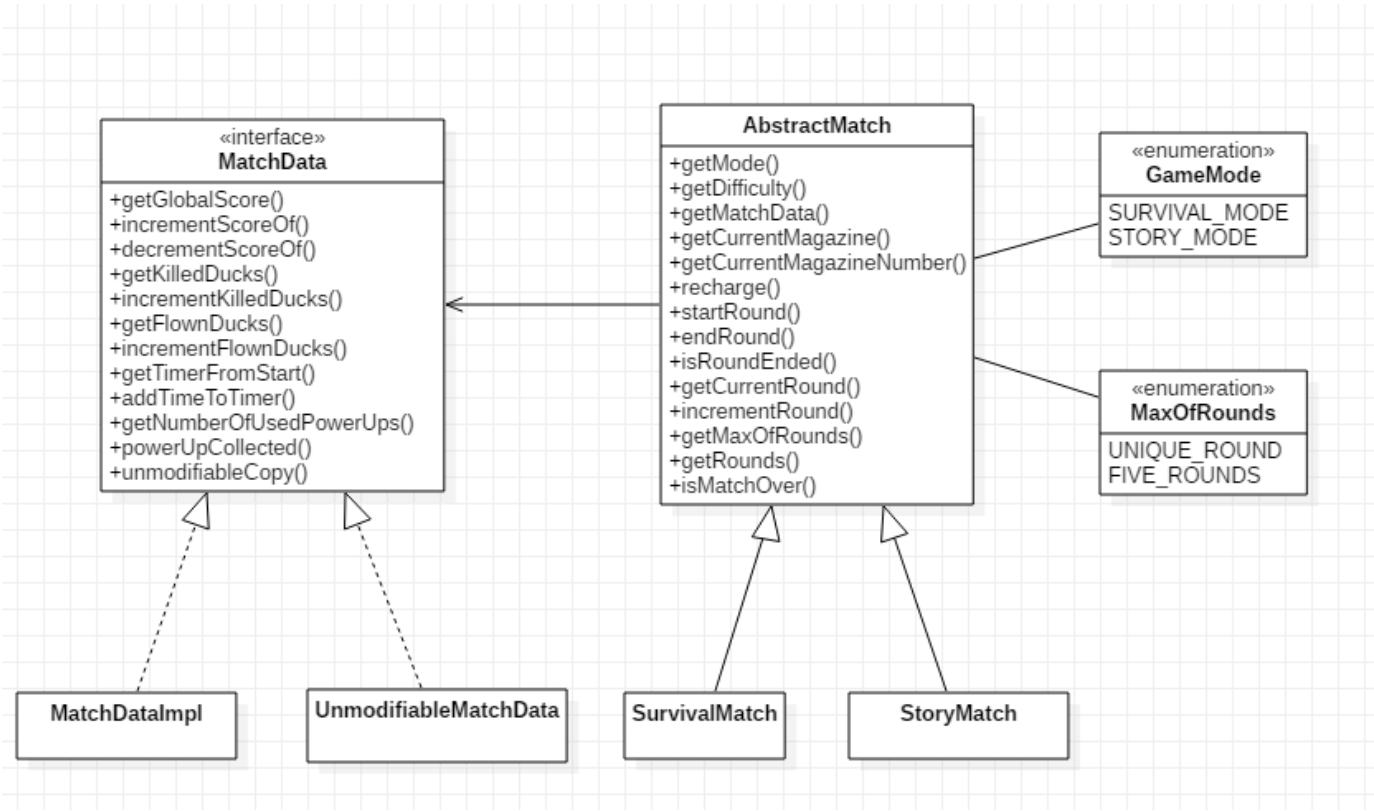
Gestione delle partite

Al momento della stesura dei requisiti del gioco, abbiamo considerato come opzionale la presenza di più modalità di gioco; confrontandoci con altri progetti e giochi, abbiamo optato per inserire le due modalità più frequenti, vale a dire Storia (avanzamento a round) e Sopravvivenza (a tempo o resistenza). Le singole partite vengono gestite all'interno del package model.matches: la classe che rappresenta i dati interni di un match è AbstractMatch; quest'ultima racchiude gli aspetti globali condivisi da ogni modalità di gioco, mentre StoryMatch e SurvivalMatch contengono le regole specifiche, rispettivamente, delle modalità Storia e Sopravvivenza. Poiché, a livello comportamentale, le due modalità di gioco sono diverse, ho deciso di definire dei metodi nella classe astratta e lasciare l'implementazione di aspetti specifici alle due sottoclassi. Ad esempio, ogni modalità ha le proprie regole di game over: nella Story Mode è necessario superare un determinato punteggio per passare al round successivo (e si ha un numero limitato di munizioni), mentre nella Survival Mode si perde quando vola via la n-esima anatra. La successione dei round è gestita tramite **Template Method**: se la sessione di gioco prevede più di un round, allora l'aumento di questi è ammesso; in questo modo, un'eventuale aggiunta di modalità di gioco a round in futuro non richiederebbe la gestione di tale questione.



