Relazione Progetto OOP 2022

Stefano Furi
Luca Rapolla
Michele Montesi
Ezmiron Deniku

24 aprile 2022

Indice

1 Analisi **2**
1.1 Requisiti 2
1.2 Analisi e modello del dominio 3

2 Design **5**
2.1 Architettura 5
2.2 Design Dettagliato 7
2.2.1 Stefano Furi 7
2.2.2 Michele Montesi 11
2.2.3 Luca Rapolla 15
2.2.4 Ezmiron Deniuku 20

3 Sviluppo **26**
3.1 Testing Automatizzato 26
3.2 Metodologia di lavoro 27
3.3 Note di sviluppo 30

4 Commenti Finali **33**
4.1 Autovalutazione e lavori futuri 33

Appendices **36**

A Guida utente **37**

B Esercitazioni di laboratorio **38**

Capitolo 1

Analisi

1.1 Requisiti

Il software costruito mira all'emulazione del *Retro Game* Pang. Con Retro Game si intende un gioco di una precedente generazione, il quale necessita di un emulatore per essere giocato al giorno d'oggi, in quanto difficilmente reperibile in forma fisica.

Requisiti Funzionali

Il gioco è composto da tre scenari principali: Menù d'inizio, sequenza di gioco e fine gioco.

- Nel *Menù di Inzio* è possibile scegliere il proprio `Nickname`, visualizzare la *Leaderboard* locale ed avviare la sequenza di gioco.
- Durante Sequenza di gioco, il giocatore controlla un personaggio, il quale dovrà far scoppiare delle sfere (ognuna sdoppiabile per tre volte) tramite l'utilizzo di un arma. Una volta finite le sfere, il giocatore passa al prossimo stage, dove la difficoltà aumenta. Durante la sessione di gioco, compare (ad intervalli randomici) un nemico il quale, planando verso un punto specifico, potrebbe colpire il personaggio principale. Vengono inoltre generati dei bonus che, se raccolti dal personaggio, incrementano il punteggio della partita corrente. Insieme a questi ultimi, nello stage di gioco possono comparire anche PowerUp, che hanno lo scopo di modificare lo stato attuale del gioco. Ogni volta che il personaggio viene colpito, viene decurtata una vita.

- Al termine delle tre vite, si passa alla fine del gioco dove il punteggio viene salvato nella *Leaderboard* e viene chiesto all'utente se si desidera effettuare una nuova partita, oppure abbandonare l'applicazione.

Requisiti non funzionali

In quanto RetroGame, il gioco deve poter funzionare anche su architetture a basse prestazioni.

1.2 Analisi e modello del dominio

Il software dovrà essere in grado di generare degli stage di difficoltà sequenziale. La difficoltà è determinata dal numero e dalla velocità delle sfere presenti nello stage di gioco. Ogni sfera si muove nello spazio rimbalzando sul terreno e sui bordi e ogni sfera ha la caratteristica di essere duplicabile nel caso in cui venga colpita da un proiettile. Con duplicazione si intende il passaggio da uno stato padre ad uno figlio, eseguibile per un massimo di due volte; in totale quindi tre stati (Padre-Figlio-Nipote). L'arma assegnata al personaggio prinicpale è un arpione il quale, a seconda del *PowerUp* attivo, può assumere tre utilizzi diversi:

- *Arpione Classico*: il personaggio può sparare un solo arpione per volta verso l'alto, e nel momento in cui l'arpione raggiunge il limite superiore dello stage, viene riaquisito.

- *Arpione Appiccicoso*: il personaggio spara verso l'alto un arpione per volta il quale, nel momento in cui raggiunge il limite superiore dello stage, si "appiccica" per un determinato intervallo di tempo.

- *Doppio Arpione*: il personaggio può sparare due arpioni per volta senza il bisogno di aspettare che il primo dei due arrivi al limite superiore dello stage.

Nello scenario è inoltre presente un uccello il quale attraversa lo stage orizzontalmente e durante la attraversata, può planare verso una determinata posizione per poi continuare il suo percorso. Anche l'uccello può essere colpito dall'arpione. Il giocatore ha a disposizione un certo numero di vite, le quali venogno decurtate in caso di collisione con un entità avversa. Nel momento in cui si finiscono le vite, il gioco termina e si registra il punteggio nella *Leaderboard* nel caso sia abbastanza alto da superare quelli precedenti. Lo *Score* viene aumentato dall'acquisizione di Bonus che vengono generati

durante la partita. Ad ogni bonus viene assegnato un punteggio diverso. Oltre ai due PowerUp che influenza l'arma, ne sono presenti altri due in grado di influire sulle sfere:

- *Time Freeze*: Ferma tutte le sfere attive e le riattiva dopo un certo periodo di tempo.
- *Bomba*: Fa esplodere tutte le sfere a meno che non siano già al terzo stato (Nipote: stato più piccolo): in questo caso vengono ignorate.

A differenza dei PowerUp rivolti all'arma, gli ultimi due citati possono allo stesso tempo semplificare/peggiorare il *Gameplay* del giocatore, in quanto il loro vantaggio/svantaggio varia in base al momento in cui vengono raccolti. Gli elementi costitutivi del problema sono sintetizzati in figura Figura 1.1.

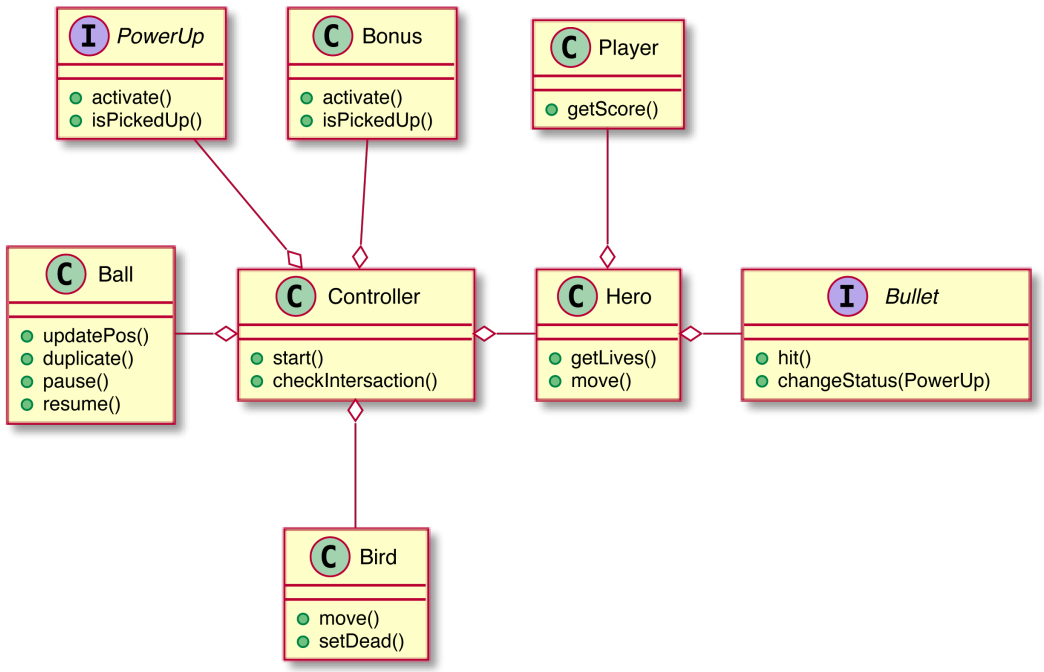


Figura 1.1: Rappresentazione UML del Contesto di gioco

Capitolo 2

Design

2.1 Architettura

L'architettura di Pang segue il pattern Architetturale MVC. In particolare, Pang si compone di un Controller composto da tre diversi sotto-controller i quali vengono istanziati all'interno di esso. Questi tre sono classi attive che fungono da gestori degli elementi portanti del model:

