Relazione Progetto
Bullet Ballet

Alessandro Pioggia, Leon Baiocchi, Federico Brunelli, Luca Rengo

Agosto | Settembre | Ottobre 2021

Indice

Sommario

Questo documento è una relazione di meta livello, ossia una relazione che spiega come scrivere la relazione. Lo scopo di questo documento è quello di aiutare gli studenti a comprendere quali punti trattare nella loro relazione, ed in che modo farlo, evitando di perdere del tempo prezioso in prolisse discussioni di aspetti marginali tralasciando invece aspetti di maggior rilievo. Per ciascuna delle sezioni del documento sarà fornita una descrizione di ciò che ci si aspetta venga prodotto dal team di sviluppo, assieme ad un elenco (per forza di cose non esaustivo) di elementi che *non* dovrebbero essere inclusi.

Il modello della relazione segue il processo tradizionale di ingegneria del software fase per fase (in maniera ovviamente semplificata). La struttura della relazione non è indicativa ma *obbligatoria*. Gli studenti dovranno produrre un documento che abbia la medesima struttura, non saranno accettati progetti la cui relazione non risponda al requisito suddetto. Lo studente attento dovrebbe sforzarsi di seguire le tappe suggerite in questa relazione anche per l'effettivo sviluppo del progetto: oltre ad una considerevole semplificazione del processo di redazione di questo documento, infatti, il gruppo beneficerà di un processo di sviluppo più solido e collaudato, di tipo top-down.

La meta-relazione verrà fornita corredata di un template LaTeX per coloro che volessero cimentarsi nell'uso. L'uso di LaTeX è vantaggioso per chi ama l'approccio "what you mean is what you get", ossia voglia disaccoppiare il contenuto dall'effettivo rendering del documento, accollando al motore LaTeX l'onere di produrre un documento gradevole con la struttura ed il contenuto forniti. Chi non volesse installare l'ambiente di compilazione in locale può valutare l'utilizzo dell'applicazione web Overleaf. L'eventuale utilizzo di LaTeX non è fra i requisiti, non è parte del corso di Programmazione ad Oggetti, e non sarà ovviamente valutato. I docenti accetteranno qualunque relazione in formato standard Portable Document Format (pdf), indipendentemente dal software con cui tale documento sarà redatto.

Capitolo 1

Analisi

Bullet Ballet è un videogioco platform a scorrimento orizzontale, del genere *Shoot 'em up* in 2D.
L'obiettivo del gioco è quello di realizzare più punti possibili per superare i propri record, in base alla distanza percorsa dal giocatore.

Ma non sarà tutto in discesa, il giocatore dovrà affrontare nemici con i più variegati equipaggiamenti, ostacoli di ogni sorta, varchi nella mappa e molto altro ancora..
Il giocatore potrà scegliere fra ben 8 mappe uniche e relative piattaforme in tema.

Inoltre, il giocatore potrà avvalersi a sua volta di effetti (bonus) sia positivi che negativi per poter sconfiggere le avversità che si presenteranno sul suo cammino.

Potrà, poi salvare tutti i suoi punteggi di gioco.

1.1 Requisiti

L'applicazione mostra, all'avvio, un menù di gioco con le seguenti voci: *New Game*, *Load Game*, *Settings*,*Quit*.

Requisiti funzionali

- *Mappa scorrevole*: Il giocatore potrà muoversi in una mappa a scorrimento orizzontale con uno sfondo statico.
- *Menù di gioco*: Non appena lanciata l'applicazione, l'utente vedrà un menù di gioco con diverse opzioni tra cui potrà scegliere.

- *Menù di pausa*: Il giocatore potrà mettere il gioco in pausa attraverso un dato pulsante della tastiera, scelto nelle impostazioni.
- *Personaggio*: Il giocatore potrà scegliere un personaggio e muoverlo in partita.
- *Nemici*: Verranno generati diversi nemici che il giocatore dovrà affrontare.
- *Ostacoli e oggetti raccoglibili*: Verranno creati nella mappa degli ostacoli che bloccheranno la strada e degli oggetti che il giocatore potrà raccogliere per ricevere un (power up) bonus od un malus.
- *Salvataggio del punteggio e classifica*: Terminata la partita, il punteggio di gioco potrà essere salvato e verrà generata una classifica finale.
- *Suoni ed effetti sonori*: La durata della partita verrà accompagnata da una colonna sonora e agli effetti sonori prodotti dall'environment di gioco.
- *Fisica di gioco*: Tutti le entità di gioco saranno dotati di una fisica ed una gravità propria.

1.1.1 Requisiti opzionali

Questi requisiti non concorrono a far parte delle funzionalità minime del gioco e per questioni di tempistica e/o di budget non verranno necessariamente implementate.

- *Oggetti dinamici*: ovvero oggetti che si muovono e che hanno una animazione.
- *Market*: per poter comprare/vendere skin (mimetiche) di gioco con valuta reale o in gioco. (o fittizia del gioco)
- *Modalità storia*: una modalità con una storia di gioco e diversi livelli che il giocatore dovrà sbloccare per poter continuare ad avanzare nel gioco.
- *Statistiche di gioco*: Varie statistiche di gioco, mostrate in diversi diagrammi.
- *Difficoltà di gioco*: Possibilità di scegliere diverse difficoltà che potrebbero aggiungere numero di nemici/ostacoli/oggetti di varia natura e/o incrementare la loro vita.

1.2 Analisi e modello del dominio

L'applicazione fornisce un giocatore che tramite input dell'utente può essere spostato a destra, a sinistra e farlo saltare. Il quale, potrà interagire con le varie entità, sia statiche che dinamiche del gioco, quali *nemici*, *armi*, *monete*, *items*, *ostacoli*. Ognuna di queste entità interagirà in maniere differenti col player:

- *Enemy*, ostacolerà il player a colpi di mitra.
- *Item*, una volta raccolta dal player, gli fornirà o un **bonus**, come una vita extra oppure un **malus** come un effetto di volevo per tot tempo.
- *monete*, faranno aumentare il punteggio e il numero di monete del gioco da usare nel mercato per poter comprare nuove skins, in game-items, ecc..
- *Armi*, dopo essere state raccolte verranno equipaggiate al giocatore.
- *Ostacoli*, se il player ci collliderà, subirà del danno.

Il gioco si conclude quando e se il player riesce a raggiungere la fine della mappa senza morire. Se il player viene ucciso prima di raggiungere la fine, allora viene eliminato e la partita è persa.

Ogni round è a sè stante, finita la partita, al player verrà assegnato un punteggio e potrà scegliere se rigiocare od uscire.

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

Capitolo 2

Design

2.1 Architettura

L'obiettivo del team era quello di poter lavorare ognuno alla propria parte indipendentemente ed evitando conflitti, per poi poterle collegare tutte assieme alla fine.

Il progetto sfrutta quindi il pattern architetturale **MVC** (*M*odel *V*iew *C*ontroller) che permette di suddividere la gestione dell'applicativo in tre parti separate:

- **Model**: dove vengono effettivamente modellate le entità di gioco. In questa parte vengono gestiti tutti gli aspetti riguardanti la logica, la gestione, la fisica, le caratteristiche e il comportamento delle componenti di gioco.
- **View**: si occupa degli aspetti grafici, quali esporre le effettive entità del *Model* sullo schermo di gioco. La *View* comprende anche il menù di gioco e la schermata di fine partita per esaminare i risultati della classifica. Riguarda, inoltre, anche la parte di generazione delle mappa che comprende vari tipi di sfondi e relative piattaforme in tema, monete, oggetti, armi e nemici.
- **Controller**: è il concreto ponte, tra il *Model* e la *View*, consente di collegare la parte logica con quella visiva. La sua funzione è quella di ricevere input della tastiera dall'utente, inviarlo al *Model* per essere elaborato e alla *View* per avere un riscontro visivo.