I dati interni alla partita sono raggruppati in una classe poiché sia Model, sia Controller, sia View hanno bisogno di comunicare con tali dati; la classe MatchData, infatti, contiene le informazioni base che costituiscono l'essenza di ogni partita

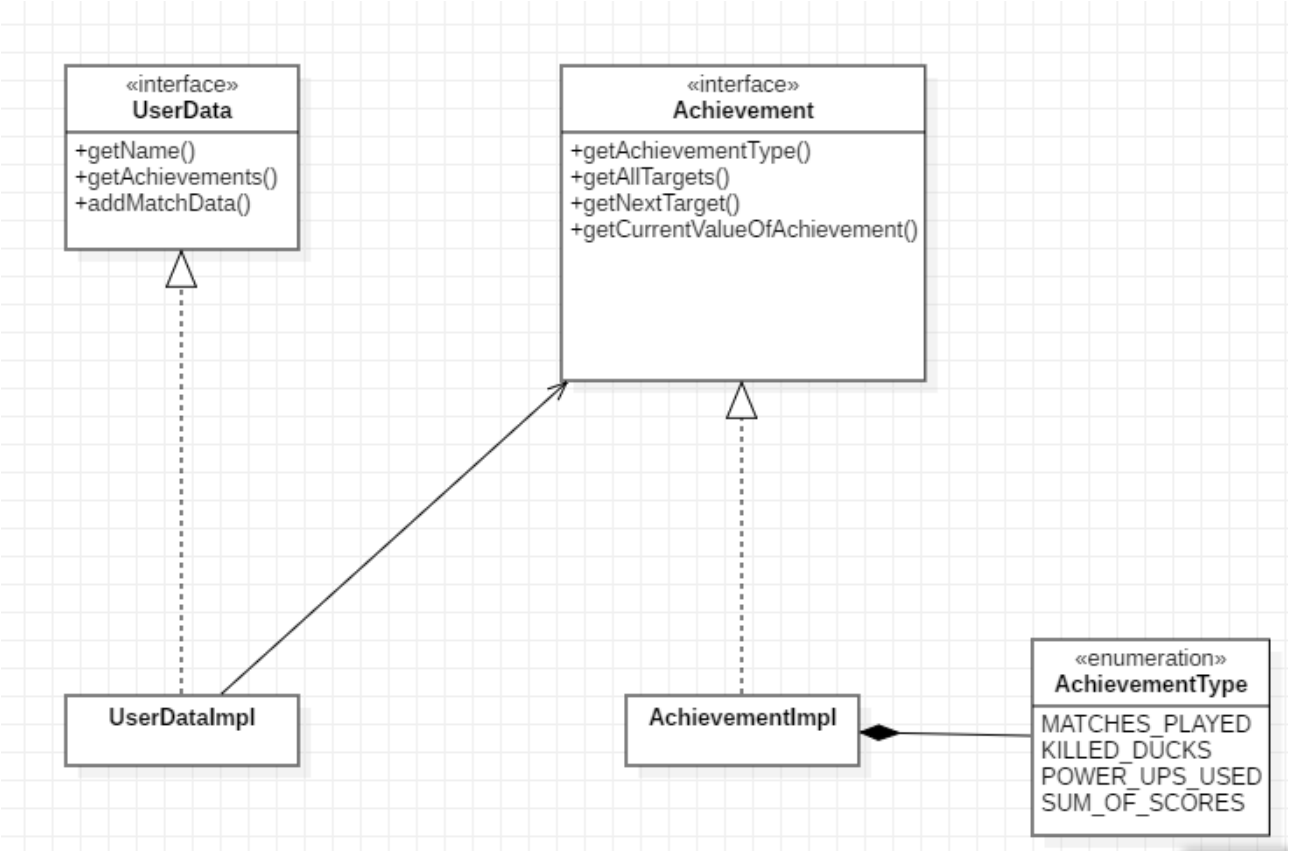
(punteggio, anatre uccise, anatre scappate, potenziamenti e tempo di gioco). L'implementazione di MatchData è estremamente semplice, con getter e setter che vengono continuamente richiamati durante la sessione di gioco; siccome le parti di View che utilizzano la classe in questione dovrebbero solo utilizzare i getter per mostrare a video determinati valori durante la partita, ho deciso di utilizzare il pattern **Proxy**, tramite un'istanza di MatchData chiamata UnmodifiableMatchData, la quale ammette l'utilizzo dei soli getter.