- `EntityHandler`
- `PickableHandler`
- `PauseHandler`

Il primo di essi ha lo scopo di gestire e mantenere le strutture dati di una sottoparte del model, e verificare le interazioni tra le diverse entità di quest'utilmo, per esempio, gestire le collisioni delle sfere con il personaggio. In particolare, questa classe fa uso di contenitori delle diverse entità; più nello specifico, le entità gestite sono `Ball`, `Bird`, `Arpion` e `Hero`. Queste ultime, sono le radici del model, e vengono incapsulate e gestite a loro volta da differenti *Handlers*. Sommando il tutto, la classe `EntityHandler` gestisce e coordina i relativi controller attivi per ogni entità (`BallHandler`, `EntityHandler`, `Hero` e `GunBag`) e fa in modo che le interazioni tra gli elementi del model avvengano correttamente.

Il secondo di essi fornisce un controllo attivo sugli elementi *Pickable* ovvero `Bonus` e `PowerUp`, e verifica se il personaggio di gioco raccoglie questi elementi.

L'ultima interfaccia invece, ha lo scopo di attivare la funzionalità della pausa durante il gioco, stoppando momentaneamente l'esecuzione dei controller degli elementi fondamentali del dominio.

Infine l'entry point della View è definito da `Visual`, una classe che ha il compito di ricevere costantemente posizioni aggiornate e "avvisare" la *GUI* che è si è verificato un cambiamento nel model, ed è quindi necessario un aggiornamento dell'interfaccia grafica. Questo permette di poter sostituire in blocco la parte di View, in quanto `Visual` e `Controller` lavorano indipendentemente dalla libreria grafica utilizzata.

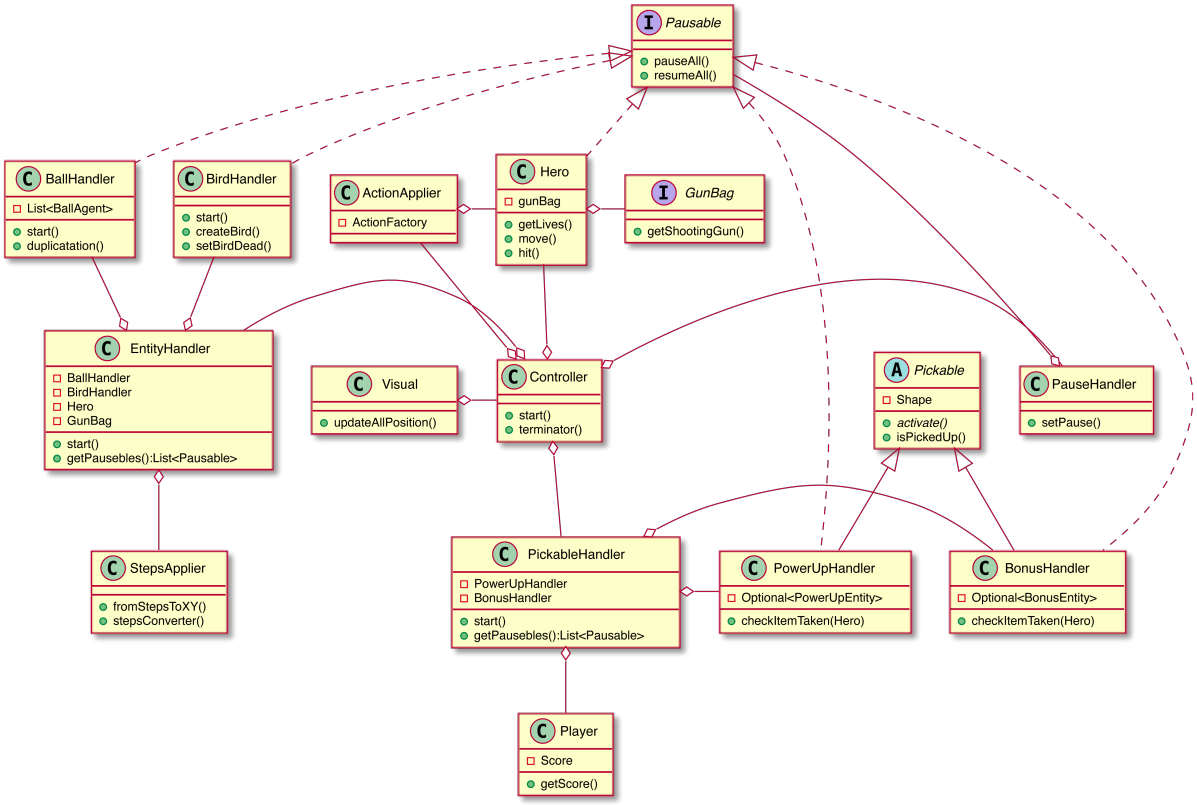


Figura 2.1: Rappresentazione UML di MVC, al centro `Controller` definisce l'entry point di tutti i sotto-controller e `Visual` permette di comunicare con la view indipentemente dalla libreria grafica implementata

2.2 Design Dettagliato

2.2.1 Stefano Furi

Gestione delle sfere e cambiamento del loro stato

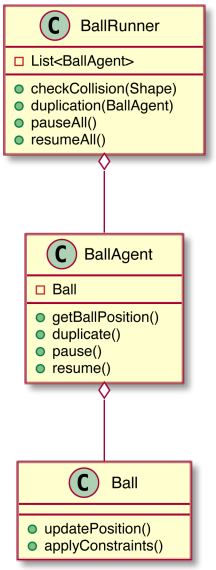


Figura 2.2: Rappresentazione UML della gestione delle Sfere

Problema: Pang deve essere in grado di gestire e cambiare lo stato (anche contemporaneamente) di tutte le sfere presenti in gioco in modo trasparente al client.

Soluzione: Il concetto più a basso livello della sfera `Ball` (struttura dati che mantiene elementi come velocità, angolo della curva, posizioni, ...) viene "wrappato" all'interno di un agente `BallAgent` (Thread), il quale ad ogni sua iterazione, aggiorna la posizione della sfera che mantiene all'interno di esso. Ogni sfera è quindi wrappata all'interno di un agente, i quali a loro volta, vengono mantenuti all'interno di un "Controller" `BallRunner` che gestisce le eventuali collisioni (contro un muro/pavimento, o contro un *Shape*(Arpione o Personaggio)), ed espone all'esterno i metodi di duplicazione delle, di pausa e di ripresa del movimento delle sfere.

Creazione di diversi tipi di sfera con comportamento differente

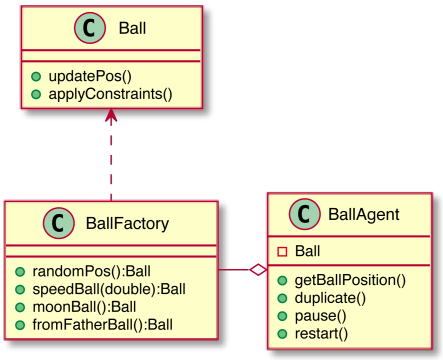


Figura 2.3: Rappresentazione UML del pattern Factory per la creazione di diverse sfere

Problema: In vista di Stage di gioco differenti e la continua creazione di nuove sfere successivamente ad una duplicazione, è necessario creare sfere con diverse proprietà e comportamenti. Questo provoca a una ripetizione continua nell'istanziazione di nuove sfere, e inoltre non rende il codice flessibile per future funzionalità aggiuntive.