Di seguito, un UML riguardante il pattern architetturale **MVC** utilizzato:

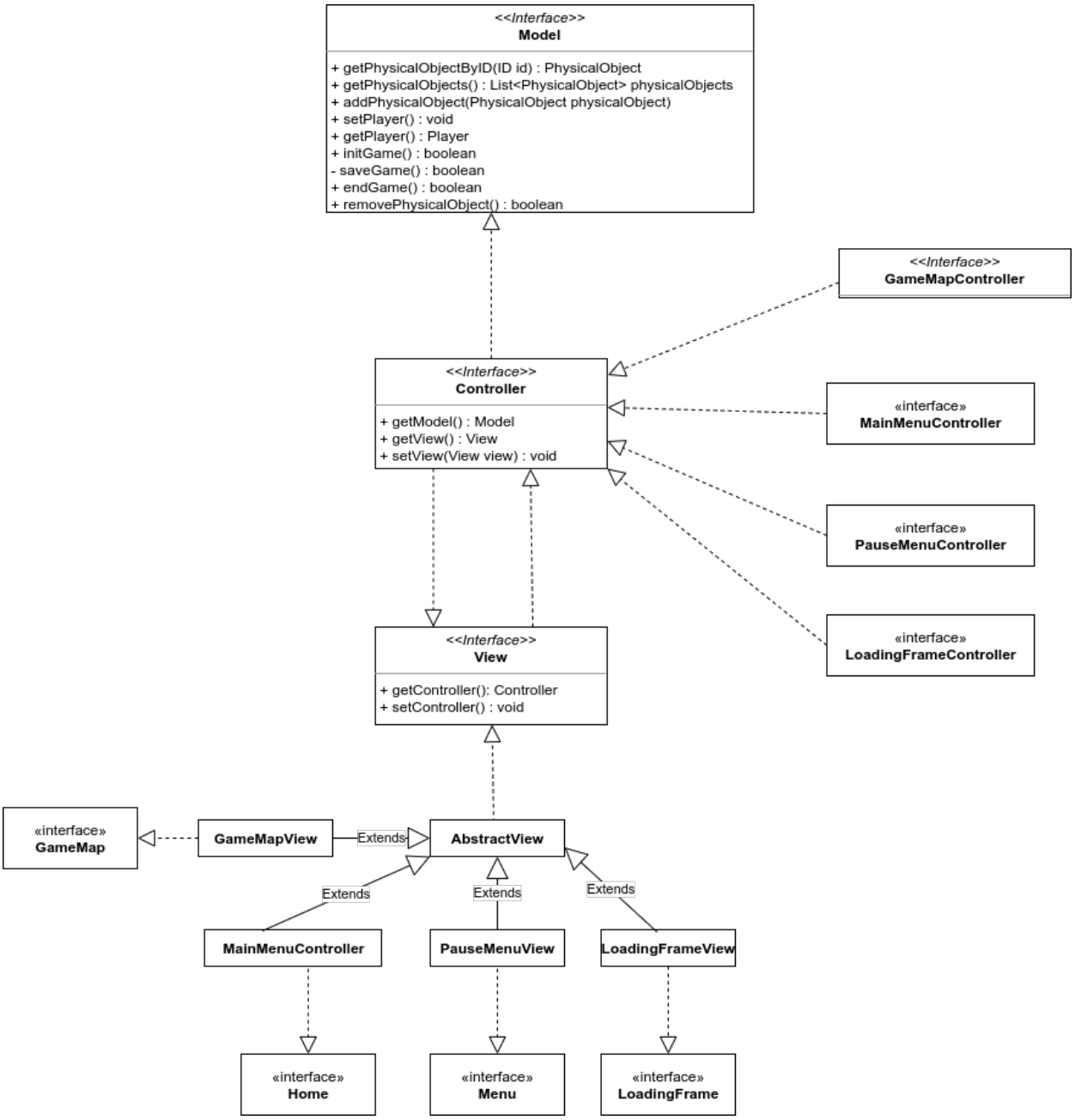

Figura 2.1: Schema UML del pattern architetturale MVC.

2.2 Design dettagliato

Alessandro Pioggia

Physical objects

Lo scopo iniziale del progetto è stato quello di creare delle vere e proprie entità di gioco(Players, Enemies, Obstacles, Items), che potessero popolare la mappa e interagire fra loro. La mia parte richiedeva che io costruissi il core delle entità fisiche di gioco(oltre all'implementazione di alcune di esse), in modo da poter creare una struttura solida alla quale potessero fare riferimento i colleghi, sfruttando il principio della generalizzazione. L'interfaccia **PhysicalObject** rappresenta la struttura base di una entità di gioco, la quale è composta da una dimensione, una posizione e contiene un riferimento all'ambiente di gioco, con relativa implementazione. In prima istanza credevamo fosse opportuno suddividere le GameEntity statiche da quelle dinamiche, separandole in due classi, però abbiamo notato che a livello pratico non ne avevamo alcun beneficio.

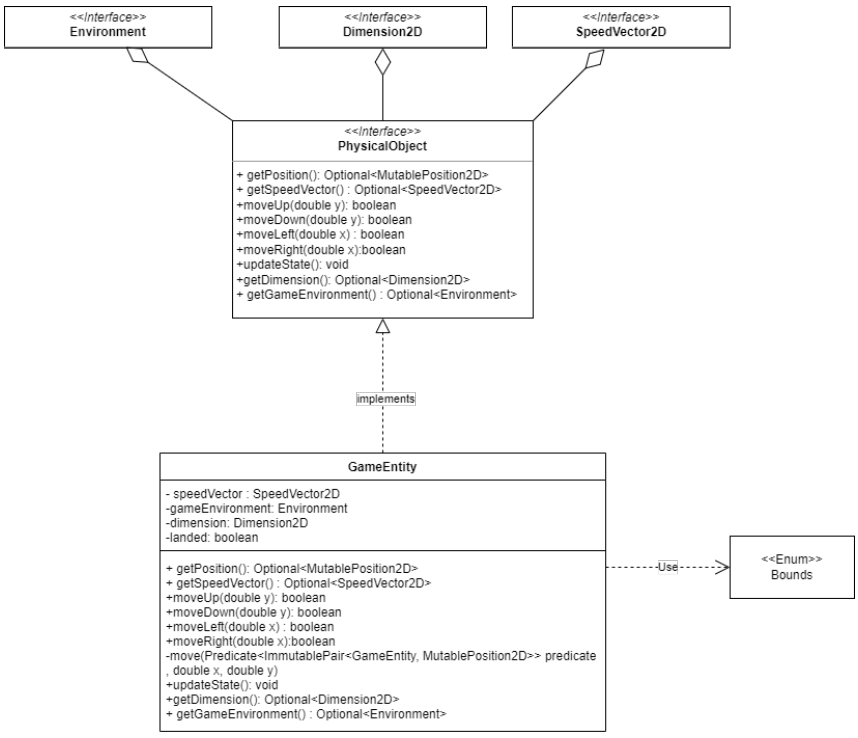

Figura 2.2: Schema UML rappresentante l'interfaccia PhysicalObject con relative dipendenze.

Item

Gli **Item** sono entità presenti all'interno dell'ambiente di gioco, le quali possono essere raccolte dal main player, il quale ne subisce l'effetto, gestito dal mio collega Leon. Per semplificare l'istanziamento ho deciso di utilizzare il factory pattern, il quale è tornato molto utile e ha ridotto la verbosità. Similmente ho concepito la classe **Obstacle**.