Utenti

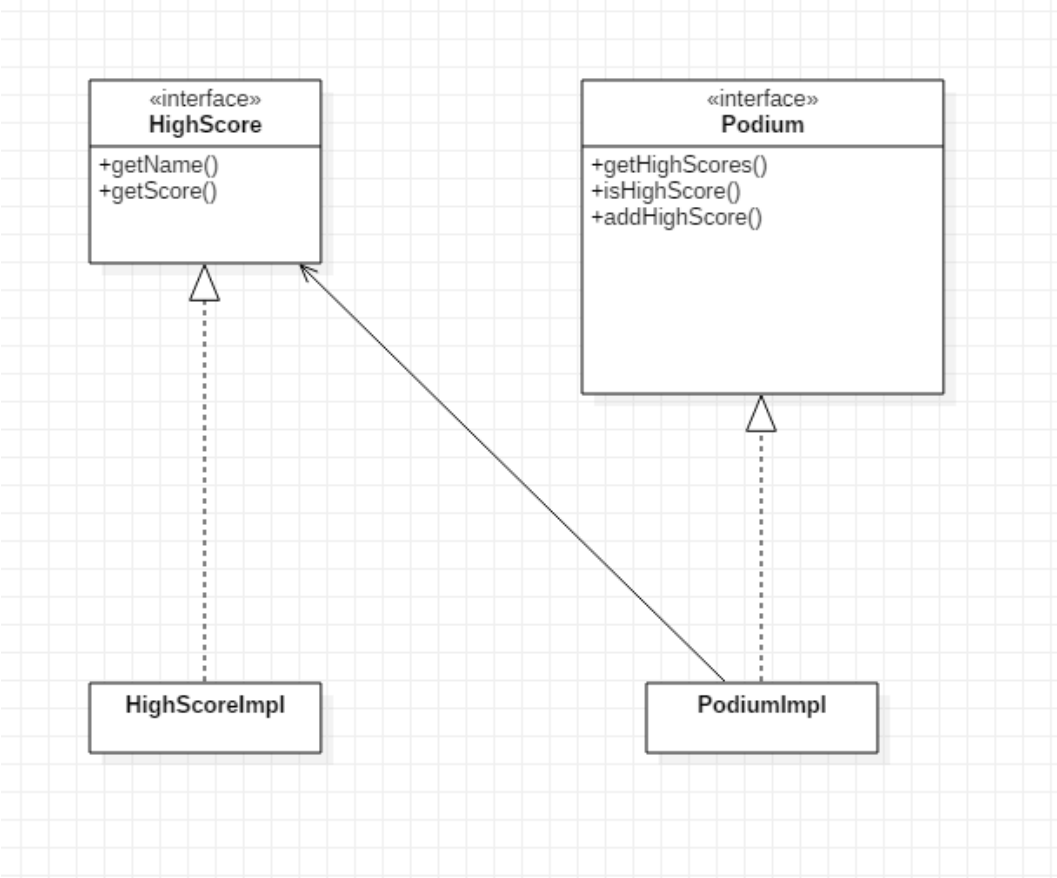
Abbiamo deciso di porre delle ideali sfide ad ogni possibile giocatore, così ci è venuta l'idea della creazione di account, contenenti un percorso da completare per il giocatore. Il concetto è quello di far giocare un utente fino al raggiungimento di determinati obiettivi: giocare tante partite, colpire tante anatre, raccogliere tanti potenziamenti e collezionare un totale di punteggi alto. Ogni account, idealmente identificato dalla classe UserData, ha un nome identificativo e una mappatura degli obbiettivi da raggiungere, i quali vengono aggiornati alla fine di ogni partita: non vengono aggiornati, invece, se il giocatore termina la sessione di gioco uscendo dal menù di pausa. Ogni obbiettivo ha una serie di target da raggiungere, al termine della quale potrà considerarsi completato: dal menù principale è possibile visitare la pagina

che mostra tali obbiettivi, in particolare essa mostra il livello attuale aggiunto e il target successivo per ogni tipo di obbiettivo.



Punteggi massimi

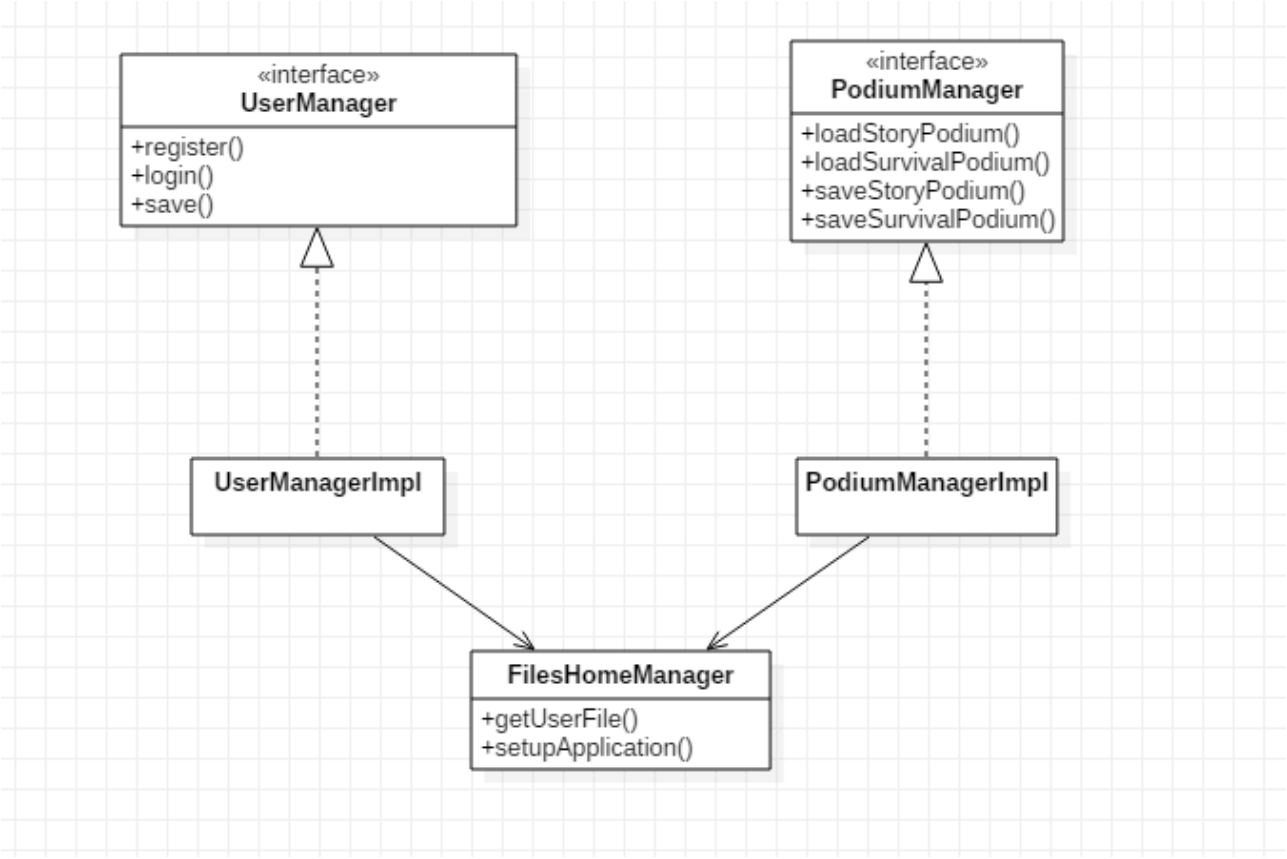
Il gioco tiene conto dei punteggi massimi fatti da tutti i giocatori che hanno effettuato l'accesso: la modalità a punti, i potenziamenti e le anatre di diverso tipo spronano gli utenti a fare del proprio meglio durante le partite, così da ottenere un record personale. Per l'esposizione dei record sono stati considerati importanti solo il punteggio ottenuto e l'username dell'utente che lo ha raggiunto. Ogni giocatore può visitare la pagina degli High Scores dal menù principale per accertarsi di essere entrato nel podio del gioco: questa pagina, infatti, mostra i 5 punteggi più alti registrati da tutti gli utenti che hanno giocato. Ovviamente, ogni modalità di gioco ha un suo podio e i punteggi massimi vengono mostrati in ordine decrescente, dando il titolo di "numero 1" all'utente col punteggio più alto. Alla fine di ogni sessione di gioco, il podio viene aggiornato con il punteggio appena ottenuto durante la partita.