Soluzione: Per la creazione di diversi tipi di sfera (sfere più lente, più veloci, con dimensioni diverse,...), è risultato ottimale l'utilizzo del pattern Factory, in quanto permette di istanziare in una classe separata tutti tipi di costruzioni per le sfere. Inoltre permette di fornire metodi funzionali per le sfere (degno di nota è `fromFatherBall()`, il quale risulta molto pratico durante la fase di duplicazione di una sfera). Il fatto di utilizzare la Factory per la creazione, aiuta l'estendibilità dell'applicazione, in quanto se si volesse implementare, per esempio, uno stage in base alla difficoltà, basterà creare una sfera tramite il metodo `speedBall()` che riceve un parametro in ingresso capace di definire la velocità della palla, e quindi, la difficoltà di evitarla.

Pickable: PowerUp e Bonus

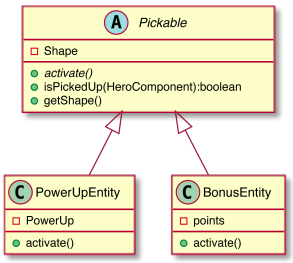


Figura 2.4: Rappresentazione UML di Template Method per PowerUp e Bonus

Problema: l'applicativo Pang richiede la generazione casuale di PowerUp per modifcare lo stato delle sfere/arpioni, e di Bonus per incrementare il punteggio del giocatore. Durante lo sviluppo, ci siamo accorti che queste entità condividevano molto del loro comportamento e struttura interna e stavano portando ad avere codice duplicato o ripetitivo.

Soluzione: Siccome le due classi differiscono solo per qualche comportamento, tramite il pattern *Template Method*, è stato possibile definire il tipo `Pickable`, il quale definisce le azioni e i comportamenti basilari di un item, per poi essere specializzato in base alle esigenze di un item.

Gestione dei diversi PowerUp ed estendibilità

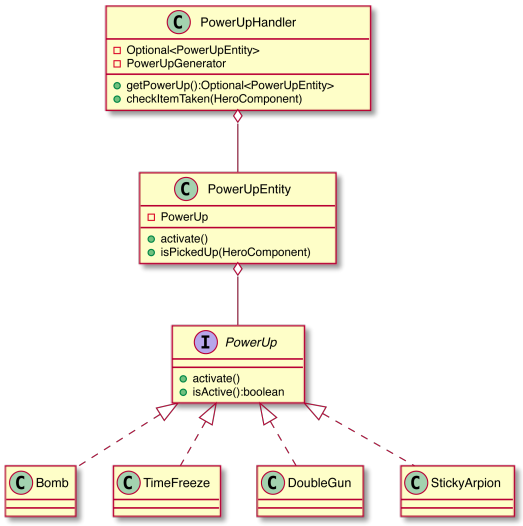


Figura 2.5: Rappresentazione UML della gestione di diversi PowerUp

Problema: l'applicazione deve essere in grado di fornire un certo numero di `PowerUp`, ognuno con effetti sensibilmente diversi, ed essi devono essere gestiti da un *controller* il quale deve generare un `PowerUp` casuale ogni X secondi. Durante lo sviluppo, ci siamo accorti che l'istanziazione singolare di ogni specifico powerUp, avrebbe causato una forte rigidità del codice e reso l'estendibilità molto complicata (se ci fosse un nuovo powerUp, bisognorebbe modificare molte classi).

Soluzione: Ogni `PowerUp` per quanto diverso dagli altri, è possibile incapsulare il suo tipo tramite un interfaccia, in modo tale che più ad alto livello, non interessa la peculiarità di quel particolare `PowerUp`, ma esso deve solo essere **attivabile**. In questo modo il controller `PowerUpHandler` mantiene un riferimento all'entità del `PowerUp` (`PowerUpEntity` composto da un `PowerUp` e una `Shape` vedi Figura 2.4) e rende possibile al client usufruire di un controller che gestisce tutti i powerUp in modo trasparente, e inoltre rende l'inserimento di un nuovo powerUp molto semplice, in quanto basta implementare l'interfaccia powerUp.

2.2.2 Michele Montesi

Gestione della Pausa

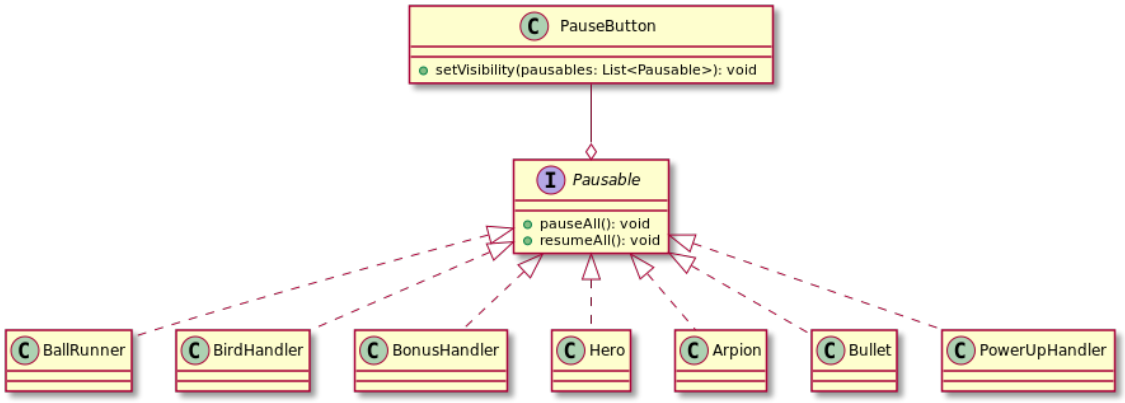


Figura 2.6: Rappresentazione UML della gestione della pausa

Problema: Il gioco deve supportare la pausa di oggetti multipli. Durante lo sviluppo ci siamo accorti che la funzione di pausa richiamava le stesse funzioni per tutti gli oggetti.

Soluzione: Dato che i vari oggetti devono essere chiamati da un gestore della pausa, si è deciso di racchiudere questi oggetti dietro l'interfaccia `Pausable`. In questo modo è possibile, per il gestore, chiamare una funzione di pausa, ed una di avvio, su una lista di `Pausable` e quindi senza ripetizioni di codice.

HUD dinamico

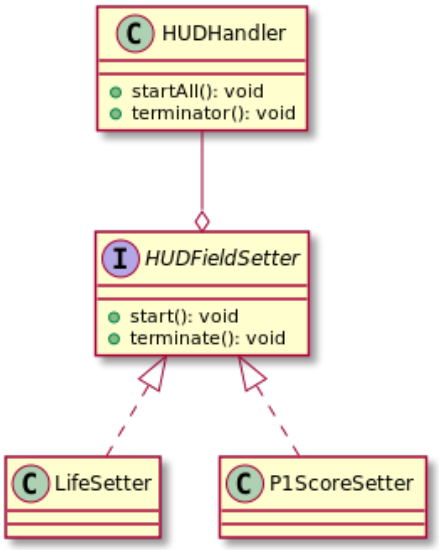


Figura 2.7: Rappresentazione UML della gestione dell'HUD dinamico

Problema: Il campo `Score` dell'HUD deve aggiornarsi quando il punteggio viene incrementato, similmente le vite.