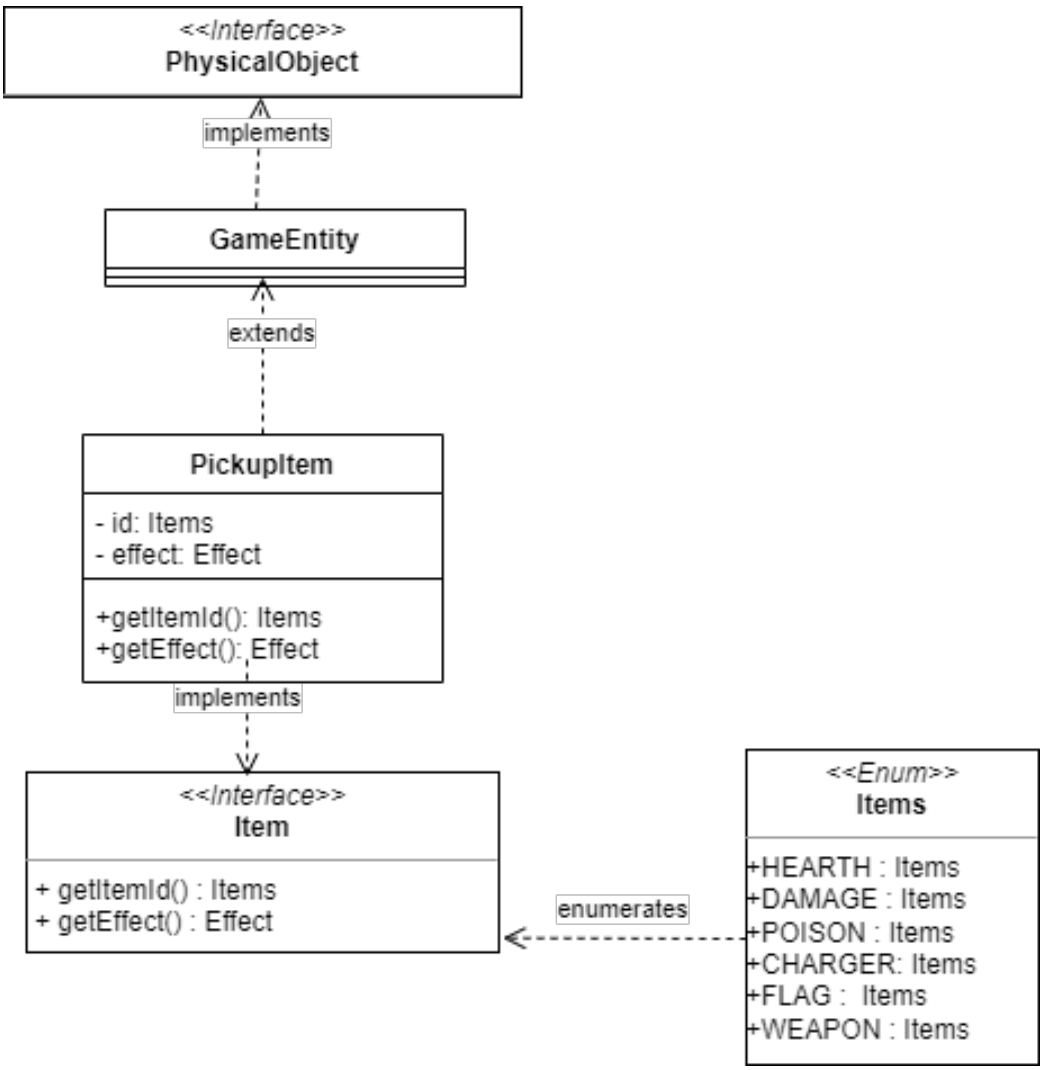

Figura 2.3: Schema UML rappresentante gli Item di gioco

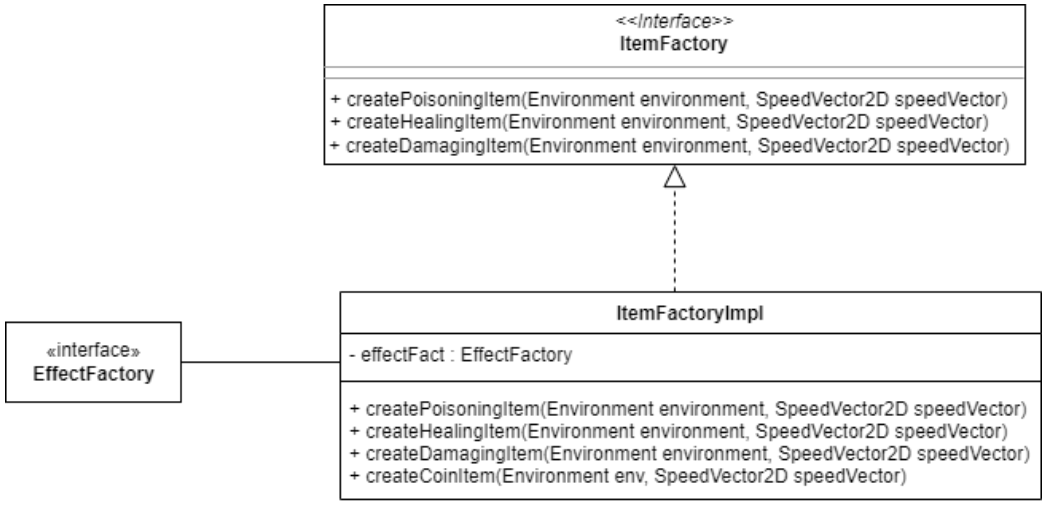

Figura 2.4: Schema UML rappresentante il factory pattern applicato agli Item

Menu

Il main menu non è altro che il menù principale, ovvero quello che viene visualizzato all'avvio dell'applicazione e permette all'utente di decidere se e quando iniziare a giocare. Per realizzarlo ho deciso di sfruttare scene builder, software che mi ha permesso di generare i file fxml, ai quali sono stati associati dei controller, permettendomi di solidificare il concetto di separation of concerns. Ho deciso di creare una classe PageLoaderImpl (che implementa l'interfaccia PageLoader) per caricare agilmente gli fxml.

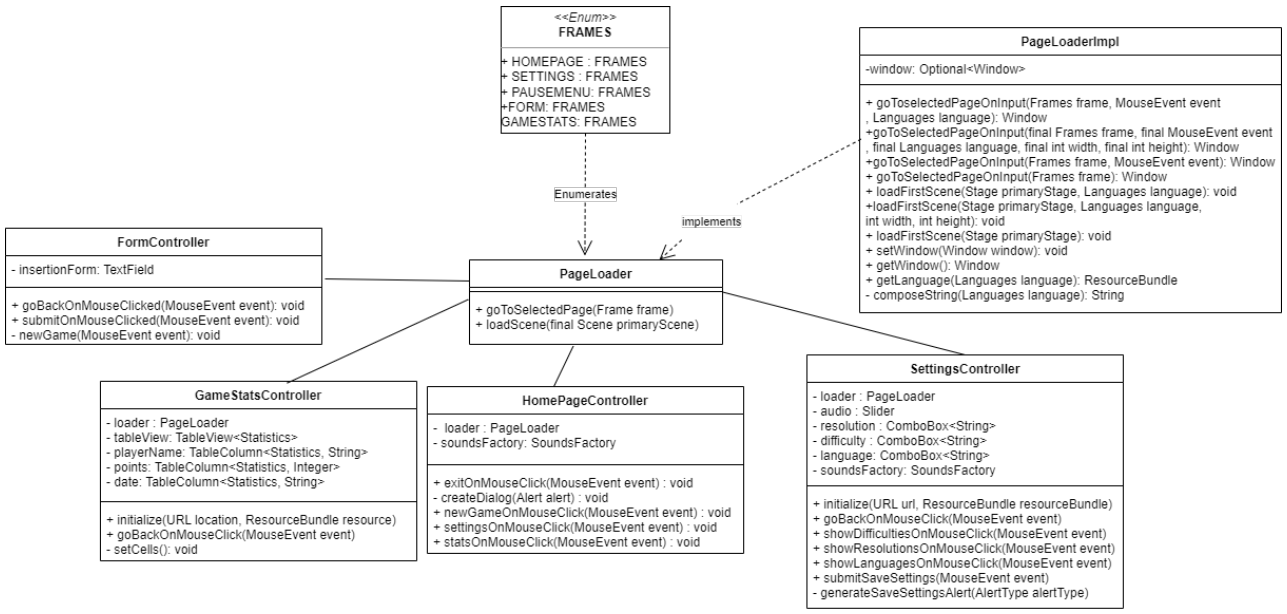

Figura 2.5: Schema UML rappresentante il main menu

PhysicalObjectSprites

Oltre alla parte di model, in comune accordo abbiamo deciso di spartirci la creazione delle relative sprites, per fare ciò ho deciso di creare uno scheletro, in modo che i miei compagni potessero fruirne per rendere la struttura più pulita. A questo proposito ho deciso di creare una classe chiamata **PhysicalObjectSprite**, che contiene metodi, validi per qualsiasi entità di gioco, per la creazione di sprite. Per rendere il tutto più intuitivo e semplice ho deciso di creare una factory, in modo che i miei compagni potessero aggiungere i propri metodi.