Salvataggi

All'interno del Controller ho inserito degli oggetti, che ho deciso di chiamare "manager", i quali hanno lo scopo di caricare e salvare altri oggetti, in particolare i file degli utenti e il podio di ogni modalità di gioco. I dati precedentemente citati vengono salvati, a partire dalla cartella home del sistema, in sottocartelle indicate: la struttura ad albero di queste directory viene creata al primo avvio dell'applicazione e ripresa in quelli successivi. All'interno della cartella degli utenti, un file di testo tiene conto di tutti gli utenti registrati: in particolare, salva username e password (dell'ultima viene salvato l'hash code), ed è quindi il file nel quale avviene la ricerca che viene effettuata al momento del login all'avvio dell'applicazione. All'interno della medesima cartella vengono salvati i binari di tutte le istanze della classe UserData: in questo modo non vengono persi gli obbiettivi raggiunti durante le partite ed è possibile continuare il percorso degli Achievement anche in momenti diversi, chiudendo e riaprendo l'applicazione. Lo stesso concetto vale per gli High Scores: nel Controller è presente un manager per ogni modalità di gioco che si occupa di caricare il podio per la

corrispondente modalità ad ogni avvio dell'applicazione e di salvarlo ad ogni modifica.



Threads: GameLoop e Render

L'utilizzo dei thread è stato essenziale all'interno del nostro progetto: abbiamo deciso di utilizzarne due, uno che aggiorna perennemente il Model dell'applicazione, l'altro che fa la medesima cosa con gli oggetti da disegnare nella View e permette di cambiare gli FPS; il primo lo abbiamo nominato GameLoop, il secondo Render. Le due classi che li rappresentano estendono Thread (classe java) e sono innestate all'interno di Controller e View, rispettivamente. Tutti i componenti del gruppo hanno lavorato su di esse, visto il quantitativo di problemi e questioni che le riguardavano: in particolare, un problema che ci ha portato via un grosso ammontare di tempo sin dall'inizio del progetto, è stato quello della pausa. La pausa è stata gestita tramite una variabile booleana che non fa eseguire l'aggiornamento del Model al GameLoop: in questo modo, il thread del Controller sta in busy waiting, ma ci è sembrata una buona soluzione; invece, il thread della View non va in pausa, ma, dato che il Model rimane invariato, mantiene gli oggetti nella stessa posizione del momento in cui viene richiamata la pausa.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per i test automatizzati abbiamo deciso di utilizzare la libreria JUnit per testare le funzionalità basi del gioco. Altre funzionalità come quelle grafiche, animazioni e di input sono state testate manualmente dato che non disponevamo degli strumenti adatti.

Le parti di modello testate sono le seguenti:

- **Cane di gioco**: test sui diversi movimenti del cane, testate anche le operazioni non lecite su di esso.
- **Anatre**: sono state testate le caratteristiche principale delle anatre come i movimenti, l'uccisione dell'anatra e lo scappare via dopo tot secondi.
- **Anatre operazioni illecite**: testate le operazioni che non possono essere fatte sulle anatre come ad esempio il fatto di non poter colpire un'anatra mentre essa vola via oppure ottenere il suo punteggio senza prima colpirla.
- **Potenziamenti:** sono stati testati le loro creazioni dopo la morte delle anatre soprattutto se rispettassero l'algoritmo, la posizione iniziale, i movimenti e la loro attivazione.
- **Caricatori:** è stato testato lo scalo dei proiettili attraverso lo sparo, dei caricatori attraverso la ricarica e del game over in caso di munizioni esaurite.
- **Round**: test sull'effettivo aumentare dei round in base alla modalità di gioco e sul limite massimo di questi.
- **Punteggi**: è stato testato l'aumento del punteggio con diversi valori, ma anche la sua decrescita, in quanto è possibile che esso decresca sparando al cane quando salta fuori dall'erba.
- **Operazioni illecite sui dati**: si è verificata la giusta applicazione del pattern Proxy sui dati di gioco.