Soluzione: Sono state create due classi, chiamate da un gestore dedicato, improntate alla visualizzazione dello stato delle variabili in questione. Queste classi, inoltre, sono state racchiuse dentro l'interfaccia `HUDFieldSetter` in quanto utilizzano gli stessi metodi. Quando lo stato delle variabili cambia, i campi interessati dell'HUD vengono così aggiornati.

Scelta di una direzione unica fino alla morte dell'uccello

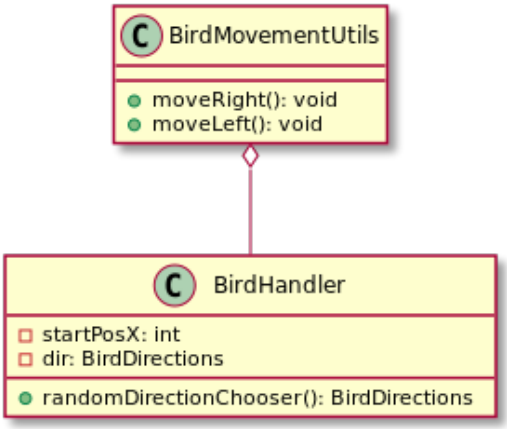


Figura 2.8: Rappresentazione UML della gestione della scelta di una direzione

Problema: La direzione dell'uccello viene generata casualmente, ma una volta generata non deve cambiare in base alla sua posizione fino alla fine del suo ciclo di vita.

Soluzione: Durante l'implementazione delle funzioni di movimento abbiamo notato che la direzione dell'uccello cambiava in base alla sua posizione. È stata creata una classe `BirdHandler` la quale gestisce il ciclo di vita dell'uccello, compresa la creazione e la distruzione. Alla creazione viene definita una posizione iniziale decisa dalla direzione precedentemente generata. In base a questa, viene chiamato un metodo esterno dedicato al movimento dell'uccello nella direzione desiderata, in questo modo la direzione non potrà cambiare in base alla posizione.

Muovere l'uccello in base alla direzione scelta

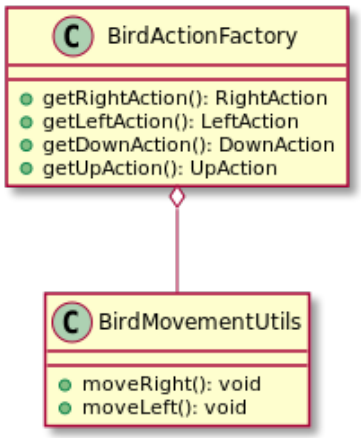


Figura 2.9: Rappresentazione UML della gestione del movimento dell'uccello

Problema: L'uccello, ad ogni movimento, deve cambiare posizione in base alla direzione generata randomicamente in precedenza.

Soluzione: È stata creata una factory di cambio posizione dell'uccello in modo che, per ogni cambio posizione, venga chiamata uno dei metodi di movimento. Grazie a questo il codice è piú pulito.

2.2.3 Luca Rapolla

Portabilita' del personaggio principale (e della sua relativa arma) su un nuovo motore grafico

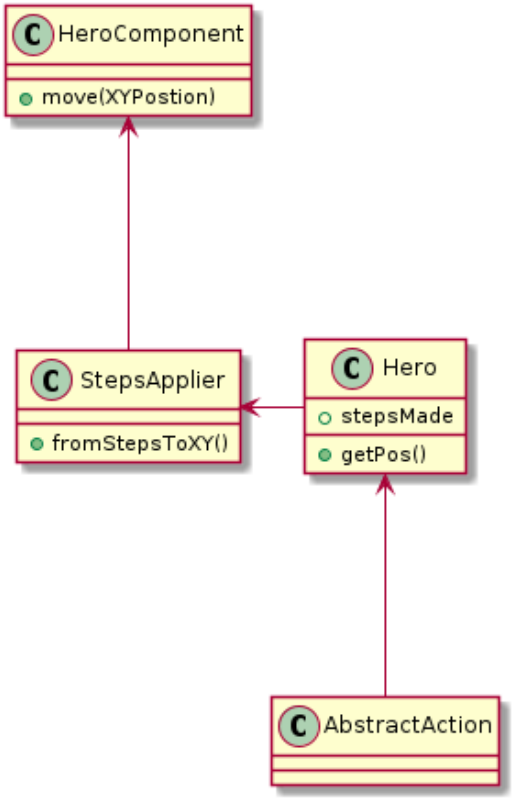


Figura 2.10: Rappresentazione UML della divisione tra Model e Visual

Problema: Il personaggio principale ha bisogno di muoversi all'interno di uno spazio il quale è determinato dal motore grafico che è possibile cambiare in base alle necessitá.

Soluzione: Il personaggio principale e la sua arma sono divisi in *Hero*, *HeroComponent* ed *Arpion*, *ArpionComponent*. Sia *Hero* che *Arpion* mantengono le loro posizione assolute a prescindere dalle dimensioni della finestra di gioco e dal motore grafico usato. Nel momento in cui si necessita di mettere in relazione le due entitá si usa un convertitore *StepsConverter* che converte i passi fatti da *Hero* in una posizione all'interno della finestra. Stessa cosa vale per *Arpion* il quale, per salire verso l'alto, esegue un determinato numero di passi verso l'alto.

Eseguire azioni tramite la pressione di tasti (*KeyBindings*)

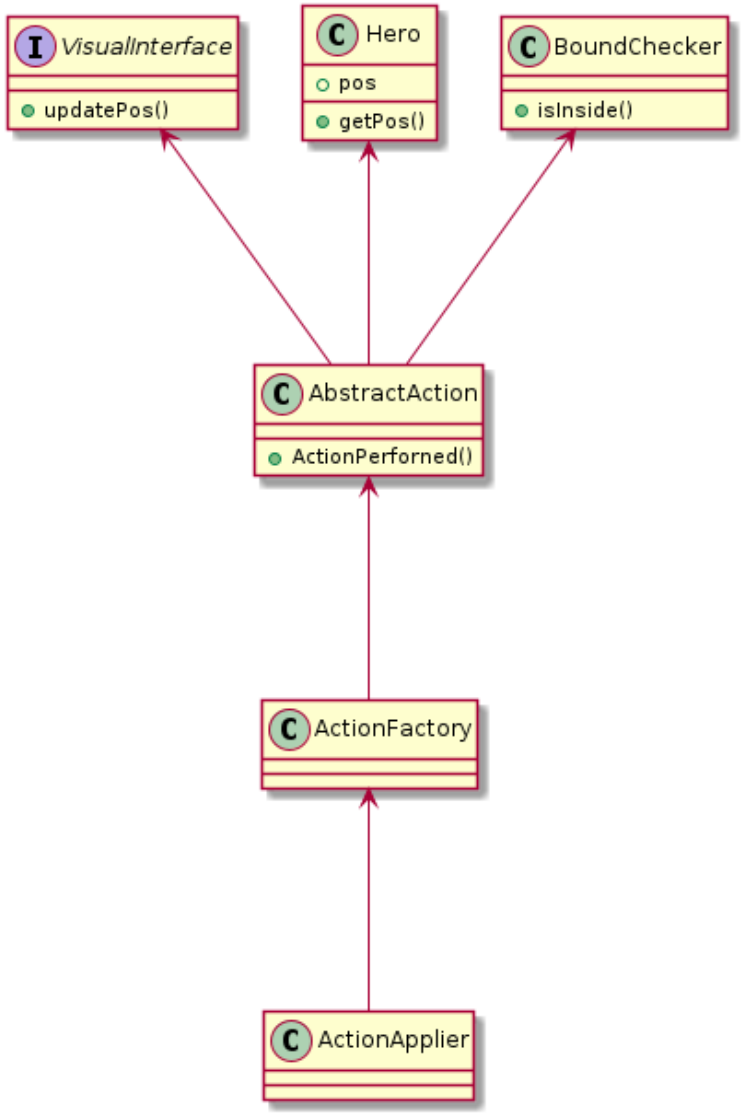


Figura 2.11: Rappresentazione UML del processo di avvenuta azione dopo la pressione di uno specifico tasto

Problema: Il giocatore deve poter comandare il personaggio principale tramite input da tastiera.