Figura 2.6: Schema UML rappresentante il main menu

Sounds

L'ultima componente del progetto che ho gestito riguarda i suoni, per fare ciò ho creato una semplice classe SoundImpl (che implementa l'interfaccia Sound). Per rendere il tutto fruibile, come visto nei casi precedenti, ho deciso di sfruttare il factory pattern(omesso dall'UML perchè la struttura è identica alle altre già presentate).

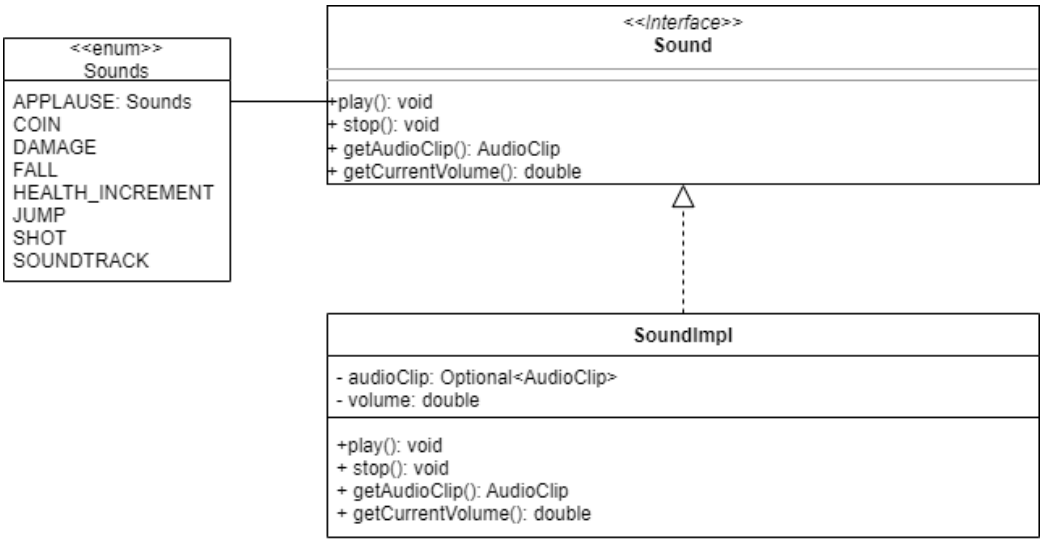

Figura 2.7: Schema UML rappresentante il main menu

Leon Baiocchi

Game environment

Inizialmente è stato essenziale trovare una modalità secondo la quale organizzare tutte le varie entità di gioco presenti così da poterle gestire agevolmente all'interno dell'ambiente di gioco. Questa ricerca mi ha portato a sviluppare il concetto di **Environment**, ossia una sorta di wrapper di tutti i componenti lato model, quindi entità, meccaniche(eventi) e fisica di gioco(gravità, collisioni). Questi componenti vengono gestiti corrispettivamente da: **EntityManager**, che si occupa della gestione delle varie entità all'interno di una struttura dati definita in *AbstractContainer*; campo di tipo *GameEventListener*, viene utilizzato per far sì che il GameEnvironment possa notificare ad un listener i propri eventi. Le collisioni vengono considerate come degli eventi insieme all'evento di game over.