Le funzionalità che sono testate manualmente:

- **Movimenti e animazioni del cane**: il cane si muove correttamente e le animazioni vengono mostrate correttamente, ad ogni iterazione con le anatre il cane compare fuori dall'erba mostrandosi in diverse immagini.
- **Movimenti e animazioni delle anatre:** le anatre volano all'interno della schermata di gioco e cambiano direzione correttamente anche quando incontrano i bordi di gioco, tutte le animazioni funzionano nel modo giusto.
- **Generazione delle anatre:** le anatre vengono sempre generate al di fuori dello schermo a destra o sinistra, vengono rispettati i limiti di anatre nello schermo secondo le due diverse modalità.
- **Le varie scene di gioco:** sono state testate ogni singola schermata per vedere se veniva adattata alle varie caratteristiche di display.

3.2 Metodologia di lavoro

Alla fine di tutto possiamo dire che la suddivisione dei compiti è risultata abbastanza equilibrata e bilanciata con quanto detto all'inizio. Non ci sono state modifiche rilevanti e non ci sono state eccessive dipendenze tra le nostre parti.

Le funzioni elencate dal gruppo sono le seguenti:

- **Mattia Achilli**: gestione delle entità di gioco: anatre, cane. Parte di View delle entità (immagini) e generazione anatre.
- **Yuri Bernardini**: gestione delle entità di tipo power up. Creazione, generazione e navigazione delle schermate di gioco, impostazioni del gioco.
- **Giulio Dulja:** gestione di munizioni e caricatori, gestione dell'input all'interno del gioco, audio.
- **Simone Romagnoli:** gestione dei dati di partita: round, punteggi, anatre uccise e scappate; punteggi record e obbiettivi personali; salvataggio su file dei dati permanenti.

Le parti in comune tra i componenti del gruppo sono:

- Le parti di codice contenti la grafica principale di gioco (ViewImpl) e il modello (ModelImpl) sono state sviluppate da tutti i componenti del gruppo poiché contengono parti comuni.
- La gestione del Controller (ControllerImpl) sono stati partecipi Achilli Mattia e Romagnoli Simone regolando le varie iterazioni tra le parti MVC.
- La gestione dei Thread e della sincronizzazione riguardanti il Game Loop e il Render per la grafica cui hanno contribuito Achilli Mattia e Romagnoli Simone gestendo anche l'aspetto di pausa del gioco (più contributo di Romagnoli).
- La gestione dello sparo attraverso l'input è stata sviluppata da tutti i membri del gruppo in quanto ognuno si ritrovava in una sua parte (come lo sparo alle papere o l'incremento di punteggio nei dati di gioco).
- Sono state create interfacce e classi di utilizzo comune soprattutto nella gestione del modello o classi con metodi statici di utility utilizzate svariate volte.

Nella fase di sviluppo abbiamo utilizzato il DVCS Git. Ognuno ha lavorato sulla propria fork, proponendo una pull request una volta arrivati a delle modifiche rilevanti anche per il lavoro altrui.

3.3 Note di sviluppo

Prima della fase di progettazione e analisi del progetto abbiamo guardato diversi video riguardanti pattern di progettazione per videogiochi da cui abbiamo appreso il pattern principale di sviluppo Game Loop utilizzato nella maggior parte dei videogiochi. Uno dei video di riferimento è stato "Game as a Lab", tenuto dal prof. Ricci. I progetti di amici da cui abbiamo preso idee e inizialmente buono spunto sulla struttura sono: oop17-MagnumChaos e oop17-OOPang.

Mattia Achilli

Stream: utilizzati per cose basilari in qualche parte di codice per diminuire righe di codice.

Lamda Expression: utilizzate per ottenere un codice più breve e compatto quando ho utilizzato gli stream.

Optional: utilizzati in campi in valori di ritorno qualora ci fossero valori nulli o non ottenibili in quel preciso istante di gioco.

Libreria Common: utilizzata per non dover ricreare classi già definite in librerie come ad esempio la classe Pair.

JavaFx: per gestire le immagini relative alle entità di gioco come quelle delle anatre o del cane.

Per i pattern progettuali ho prima consultato le dispense viste a lezione per ricordare i pattern basilari mentre per pattern più specifici ho guardato diversi file in rete di cui purtroppo non ricordo i link. Per gestire la creazione delle anatre nelle due modalità ho preso grande spunto dal progetto Magnum Chaos, in seguito ho modificato diverse cose perfezionando ulteriormente la progettazione e aggiungendo altro codice. Per altri parti di codice riguardanti il linguaggio Java ho sempre preso spunto dalle dispense viste a lezione durante il corso.

Bernardini Yuri

Lambda: ho scelto in alcuni punti l'utilizzo di lambda per avere codice più compatto e veloce da scrivere.
Stream: usati soprattutto per classi che richiedevano operazioni di filtraggio, inserimento o ordinamento.
Libreria Common: utilizzata per non dover ricreare classi già definite in librerie come ad esempio la classe Pair.
Optional: usati in vari contesti tra cui nella generazione di PowerUp poiché non tutte le anatre alla morte ne possiedono uno.
FXML e JavaFx Scene Builder: utile per descrivere l'aspetto estetico delle varie scene di gioco.
Css: per attribuire decorazioni ai componenti delle scene del gioco.