Soluzione: Per rendere il personaggio principale meno dipendente possibile dall'interfaccia di gioco si e' deciso di creare un sistema il quale collegando i tasti di gioco al pannello, questi richiamano semplici azioni che avranno effetto direttamente sulle caratteristiche del personaggio principale, al quale si chiede solo di cambiare i suoi attributi.(e.g. Volendo fare un passo verso sinistra, viene prima ricavata la posizione assoluta del personaggio principale da *Hero* tramite *StepsApplier*, in seguito viene fatto un controllo sulla prossima posizione in cui il personaggio principale dovra' spostarsi, se il controllo passa viene aggiornato *Hero* e di conseguenza *HeroComponent*.

Collisione del personaggio principale con un'entità ostile

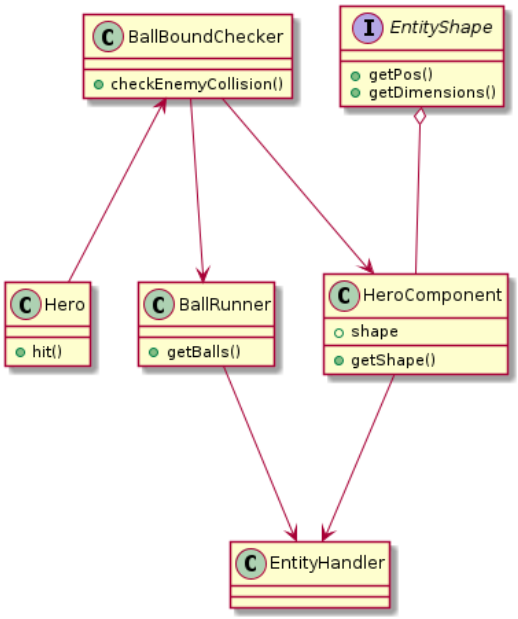


Figura 2.12: Rappresentazione UML del processo di rilevamento di una collisione ed applicazione degli effetti relativi all'evento

Problema: Gestire l'evento di una collisione tra entità ed apllicarne gli effetti agli attori coinvolti.

Soluzione: La riflessione di *Hero* sulla finestra di gioco, ovvero *HeroComponent*, utilizza un'interfaccia *EntityShape* la quale tiene traccia delle dimensioni e della posizione di un'entità nello spazio. Utilizzando un controller, il cui unico scopo è controllare le possibili collisioni tra le entità, si mette in relazione la *Shape* del personaggio principale con la *Shape* delle sfere. In caso di avvenuta collisione, si comunica l'evento ad *Hero* il quale procederà a diminuire le vite disponibili e procederà a far duplicare la sfera che ha colpito il personaggio principale. La gestione dell'evento è equivalente nel caso si collida con l'entità uccello, l'unica differenza è che il nemico muore invece che duplicarsi.

Armi del personaggio principale

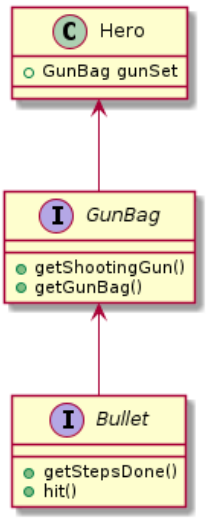


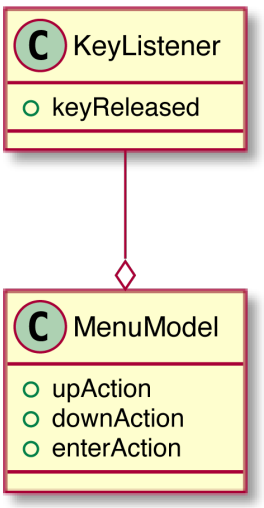
Figura 2.13: Rappresentazione UML della modellizzazione dell'arma data in dotazione al personaggio principale

Problema: Assegnare al personaggio un arma capace di sparare diversi proiettili.

Soluzione: Il personaggio principale e' dotato di un'arma *GunBag* la quale puo' sparare dei proiettili, *Bullet*. Il punto forte di questa progettazione e' l'alto potenziale di estendibilita' delle interfacce *Bullet* e *GunBag*, in questo modo e' possibile assegnare al personaggio principale un altra tipologia d'arma semplicemente implementando le interfacce sopra citate. L'unico punto debole di questa implementazione e' l'utilizzo di una `enum` per modellizzare le tipologie dei *powerups* il che rende *Arpion* non estendibile.

2.2.4 Ezmiron Deniuku

Input Utente



Problema: la classe *MenuModel* deve riuscire a prendere in input i tasti premuti dall'utente.

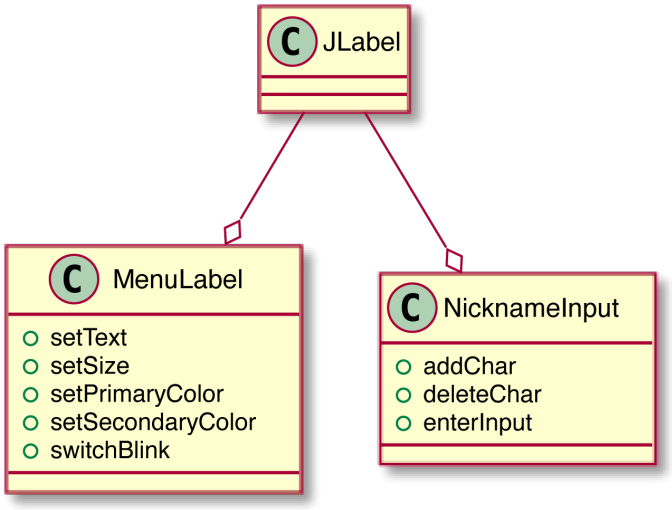
Soluzione: *MenuModel* estende *KeyListener* di cui eredità il metedo astratto keyReleased e sviluppa i suoi metodi.

Etichette Dinamiche

Problema: necessità di avere delle etichette dinamiche per far capire quale voce è selezionata

Soluzione: utilizzo della classe *MenuLabel* che permette di settare colori, dimensioni, testi e rendere le scritte dinamiche tramite il metodo `switchBlink()`. Gli altri metodi servono per realizzare etichette generiche come "PANG"

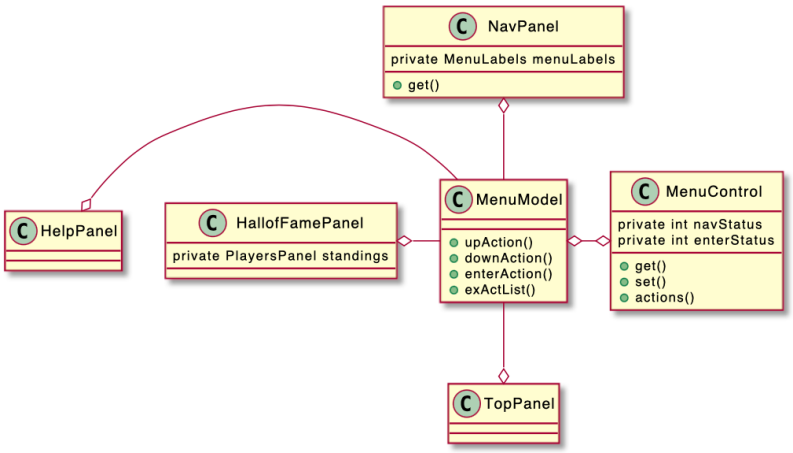
Visualizzazione di caratteri digitati



Problema: avere un modo di far visualizzare i caratteri che l'utente inserisce per il nome

Soluzione: utilizzo *NicknameInput* per creare l'etichetta dell'inserimento nel nome ("MARCO12") che tramite `deleteChar()` o `addChar()` riesce a aggiornare di volta in volta. Il metodo `enterInput()` servirà al momento del lancio del gioco.