Game engine

Generazione mappa

Federico Brunelli

Weapon e Bullet

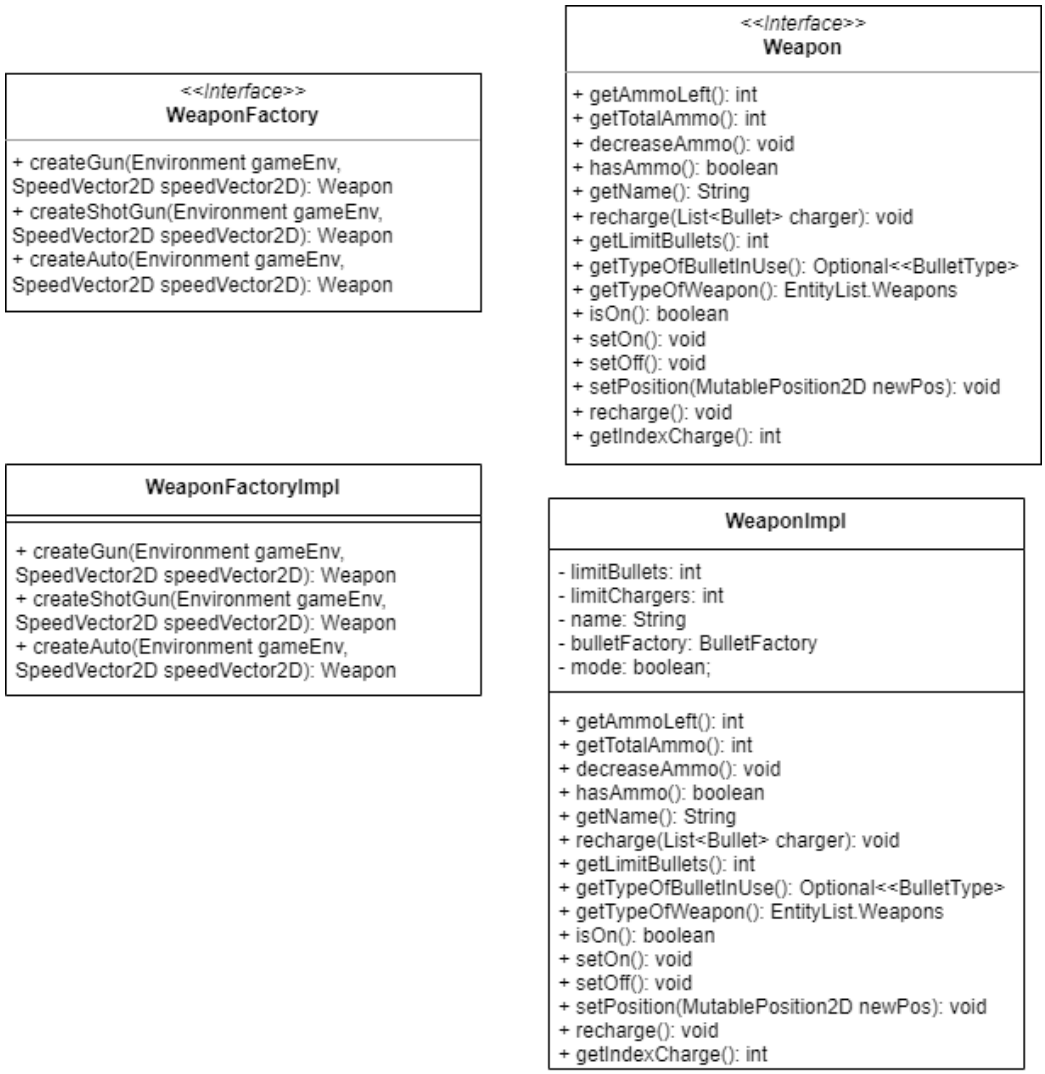

Figura 2.8: Schema UML raffigurante la struttura di Weapon

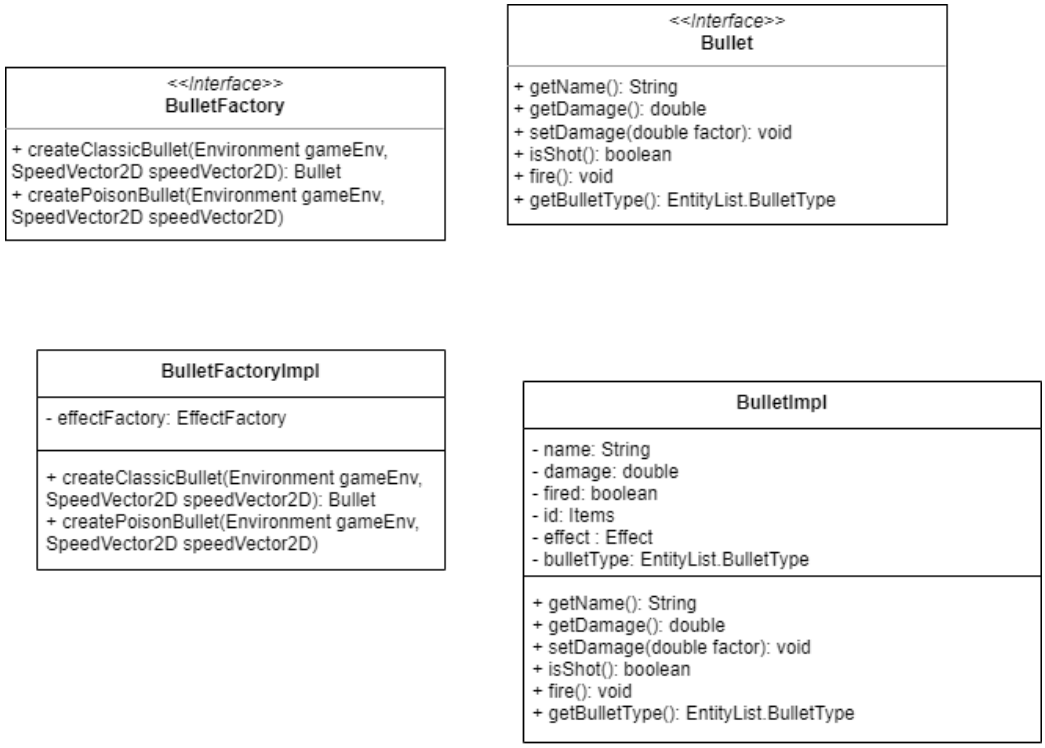

Figura 2.9: Schema UML raffigurante la struttura di Bullet

Menu di pausa

Collisioni

Luca Rengo

La mia parte riguardava l'implementazione dei personaggi di gioco e dei diversi aspetti riguardanti la generazione della mappa e dei suoi componenti a livello di *View*.

Mi competeva, inoltre, l'inizializzazione delle varie scene di gioco, dell'implementazione delle sprites, del salvataggio dei punteggi e delle statistiche completati dal giocatore, delle impostazioni di gioco ed il loro caricamento.

Characters

Per quanto concerne l'implementazione dei personaggi di gioco, ho sviluppato un'interfaccia *Characters* che pone a tutte le figure l'implementazione di un contratto con le varie azioni comuni, i vari metodi che un personaggio può eseguire.

Questo è stato trattato dalle classi *Player* ed *Enemy* che contengono le proprietà e caratteristiche dei personaggi come la loro vita, il mana, il nome, le loro abilità e capacità come quella di muoversi e saltare.

Ho utilizzato poi un enum, *EntityList*, per poter rappresentare diversi tipi di personaggi con diverse caratteristiche che ho settato nel metodo *setPlayerType()* e *setEnemyType()* delle rispettive classi.

Mi sono avvalso, inoltre, del pattern architetturale **Factory** per raccogliere le varie tipologie di *Player* ed *Enemy* attraverso l'interfaccia *FactoryCharacters* e la sua implementazione *FactoryCharactersImpl* per riorganizzare meglio i vari personaggi del gioco e rendere la loro creazione più chiara e semplice, evitando così di dover passare, ogni volta, un eccessivo numero di parametri da inizializzare.

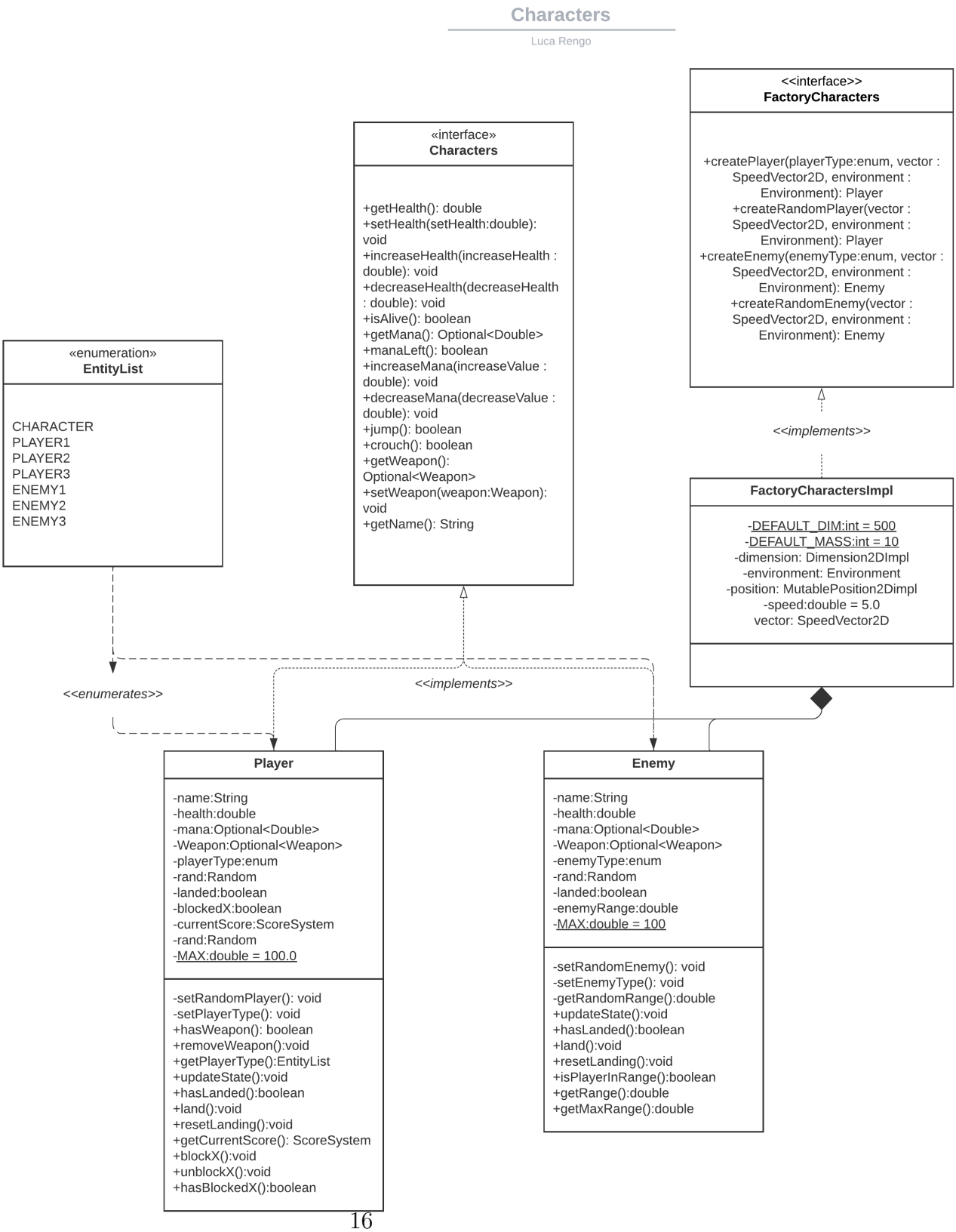

Figura 2.10: Schema UML del *Model* relativo ai *Characters*.

Mappa di gioco

Per quanto riguarda la *Mappa* mi sono occupato degli aspetti della *View*.

È presente l'interfaccia *GameView* che viene implementata dalla classe *MapScene* che implementa una specifica scena del gioco, ovvero quella della mappa, inizializzando tutte le sprites delle varie entità di gioco e generandole sullo schermo. Poi abbiamo la classe astratta *AbstractScene* che ha il compito di rappresentare una generica scena edl gioco, come la mappa di gioco.