Per l'apprendimento di nuove funzionalità mi sono avvalso di tutto il materiale messo a disposizione dal corso soprattutto le slide di corso e javadoc.
Aggiungo che le varie immagini di gioco sono state prese dal sito: https://github.com/vitcb/DuckHunt altre create da zero o modificate tramite un editor grafico preinstallato su Windows in maniera minimale (Paint).
Per un veloce apprendimento di utilizzo di java fx ho consultato il seguente link: https://www.youtube.com/watch?v=-bYXID7toG4 in cui mi ha spiegato i vari componenti e le loro caratteristiche.
Dopo di che seguendo le slide del nostro corso di laboratorio ho installato JavaFx Scene Builder e iniziato a creare numerosi progetti di prova per apprenderne i diversi funzionamenti.
Detto ciò, tutte le scene di grafica sono frutto di una serie di miei continui esperimenti e tentativi, nonché la realizzazione del foglio di stile css è frutto di mie conoscenze in ambito di altri linguaggi di markup studiati negli anni passati; l'unica decorazione che

ho ritenuto interessante riusare è la forma del pulsante per tornare indietro (http://fxexperience.com/2011/12/styling-fx-buttons-with-css/).
Mentre per le tecniche di navigazione tra le varie scene di gioco ho ritenuto interessante riutilizzare la metodologia usata nel Progetto Magnum Chaos, poiché utilizzano una semplice e intuitiva divisione delle classi SceneLoader e SceneFactory.

Giulio Dulja

Lambda Expression: utilizzate dove necessarie a semplificare codice altrimenti più complesso e meno intuitivo.
Optional: utilizzati per la gestione dell'input da parte dell'utente.

Per l'utilizzo del pattern command ho utilizzato diverse fonti, tra cui siti web o video, e ho inoltre consultato i progetti precedentemente riportati per l'implementazione delle interfacce per la gestione di comandi e per la gestione dell'input. I file audio come le immagini sono stati presi da un progetto trovato in rete (https://github.com/vitcb/DuckHunt) tranne per lo sparo che abbiamo invece preso su internet.

Simone Romagnoli

Stream: ho cercato di utilizzare al meglio la versatilità degli Stream sia per semplici creazioni di oggetti, sia per costrutti più complicati, come, ad esempio, l'esibizione dei punteggi record nello specifico menù "High Scores".

Lamda Expression: utili nell'utilizzo di strategy e, in generale, nella semplificazione del codice.

Optional: essenziali per il caricamento degli utenti e del podio qualora partissero delle eccezioni.

In merito allo sviluppo delle parti dell'applicazione di cui mi sono occupato, oltre ad essermi impegnato su una buona base per i dati impliciti delle partite, mi sono concentrato sulla serializzazione delle classi UserData e Podium e sulla relativa gestione dei dati sul filesystem. In seguito ad una prima fase di esperimenti, mi sono documentato su possibili soluzioni esterne e ne ho trovata una convincente nel lavoro del progetto oop17-OOPang: oltre all'interessante meccanismo di hashing utilizzato per la protezione delle password, il codice in questione mi è sembrato semplice a prima vista, ma di totale efficienza; così ho deciso di utilizzare alcuni meccanismi ivi utilizzati, in particolare, quello di salvataggio dei dati su file con l'aiuto delle classi java.io.FileInputStream e java.io.FileOutputStream, e la semplice struttura delle cartelle create nella home del sistema. Ho quindi compattato queste classi all'interno

del package controller.files. Per quanto riguarda l'utilizzo dei pattern, ho, innanzitutto, consultato le slide viste a lezione e ho notato che, per la mia parte di sviluppo, non molti erano applicabili: perciò ho cercato di utilizzarli solo dove avrebbero favorito l'estendibilità del progetto per eventuali cambiamenti o aggiunte future.

Capitolo 4

Commenti Finali

4.1 Autovalutazione e lavori futuri

Mattia Achilli

Sono molto soddisfatto del lavoro svolto dato che non avevo mai sviluppato né tanto meno visto come fosse progettato un videogioco seppur non di altissimo livello.
Mi è piaciuta soprattutto la parte di analisi e progettazione all'inizio in quanto è molto importante per poter sviluppare al meglio senza dover cambiare parti di progetto durante l'implementazione.

Ho imparato a conoscere e ad utilizzare diversi pattern di progettazione conosciuti e molto utili per rendere di alta qualità il codice scritto.
Mi è piaciuto anche interessarmi del lavoro fatto da altri per dare suggerimenti, aiuti o semplicemente per imparare qualcosa di nuovo come per la parte grafica.
Ho sicuramente migliorato le mie doti di progettazione e di programmazione ad oggetti che sicuramente approfondirò.

Penso che questo progetto possa avere un futuro migliorando diversi aspetti tra cui sicuramente la parte grafica, e l'implementazione di altre entità o elementi all'interno del gioco però per cause di tempo rimarrà purtroppo solo per scopi didattici.

Bernardini Yuri

Sono piuttosto soddisfatto del lavoro prodotto, ho cercato di sviluppare codice più semplice e comprensibile possibile così da rendere facili eventuali modifiche future.
Questo è stato la mia prima applicazione sviluppata in java nonché il mio primo corso affrontato sull'argomento;
In passato avevo partecipato ad altri progetti di gruppo ma lo sviluppo di questo, in particolare con la creazione di un gioco dinamico, penso sia stato il più interessante e quello che mi ha fornito maggiori conoscenze sotto molteplici aspetti.
Grazie a questo progetto ho compreso e imparato il vero uso di Git, software che conoscevo già ma che non ero in grado di usare correttamente.

Sono contento di aver lavorato in un gruppo dove non conoscevo la metà dei componenti e in cui vi ho trovato punti in comune, per questo aspetto vicino ad un ambiente di lavoro vero e proprio.
Penso però che dalla mia parte, visto il poco tempo a disposizione tra lavoro e università, questo progetto non potrà avere ulteriore futuro.