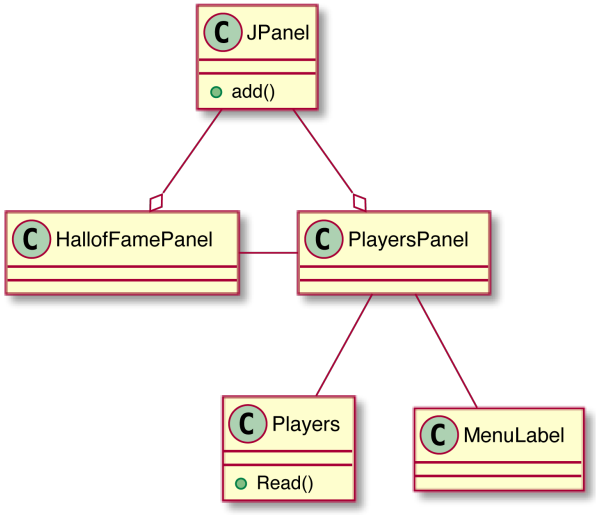
Menu Interattivo



Problema: cambiare i pannelli in base alla propria selezione. e.g.se l'utente seleziona la voce "HALL OF FAME", visualizzare la hall of fame del gioco.

Soluzione: MenuModel prende gli input dell'utente e tramite i suoi metodi `upAction()`, `downAction()`, `enterAction()` li manda a *MenuControll*, la quale tiene conto di tutti gli input già avvenuti(tramite *navStatus* e *enterStatus*) ed elabora in seguito una nuova azione da eseguire e la restituisce a *MenuModel*, che tramite `exActList()` agisce sulla gui andando a modificare i componenti di questa(le varie classi Panel).

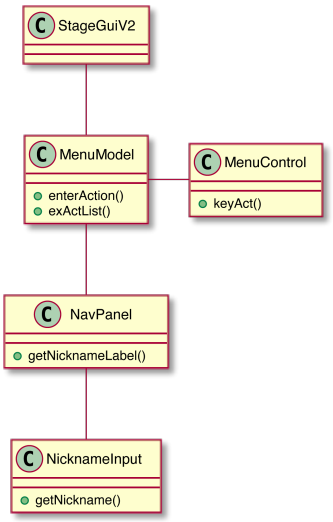
Salvataggio su file della *Leaderboard*



Problema: Riuscire a creare il panello con la hall of fame con tutti i giocatori che si riescono a estrarre dal metodo `Read()` della classe *Players*.

Soluzione: La classe *HallOfFamePanel* estende *JPanel* e col suo metodo `add()` aggiunge a sé PlayersPanel che anche essa ha esteso *JPanel* ed a sua volta ha aggiunto i dati dei giocatori tramite il metodo sotto forma di *MenuLabel*.

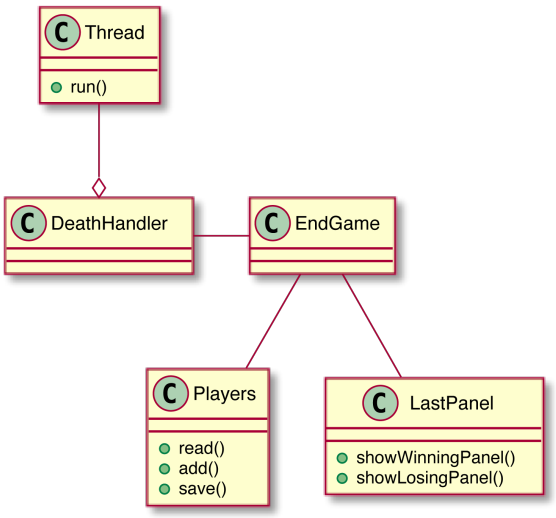
Avvio sessione di gioco



Problema: avviare il gioco quando l'utente inserisce il nickname e preme ENTER

Soluzione: MenuModel riceve l'input dall'utente e tramire `enterAction()` lo manda a *MenuControl*, quest'ultimo elabora l'input e col metedo *keyAct* lo manda a *MenuModel* che col metodo `exActList()` avvia `StageGuiV2` che si prende come parametro un *Player* che viene generato (`NavPanel.getNicknameLabel().getNickname()`) sul momento prendendo il nome che l'utente aveva scritto fino a quel momento.

Schermata di Fine Gioco e Salvataggio



Problema: salvare i dati del giocatore in caso di vittoria e mostrare il pannello finale

Soluzione: La classe DeathHandler che estende Thread verifica nel metodo `run()` lo stato di vittoria o sconfitta e passa l'informazione a EndGame che in base a questo chiama i metodi di LastPanel facendo visualizzare il pannello finale. In caso di vittoria EndGame chiama i metodi di Players per aggiungere le stastische dell'ultimo giocatore e salvarle in caso di score rientrante in top 10.

Capitolo 3

Sviluppo

3.1 Testing Automatizzato

Il testing è stato svolto principalmente in maniera "visiva", ovvero che in base al comportamento di un entità, abbiamo stabilito se il funzionamento era corretto o meno.

Il testing non è avvenuto in modo "automatico". Per le sfere, il testing è avvenuto attraverso l'interfaccia grafica e specialmente in fase iniziale, è stato necessario verificare che le formule utilizzate per emulare il moto parabolico della sfera avessero un riscontro visivo. Sempre per quest'ultimo motivo, i test riguardo la duplicazione e lo stop delle sfere sono sempre avvenuti attraverso interfaccia grafica e un pannello di controllo con dei bottoni, in modo tale da verificare il relativo effetto. Riguardo al pannello di controllo, nelle versioni iniziali era stato inserito nell'Action Listener dei bottoni, l'istanziazione del PowerUp Freeze e Bomb, ma con l'aggiornamento delle classi dei powerUp quei test sono risultati obsoleti (ora sono chiamati direttamente i metodi *Duplicate* e *Pause*). Allo stesso modo "visivo", sono avvenuti i test riguardo a tutte le entità che possono collidere tra di loro, in particolare: sfere-arpioni-eroe, uccello-arpioni-eroe ed eroe con bonus e powerUp.

Per quanto riguarda il testing del corretto funzionamento dell'uccello è stato necessario tenere il testing manuale in quanto la direzione iniziale è randomica e in quanto quando colpisce l'Hero non deve scomparire mentre quando viene colpito da un arpione deve essere distrutto. È stato creato un frame di test apposito.