In *MapScene* vengono generate le sprites dalle classi *Platform*, *Coin*, *MainEnemy*, *MainPlayer* con la sua animazione implementata in *SpriteAnimation*.

Queste entità vengono posizionate in una precisa posizione in base alla locazione di un carattere in un array di stringhe che si trovano in dei files .txt che vengono caricati dalla classe *LevelLoader*.

Ad ogni entità del gioco è associato un carattere (un numero o un simbolo) presenti nell'enum *LevelEntity* che si trovano nei files .dat nella directory *levels/*.

Le classi *Platform* e *Coin* sono simili ed entrambe hanno un proprio enum per le proprie varie tipologie e utilizzano una ImageView per settare l'immagine, le proprietà e le posizioni sullo schermo.

MainEnemy è una semplice ImageView statica mentre *MainPlayer* è una ImageView dinamica, ovvero con un'animazione. *SpriteAnimation* è la classe che carica da *MainPlayer* l'immagine e le sue caratteristiche (come width dell'immagine, height, ecc..) ed interpola le varie immagini della spritesheet del giocatore per creare la sua animazione che dura tot millisecondi.

Quale sfondo della mappa da mostrare e la relativa piattaforma in tema e quale moneta vengono settate nella classe *BackgroundMap* che tiene traccia di questi elementi della View.

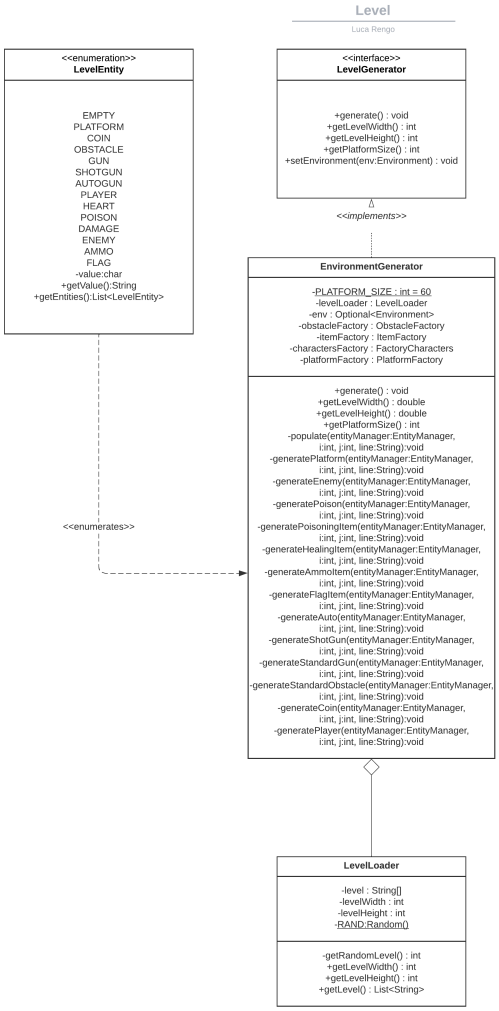

Figura 2.11: Schema UML della generazione dei livelli in *LevelLoader*, *Level-Generator* e *Environment Generator*.

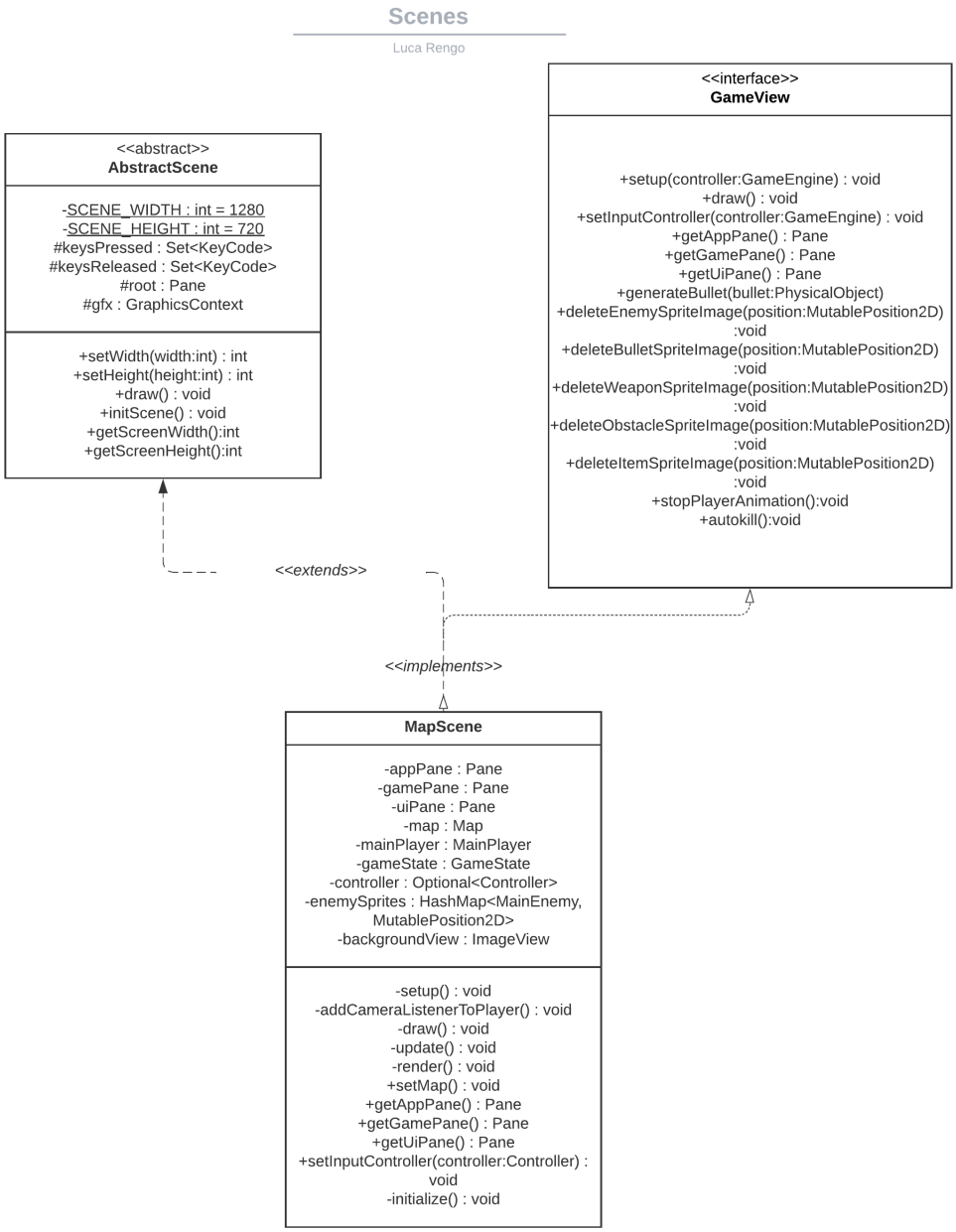

Figura 2.12: Schema UML della *View* relativa alla *AbstractScene*, *MapScene*, *MainPlayer* e *MainEnemy*.