Giulio Dulja

È stato molto stimolante poter passare in così poco tempo dal non avere idea di come sia realizzato un videogioco, seppur di livello non troppo avanzato, al realizzarlo in autonomia.

Una parte di cui ho sicuramente notato l'utilità è stata quella della progettazione iniziale, nel bene e nel male, infatti, non ancora molto esperti, avevamo considerato mirino e munizioni delle entità, come il cane e le anatre. Mettendoci poi a lavorare al progetto ci siamo invece accorti di come dovessero invece essere strutturati in modo diverso, fortunatamente più semplice.

Ho imparato e apprezzato il lavoro di gruppo utilizzando git, strumento che avevo molto sottovalutato durante le lezioni. Mi è piaciuto in particolare come sia stato possibile confrontarci e aiutarsi in ogni aspetto del videogioco, potendo così ampliare la conoscenza della programmazione ad oggetti.
Questa esperienza sarà infatti sicuramente utile per lavori di gruppo e realizzazione di progetti simili.

Simone Romagnoli

Grazie al lavoro svolto sono riuscito a comprendere le parti più critiche della gestione di un software complesso: non avendo mai sviluppato prima un software di questa grandezza, penso e spero che questo “scoglio” iniziale mi abbia dato una buona spinta nel mondo informatico. L’utilizzo del DVCS Git è stato molto istruttivo dal mio punto di vista: mi ha aiutato a comprendere il giusto metodo di sviluppo e di lavoro in gruppo. Inizialmente, tendevo a buttarmi frettolosamente sulla scrittura del codice senza prima passare da una fase di progettazione ed organizzazione, nonostante le raccomandazioni di professori e colleghi: il lavoro di gruppo poi mi ha spronato a seguire “il flusso organizzativo” di tutti e, in generale, scrivere codice successivamente è risultato molto più logico e intuitivo. Per questo progetto ho avuto la fortuna di poter documentarmi studiando e analizzando progetti interi degli anni passati, evitando problemi e questioni affrontati in altri lavori: la fase di sviluppo che chiamo “di documentazione” mi è stata utile, non solo per capire i meccanismi adatti per diversi problemi, ma anche le questioni più frequenti nel mondo della programmazione a oggetti. Vista la disomogeneità delle conoscenze nel gruppo, ho spesso costretto gli altri a rallentare per poter stare al mio passo, ma, nonostante ciò, credo che il lavoro di gruppo sia comunque stato svolto nella maniera corretta. Personalmente reputo questo progetto come un trampolino per i lavori futuri: credo mi serva sia come dimostrazione delle mie capacità, sia come ammonimento per non commettere, negli anni a venire, errori commessi in passato.

4.1 Difficoltà incontrate e commenti per i docenti

Bernardini Yuri

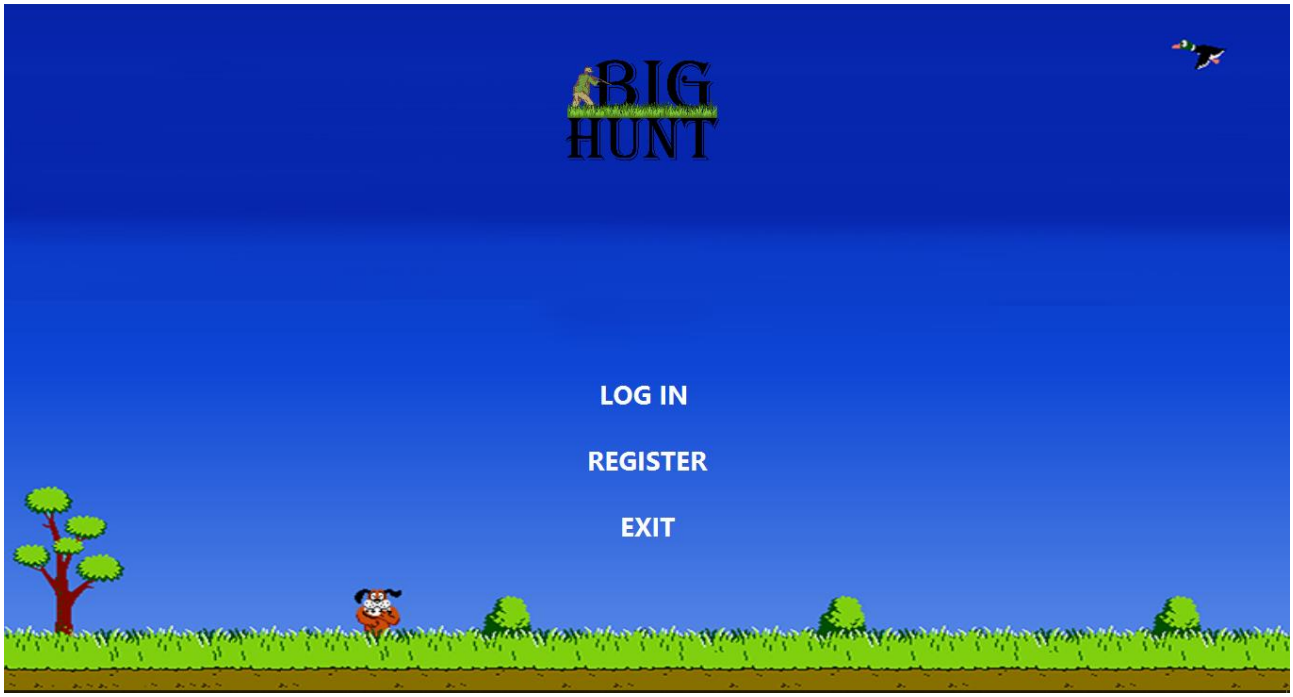
Sicuramente l’aspetto che mi ha portato via più tempo è quello per la realizzazione delle varie schermate di gioco, dato che avendo affrontato poco questi aspetti a lezione ho dovuto studiarmi e comprendere questo nuovo ambiente. La parte più scomoda è quella in cui cercavo di riadattare le schermate alle varie risoluzioni dei display.
Inizialmente avevamo pensato di poter cambiare la risoluzione di gioco. Ho notato che i vari contenitori hanno proprietà che li rendono adattabili in automatico ma poi però il loro contenuto, specialmente il font e le dimensioni dei componenti non