Per quanto riguarda il personaggio principale, sono stati svolti test principalmente in maniera "visiva" seguendo passo passo lo sviluppo del personaggio e della sua arma. I pirmi test svolti erano effettuati sulla corretta gestione dei movimenti comandati da *input* da tastiera. Con l'aggiunta dei

powerups e dell'Arpione, e' stato costruito un controller ad hoc il quale agisce direttamente sulle caratteristiche degli attori in gioco in modo da simulare gli effetti dei *powerups*. Per quanto riguarda le interazioni con gli altri attori (Uccello e Sfere) sono stati usati dei *log* appositi in grado di farci capire il modo in cui gli eventi si manifestavano.

Per quanto riguarda il testing relativo alla *Hall of Fame*, si e' creata una finta entita' `Player` in modo da testare l'avvenuto salvataggio.

3.2 Metodologia di lavoro

Michele Montesi

- Creazione dell'uccello come nemico aggiuntivo all'interno del gioco.
- Creazione dello stage di gioco completo di HUD dinamico (spiegato nel capitolo precedente).
- Creazione di utility di sistema come *MainImagesLoader* e *MainFontLoader*, ovvero dei loader generici usati poi da loader più specifici.
- Creazione di utility di sistema per acquisire altezza e larghezza dello schermo principale.
- Creazione del menu di pausa e del suo funzionamento. Assieme a questo viene associata una utility che permette di uscire dal gioco con la pressione di un tasto.

Le parti di codice sono state sviluppate prima come stand-alone objects in modo da poter eseguire testing sufficienti. Una volta pronte sono state integrate con il resto del codice facendo il refactoring di tutto il necessario. Sicuramente quest'ultima parte è stata la piú difficile in quanto gestita male durante la fase di design. L'utilizzo del *DVCS* ha ricoperto un vantaggio fondamentale sia per quanto riguarda la collaborazione, sia per la cronologia del codice. É stato usato su tre *Branch* principali:

- `Main`: *Branch* principale, nel quale abbiamo pubblicato *release* completamente funzionanti.
- `Feature`: *Branch* di lavoro, qui e' stato effettuato la gran parte del lavoro sul progetto.
- `Feature Safe`: *Branch* di lavoro sul quale poter testare parti di codice che possano temporaneamente sporcare il codice, in quanto questo *Branch* é di test.

Stefano Furi

I seguenti aspetti sono quelli su cui è stato effettuato un lavoro individuale.

- Progettazione, modellazione e implementazione delle sfere (package `ball`) sia negli aspetti di model e controller sia dal punto di vista di view.
- Progettazione di uno scheletro di controller in grado di gestire entità autonomamente, e riferire cambiamenti di stato alla view (vedere come riferimento classe `Visualiser` in `ball.gui`).
- Progettazione dei `PowerUp` e i `Pickable`.
- Implementazione dei `PowerUp Time Freeze` e `Bomb`.
- Implementazione del sotto-controller `PickableHandler`.

Aspetti su cui è stato effettuato un lavoro collettivo:

- Progettazione e Sviluppo di `PowerUpEntity` e `PowerUpHandler` in collaborazione con Rapolla.
- Sviluppo su scheletro sopracitato della classe `EntityHandler` assieme a Rapolla e Montesi.
- Modellazione del concetto di `Pausable` assieme a Montesi.

Lo sviluppo della propria sottosezione (che comprendeva sia la parte di model sia di controller e view) è avvenuto completamente autonomamente. Il metodo di integrazione è stato quello di definire in fase di design, degli *eventi* a cui il proprio dominio dovesse reagire (e.g. una Sfera che colpisce un arpione scaturisce una duplicazione). Nonostante questo, in fase di integrazione, abbiamo riscontrato difficoltà nel far comunicare le varie entità, ed è stato necessario un lavoro di riadattamento per quasi ognuna di esse.

Per quanto riguarda l'utilizzo del DVCS, è stato usato localmente un branch denominato *develop* per modificare/migliorare elementi costitutivi del modello mantenendo sempre una versione "funzionante" nel branch principale di sviluppo in caso di errori irrimediabili. Similmente, il branch remoto *feature_safe* è stato utilizzato da tutti i membri del gruppo durante la fase di integrazione, affinchè non venisse intaccata da possibili problemi la versione corrente funzionante nel branch principale di sviluppo *feature*. Infine, nel branch *main*, oltre alla preparazione iniziale della repository, sono state aggiunte solo versioni completamente funzionanti dell'applicazione.

Luca Rapolla

I seguenti aspetti sono stati svolti in maniera individuale.

- Progettazione, modellizzazione ed implementazione del personaggio principale (package `PangGuy`) negli aspetti *model*, *view*, *controller*.
- Progettazione, modellizzazione ed implementazione dei *Keybindings* (package `PangGuy`).
- Progettazione e modellizzazione dei *Bonus* (Package `Bonus`).
- Progettazione, modellizzazione ed implementazione dell'arma ed i suoi proiettili (Package `PangGuy`).
- Progettazione, modellizzazione ed implementazione dei *powerups* `DoubleArpion` e `StickyArpion` (Package `PowerUp`).

Parti sviluppate in gruppo:

- Progettazione e sviluppo di *PowerupEntity* e *PowerupHandler* con Furi.
- Progettazione e sviluppo di *DeathHandler* con Deniku.
- Implementazione dei *Bonus* con Deniku.

Lo sviluppo delle varie parti del codice e'stato svolto singolarmente. Causa l'inseperienza nel prendere parte in lavori di gruppo, abbiamo verificato, soprattutto in caso di unione del personaggio principale con le sfere, la difficolta' nel relazionare efficacemente le due parti in quanto abbiamo constatato di esserci impegnati piu' sulla praticita' di una classe che alla fusione con altre classi.

Per quanto riguarda l'utilizzo del *DVCS*, e' stato usato su tre *Branch* principali:

- `Main`: *Branch* principale, nel quale abbiamo pubblicato *release* completamente funzionanti del gioco in modo da avere sempre una versione pronta e funzionante.
- `Feature`: *Branch* di lavoro, qui e' stato effettuato la gran parte del lavoro sul progetto.
- `Feature Safe`: *Branch* di lavoro secondario, usato principalmente in casi di grosse modifiche che avrebbero potuto influire negativamente sull'integrita' del progetto.

Ezmiron Deniku

•

Arrivando dalla programmazione ad oggetti fatta in *C*, ho dovuto reinventarmi completamente, e queste inizialmente lo ho fatto riguardando le slide delle lezione da me saltate poiché ero in *Erasmus*, cercando di capire non solo questo nuovo linguaggio, ma anche cercando di unire a quanto ho imparato nella università ospitante.

Il grosso del mio lavoro si basava sulla parte grafica, cosa da me esplicitamente richiesta in fase iniziale, poiché come dissi per email al prof. Viroli in lituania non mi avevo mai visto come lavorare a parti grafiche. Inizialmente ho cercato di prendere spunto dall'operato Michele Montesi che doveva creare il pannello basso nel gioco, ma ovviamente questo non è bastato poiché la mia gui doveva essere dinamica e quindi ho fatto qualche ricerca su stackoverflow e ho letto documentazioni di librerie come GridBagLayout e GridBagConstraints.

3.3 Note di sviluppo

Michele Montesi

- Utilizzo di lambda nel metodo *setVisibility()* di `pauseMenu.components.PauseButton`.
- Uso di $Optional < BirdActor >$ in `bird.controller.BirdHandler`.
- Uso di $Optional < BirdShape >$ in `bird.controller.BirdHandler`.
- Caching degli sprite per l'uccello in `bird.utilities.BirdPNGLoader`.