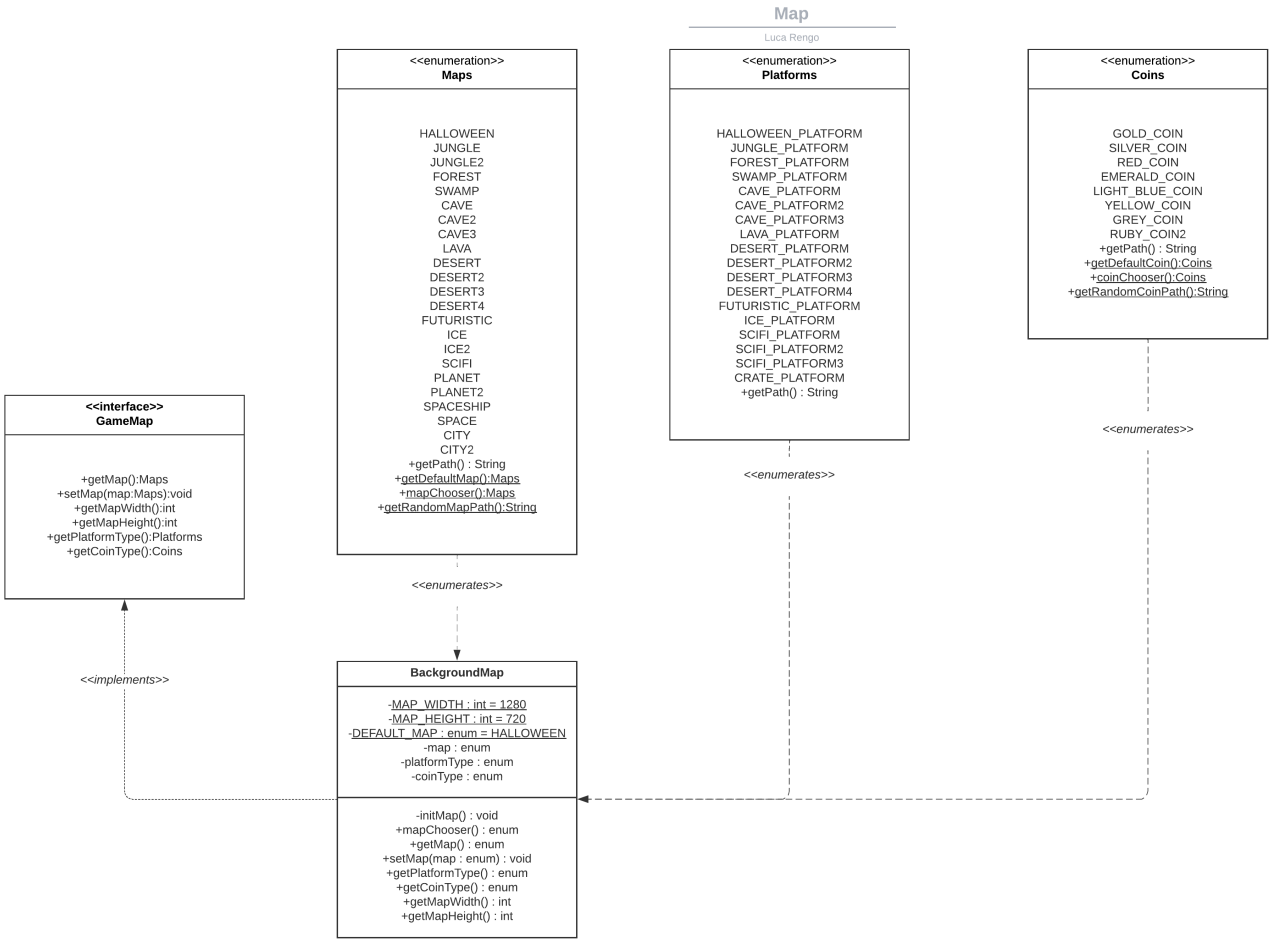

Figura 2.13: Schema UML della *View* relativa alla *Map*, alle *Platform* e ai *Coin*.

Salvataggio

Per quanto riguarda l'operazione di salvataggio delle statistiche di gioco, mi sono avvalso della classe *Save* che crea un file di salvataggio in cui vengono contenuti il nome dei giocatori e i loro relativi punteggi e data in cui la partita è stata effettuata.

Nel *GameOverEvent* quando il player muore e quindi il gioco finisce, viene chiamato il metodo *saveGameStatistics()* e gli si passa il nome, il punteggio e la data attraverso un *SimpleDateFormat*.

Per fare questo, usufruisco di un *jsonArray* e se nel file erano già presenti dei dati, li decripto e li appendo e li riencripto altrimenti aggiungo i dati e li encripto.

Per poter caricare questi dati dal file, lo decripto, faccio il parsing della stringa in chiaro attraverso un *jsonParser* e ottenendo gli *jsonArray* e da questo formo e restituisco una *Mappa<String, MutablePair<String,String»*.

Inoltre, ho un metodo per modificare i dati che erano stati precedentemente salvati per evitare di dover sovrascrivere il file ogni volta.

Per quanto riguarda il salvataggio e il caricamento delle impostazioni di gioco ho usato il medesimo meccanismo, ma usando dei *jsonObject* perchè ogni dato può essere considerato in maniera separata.

Inoltre, le impostazioni vengono poi automaticamente recuperate e caricate nel menu delle Impostazioni all'avvio del gioco attraverso la classe *SettingsController* e in cui, una volta premuto Salva vengono salvate su file e mostra a schermata un *Alert* con successo se l'operazione è andata a buon fine altrimenti un *Alert* di errore.

Per quanto riguarda i livelli non vengono memorizzati come .json, perché non ho voluto salvarli tutti nello stesso file, ma in file separati e quindi ho optato per lasciarli in .txt ed come sempre criptarli.

Ogni file di ogni livello è composto da righe (matrici) di caratteri, ciascun carattere rappresenta un'entità di gioco, queste sono segnate nell'enum *LevelEntity*.

Ci son due tipi metodi per caricare i files, uno per caricare il livello in .txt, questo può essere utile per testare i livelli e poterli modificare, cosa non possibile con i files criptati ed uno che decripta e carica i files .dat.

- Per quanto riguarda i .txt usufruisco di un *BufferReader* per leggere riga per riga del file e restituirle.
- Mentre per i files .dat decripto il livello, ottengo una stringa in chiaro e attraverso un'espressione *regex* separo riga per riga e dopodichè le restituisco.

Ho poi un metodo per encriptare i files se sono presenti dei files in .txt

Infine, ho un metodo per resettare il file in base alla path passata come parametro e cancellare tutti i dati che erano stati immagazzinati.

2.2.1 Criptare e Decriptare i dati del salvataggio

La classe *SecureData* si occupa di criptare e decriptare dati e files.

Qui, usufruisco di un *SecureRandom* per generare un numero pseudo-casuale per creare un IV, (initialization vector) che serve per aggiungere casualità al processo di criptazione, e la salatura che serve per aggiungere dei bytes aggiuntivi alla password prima che passi per l'algoritmo di hashing.

Poi genero la password attraverso una *SecretKey* fornendo l'algoritmo, la password, la salatura, il contatore delle iterazioni e lo spazio della chiave e lo standard di criptazione, in questo caso AES (*Advanced Encryption System*).

Per la criptazione di dati, passiamo al metodo il messaggio e la password che vogliamo usare, poi otteniamo l'iv e la salatura e la chiave dopodichè settiamo la modalità del cifrario su **ENCRYPT** e gli passiamo la chiave e un *GCMParameterSpec* che specifica un insieme di parametri necessari al cifrario usando la modalità **Galois**/**Counter Mode** e dopodiché criptiamo il nostro messaggio e lo restituiamo.

Per la decriptazione, recuperiamo dal messaggio criptato, la salatura e l'iv e ciò che rimane del messaggio, otteniamo la chiave attraverso la password che deve essere la stessa usata per la criptazione, dopodichè col cifrario in modalità **DECRYPT** decriptiamo il messaggio e lo restituiamo.

Per criptare direttamente il file leggiamo tutti i bytes di esso e li criptiamo utilizzando il metodo *encrypt* dopodichè li scriviamo sul file.

Per decriptare direttamente il file leggiamo tutti i bytes e chiamiamo il metodo *decrypt* e restituiamo il messaggio.