cambiavano; Ero giunto ad un punto in cui tutte le varie scene di gioco si riadattavano ad eccezione del Canvas nella scena di gioco, consultando i vari forum su internet ho notato che sono problemi molto ricorrenti con soluzioni non chiare e diverse così visto il poco tempo a disposizione rimasto abbiamo deciso di rendere il gioco unicamente full screen in base, naturalmente, allo schermo su cui è eseguito ma resta comunque facilmente estendibile per gestire altre risoluzioni.
In Negativo ritengo che il tempo passato per concentrarmi sul progetto mi abbia tolto molte ore ad altre materie e, a proposito dell'argomento affrontato in precedenza penso che sarebbe stato bello poter creare a lezione applicazione con una grafica un po' più importante e vicina ai progetti proposti da noi studenti.

Simone Romagnoli

Inizialmente ho avuto difficoltà nello sviluppo delle classi viste come singole entità: non mi era ben chiaro che tipo di relazione dovessero avere e come avrebbero dovuto comunicare tra di loro tramite il Model e il Controller; ma tale problema si è risolto automaticamente progredendo con il progetto. Un altro problema che mi ha tenuto incollato al computer è stato quello della pausa del gioco: inizialmente pensavo di implementarla con i metodi wait() e notifyAll() della classe Thread, ma il risultato non è stato soddisfacente; di conseguenza, ho deciso di farla con busy waiting del GameLoop e di "unire" la scena di pausa e quella di gioco in un unico StackPane. Oltre a consigliare di porre maggiore attenzione sulla programmazione concorrente e sull'utilizzo di SceneBuilder, non ho nessun commento considerevole per i professori.

Appendice A
Guida utente



Appena avviato il gioco, l'utente si troverà davanti ad una schermata di login: in caso di mancato account sarà costretto a registrarsi.
Una volta effettuato l'accesso, si presenterà una schermata con le seguenti voci:

- **New game:** dà la possibilità di iniziare una nuova partita, si dovrà selezionare tra due modalità:
    - **Story:** in questa modalità il giocatore dovrà ottenere un certo numero di punti ad ogni round colpendo le anatre.
      La partita può inoltre terminare se:
        - Il giocatore non raggiunge il minimo punteggio stabilito per ogni round
        - Finisce tutti i proiettili disponibili (20 caricatori da 8 colpi l'uno).
        - Tutti i round sono stati completati.
    - **Survival:** in questa modalità a sopravvivenza il giocatore dovrà cercare di colpire più anatre possibili.
      I colpi a disposizione saranno illimitati e la difficoltà di gioco aumenterà man mano che passa il tempo.
      Il gioco termina quando sono volate via un numero prestabilito di anatre che varia dalla difficoltà selezionata.

- **Achievements:** tale pagina esibisce la situazione degli obbiettivi personali del giocatore; nello specifico, mostra il valore raggiunto dall'utente, seguito dal prossimo target nella serie degli obbiettivi oppure dalla scritta "ACHIEVEMENT COMPLETED" nel caso la serie di obbiettivi sia terminata.
- **High Score:** mostra il podio (i 5 punteggi più alti) per ogni modalità di gioco, specificando la posizione nella classifica, il punteggio e il nome del giocatore che l'ha registrato.
- **Manual:** espone un breve riassunto del gioco, con la spiegazione delle conseguenze che occorrono quando si colpisce un potenziamento e il punteggio che viene accreditato al giocatore quando uccide le anatre in base al loro colore.
- **Settings:** permette di cambiare le impostazioni di gioco quali la difficoltà di gioco, gli fps desiderati e la presenza o meno di audio.

Una volta iniziata la partita il giocatore dovrà colpire le anatre attraverso l'utilizzo del puntatore mouse sparando tramite il tasto sinistro del mouse, al finire delle munizioni del caricatore l'utente dovrà premere il tasto "**R**" per caricare (come mostrato dalla comparsa di un messaggio).

Durante il corso della partita si potranno ottenere dei potenziamenti che vengono rilasciati una volta uccise le papere, i potenziamenti sono sotto forma di stella variano di colori e sono i seguenti:

- **Stella Verde:** Punteggio doppio per le prossime 3 anatre uccise.
- **Stella Gialla:** I proiettili non vengono tolti quando si spara per diversi secondi.
- **Stella Rossa:** Questo potenziamento prevede che 3 anatre rallentino i loro movimenti.
- **Stella Nera:** Uccide immediatamente 3 anatre.

Premendo il tasto "**ESC**" il gioco si metterà in pausa, basterà premere nuovamente "**ESC**" o "**RESUME GAME**" per far sì che il gioca riprenda oppure basterà cliccare su "**GO TO MENU"** per tornare al menù di gioco. Alla fine della partita verranno mostrati diversi dati riguardanti la partita tra cui: punteggio ottenuto, anatre uccise e tempo trascorso.