- **Risorse**
  - Aggiungere componenti in una specifica posizione di un JPabel: `https://stackoverflow.com/questions/2510159/can-i-add-a-component-to-a-specific-grid-cell-when-a-gridlayout-is-used`
  - Rimuovere alone attorno ad uno sprite: `https://stackoverflow.com/questions/71797654/bufferedimage-drawn-on-jlabel-doesn-keep-transparent-background`

Stefano Furi

- Utilizzo dell'interfaccia funzionale `Function` in `ball.physics.Trajectory` nel metodo `relativeSpeed`.
- Utilizzo di generici nella classe astratta `utilities.Pos2D`.
- Utilizzo di Stream in `EntityHandler` e `PickableHandler` nei motodi `getPausable()` e un simile utilizzo è presente in `Controller`.

Porzioni di codice reperite in rete:

- Moto del Proiettile: `https://www.101computing.net/projectile-motion-formula/`
- Snippet intersezione tra Rettangolo e Cerchio: `https://stackoverflow.com/questions/401847/circle-rectangle-collision-detection-intersection`
- Snippet sovrapposizione tra Rettangoli: `https://stackoverflow.com/questions/23302698/java-check-if-two-rectangles-overlap-at-any-point`

Luca Rapolla

- Uso di Optional nello sviluppo dell'arma in `pangGuy.modularGun`, in `bonus.BonusHandler.java`, in `powerup.PowerUpHandler.java`
- Uso di *Generics* in `FullPair.java`
- Uso di una Factory in `panguy.actions.ActionFactory.java`

Porzioni di codice reperite in rete:

- KeyBindings (Ho preso solo spunto per la funzione da usare, il codice in se era di male fattura): `https://www.youtube.com/watch?v=IyfB0u9g2x0&ab_channel=BroCode`

Ezmiron Deniku

Quando andavo a salvare e leggere le prime volte cercavo di tenere un file di testo ("bestPlayersSaves.txt") nella cartella delle risorse e per poi sovrascriverlo sempre lì.

Per cercare il filepath utilizzavo ClassLoader.getSystemResourceAsStream(), andando poi a scrivere con getResoursce(dandogli il classhpath), ed il tutto

funzionava finchè si lanciava da eclipse, ma poi andando a lanciare da jar non funzionava.

Su siti come stackoverflow ho trovato che dovrebbe essere impossibile creare file all'interno di un jar.

Quindi ho ovviato al problema, con l'auito di Michele Montesi, andando a creare il file di salvataggio fuori dal jar nella cartella app del progetto.

Capitolo 4

Commenti Finali

4.1 Autovalutazione e lavori futuri

Michele Montesi

Le ore ricoperte sono bastate per completare le sezioni di progetto a me attribuite. Non é stato attribuito tempo sufficiente al design, il quale ha portato a diversi problemi nella rifattorizzazione di vari componenti con quelli dei colleghi.

Punti di forza

- Il menu e la funzione di pausa funzionano correttamente.
- L'HUD si aggiorna automaticamente.
- L'uccello si integra con l'ambiente di gioco senza creare conflitti di alcun tipo.
- Riuso del codice

Punti di debolezza

- Lo stage non viene generato sempre in 16:9 ma prende le dimensioni in base all'altezza e alla larghezza dello schermo.
- Scarsa comunicazione nelle prime fasi, la quale ha causato riprogettazione e refattorizzazione del codice.
- Possibile utilizzo di troppi thread.

Il ruolo che ho ricoperto consiste nella creazione del nemico `Bird` e delle sue funzionalità, nonché alla creazione di varie utility di sistema per lo stage. È stato anche creato il menu e la funzione di pausa e di uscita dal gioco mentre in pausa. Mi sono sempre reso disponibile per controllare il codice dei miei colleghi ed aiutarli a risolvere eventuali problemi.

Stefano Furi

La sottoparte del modello dell'applicativo assegnata, è risultata sufficiente per ricoprire le ore di lavoro richieste dai docenti. La fase di progettazione individuale è stata più che altro funzionale (come gestire il tutto e integrarlo con gli altri elementi) e non è stato attribuito abbastanza tempo al design. Infatti, per porzioni più corpose come la sfera, non sono praticamente presenti *design patterns* ma solo soluzioni più o meno efficaci in base al problema riscontrato. D'altro canto, per la modellazione dei PowerUp, è stata attribuita più attenzione al design, in quanto già in fase di analisi con i colleghi, venivano fuori molte problematiche legate alla proliferazione di istanze molto simili ma che differivano di poco nel comportamento. Riassumendo quindi:

- PowerUp estendibili e molto riuso di codice.
- Diversi livelli di controllo e di astrazione per le sfere, rendendo comprensibile al client le funzioni e potenzialità delle sfere.
- Scorporamento del controller principale in sotto controller, in modo tale da delegare a ognuno di essi una sottoparte del dominio applicativo e rendere più semplice gestire eventuali errori.

D'altra parte, i punti di debolezza principali:

- Insufficiente progettazione individuale del design.
- Scarsa considerazione degli aspetti necessari per l'integrazione di tutti i componenti del sistema, la quale ha causato molto lavoro di riprogettazione e adattamento.
- Possibile uso improprio delle funzionalità dei `Thread`.
- Poca considerazione delle performance.

All'interno del gruppo di lavoro, in quanto avendo iniziato e "finito" per primo la parte di modello, mi sono reso disponibile nell'aiutare i miei compagni nell'interfacciamento delle loro classi con il controller principale. Inoltre, ho aiutato e collaborato con i miei colleghi nella creazione e corretta implementazione delle entità in grado di gestire l'integrazione di tutti gli elelemti del dominio (i sotto-controller) durante la fase finale di sviluppo.

Luca Rapolla

La sottoparte del software realizzata e' risultata sufficente sia in ambito di difficolta' che in ambito di tempo utilizzato per produrla rispetto alle ore di lavoro richieste. Un punto molto forte del lavoro di gruppo, secondo il mio parere, e' stato quello di conoscerci tutti di persona, questo ha abbattuto la barriera dell'imbarazzo verso uno sconosciuto e ci ha dato la possibilita' di essere piu aperti e sinceri verso gli altri componenti del gruppo. A mio avviso l'uso del *DVCS* poteva essere perfezionato ma, essendo la prima esperienza "sul campo", comprendo i nostri limiti. Per quanto riguarda il codice invece, posso delineare i seguenti lati positivi:

- Il dialogo ben riuscito tra personaggio principale e le sue armi.
- La semplice assegnazione delle immagini dei *Bonus* tramite i loro punti.
- Il dialogo tra le sfere e l'arpione.

Per quanto riguarda i lati negativi invece:

- I troppi pochi *powerups* creati.
- La richiesta modestamente alta di CPU causata dai Thread.

In futuro vorrei aggiungere piu armi al personaggio principale e creare piu' *Powerups*.

Appendices

Appendice A

Guida utente

Utilizzo della *LeaderBoard*

- All'avvio del gioco viene controllato se il file di salvataggio della *LeaderBoard* si trova nella stessa directory del JAR. Nel caso in cui non ci sia viene creato.

Controlli

- `Frecce direzionali:` Movimento a destra e sinistra del personaggio principale.
- `Barra spaziatrice:` Utilizzo dell'arma attuale.

Appendice B

Esercitazioni di laboratorio

Michele Montesi

- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=87881#p138074
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=87880#p138072
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=88829#p138076
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=89272#p138077
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p138555

Stefano Furi

- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=87880#p138782
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=88829#p138781
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=89272#p138780
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p138779

Luca Rapolla

- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p138405
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=89272#p138048
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=88829#p136506
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=87880#p136596
- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=87881#p138042