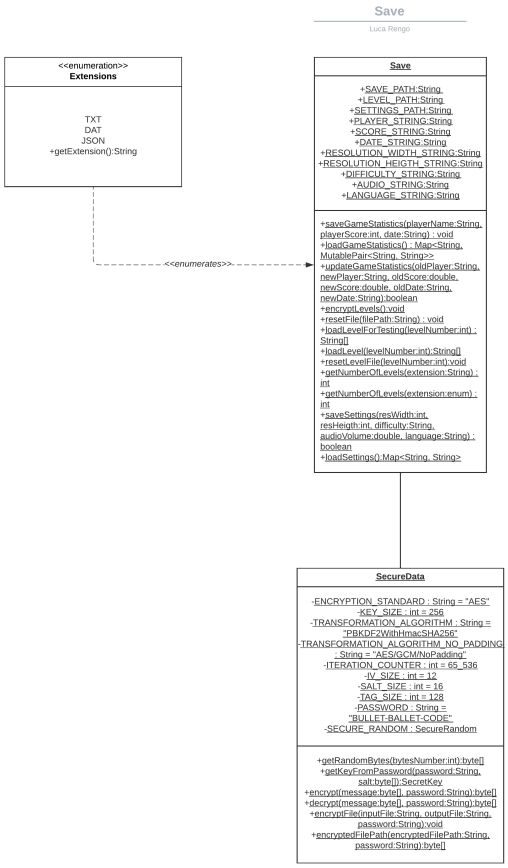

Figura 2.14: Schema UML del *Model* relativo al *Salvataggio* del gioco.

2.2.2 Lingue del gioco

Per quanto concerne la traduzione del gioco ho adoperato **JavaFX Internazionalization** che permette di lavorare sulle stringhe dei files fxml e di cambiarle in base alla lingua selezionata nel menu di impostazioni.

Le traduzioni delle varie lingue si trovano in dei files bundles **.properties**. In *SettingsController* possiamo caricare e settare la lingua dal file con l'enum *Languages*, poi quando viene caricata una nuova pagina del menù viene recuperato dal file delle impostazioni la lingua e così utilizza il file **.properties** corrispondente.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per il seguente progetto software è stato largamente sfruttato il testing automatizzato, dal momento che è stato scelto un approccio TDD(test-driven-development). Il nostro gruppo ha considerato opportuno testare prevalentemente la componente di logica(model) oltre ai vari controller. Abbiamo inoltre attribuito importanza alla "pulizia" dei test, per garantire la loro manutenibilità, effettuando i seguenti accorgimenti :

- Utilizzo di una struttura fissa secondo il quale in primis si preparano i dati del test, successivamente si opera su di essi ed infine si controlla che i risultati siano quelli previsti.
- Tentativo di mantenere i test indipendenti, in modo da poter individuare gli errori in modo preciso ed analitico.
- Dichiarazione dei metodi di test con nomi autoesplicativi, in modo da poter individuare lo scopo del test senza la necessità di inserire commenti ridondanti e voluminosi.

Alessandro Pioggia

- GameEntityTest;
- ObstacleTest;
- SpeedVector2DTest;
- Dimension2DTest;

Leon Baiocchi

- todo : scrivere le classi di test utilizzate

Federico Brunelli

- todo : scrivere le classi di test utilizzate

Luca Rengo

- CharactersTest
- MapTest
- SaveTest
- SecureDataTest

3.2 Metodologia di lavoro

Nella prima fase di realizzazione del progetto ci siamo dedicati all'analisi, in cui abbiamo lavorato in maniera coordinata per definire la struttura e le funzionalità del progetto. Successivamente siamo passati al design, in cui è stato deciso di creare uno scheletro in UML che descrivesse a grandi linee le entità principali necessarie per il funzionamento, definendo già le dipendenze fra esse. Il processo descritto ci ha permesso di definire una suddivisione del lavoro in 4 parti, in modo da garantire la prosecuzione del lavoro individuale con la fase di design dettagliato.
Per facilitare lo sviluppo abbiamo sfruttato un real-time-mantained UML, ovvero una repository in cui ciascun membro, una volta terminata una sessione di lavoro, aveva la premura di aggiornare uno schema UML, costruendo/modificando un diagramma delle classi che descrivesse il lavoro svolto.
Per quanto riguarda il DVCS abbiamo optato per l'utilizzo di git, in accordo con le nozioni apprese a lezione. La metodologia utilizzata è stata la seguente:

- Sviluppo in feature-branches, ovvero branches in cui venivano realizzate singolarmente le feature dell'applicativo;
- Ognuno di noi aveva a disposizione un numero definito di branches indipendenti dagli altri;
- I feature-branches venivano poi confluiti nel main-branch.

Alessandro Pioggia

Lavoro svolto :

- Realizzazione del menù principale con relativi reindirizzamenti.
- Creazione dello scheletro per l'implementazione delle entità di gioco fisiche, ovvero la creazione dell'interfaccia PhysicalObject la relativa implementazione;
- Implementazione di oggetti di gioco pickable (Item) e di ostacoli(Obstacles), realizzazione di sprites compresa;
- Creazione di commons, quali Dimension2D e SpeedVector2D;
- Creazione dello scheletro PhysicalObjectSprite, per la creazione delle sprite (aspetto di view);
- Gestione dei suoni.

Leon Baiocchi

- todo : scrivere quello che si è fatto nel progetto.

Federico Brunelli

- todo : scrivere quello che si è fatto nel progetto.

Luca Rengo

- Modellazione del *giocatore* e dei *nemici*.
- Composizione di una classe astratta per la formazione di una generica scena di gioco.
- Generazione della Mappa di gioco e delle relative entità dal punto di vista della *View* come le *piattaforme*, *monete*, il *giocatore* e i *nemici*.
- Implementazione dell'animazione del player.
- Creazione di una classe per il salvataggio dei dati e delle statistiche di gioco.

3.3 Note di sviluppo

Alessandro

- ***JavaFx*** : utilizzato per la realizzazione del menù;
- ***Optionals*** : per evitare di ritornare valori null;
- ***Apache library*** : libreria che ho importato per l'utilizzo delle classi Pair preimplementate;
- ***Gradle*** : strumento che ho utilizzato per importare le varie componenti di javafx;
- ***Reflection*** : utilizzata per la creazione delle statistiche, perchè richiesta da javafx;
- ***JUnit*** : utilizzata per il testing;
- ***Streams e lambda*** : utilizzate dove possibile, per migliorare la leggibilità e l'efficienza.

Leon Baiocchi

- todo : scrivere quello che si è fatto nel progetto.

Federico Brunelli

- todo : scrivere quello che si è fatto nel progetto.

Luca Rengo

- **JavaFX**: sfruttato per la realizzazione della mappa e delle sue componenti.
- **Factory**: adoperata per raggruppare le mie classi ed instanziarle più facilmente.
- **JUnit**: beneficiata per l'esaminazione delle parti di codice create.
- **Lambda**: utilizzate per semplificare alcune parti del programma.
- **Gradle**: impiegato per importare i vari moduli di JavaFX, JUnit e le varie dipendenze e per tenere il progetto ben organizzato.
- **Apache library**: utilizzata per usufruire delle classi dei Pair.

- **json-simple**: adoperata per salvare i file in formato .json
- **JavaFX Internazionalization**: usata per la traduzione del gioco in diverse lingue.
- **Optionals**: usati quando avevo valori opzionali.
- **javax.crypto, java.security**: per encriptare e decriptare i files.
- **Regex** : utilizzato per separare riga per riga nel caricamento dei dati dei livelli.

Capitolo 4

Commenti finali

In quest'ultimo capitolo si tirano le somme del lavoro svolto e si delineano eventuali sviluppi futuri.

Nessuna delle informazioni incluse in questo capitolo verrà utilizzata per formulare la valutazione finale, a meno che non sia assente o manchino delle sezioni obbligatorie. Al fine di evitare pregiudizi involontari, l'intero capitolo verrà letto dai docenti solo dopo aver formulato la valutazione.

4.1 Autovalutazione e lavori futuri

Alessandro Pioggia

Il seguente progetto mi ha permesso di crescere tanto e sopratutto di riconoscere e individuare gli errori da me commessi. A mio avviso si è rivelato fondamentale seguire le direttive a noi impartite, specialmente in un lavoro a gruppi, anche un solo metodo non chiamato in maniera adeguata può mettere in seria difficoltà i compagni. Devo ammettere che non è stato facile gestire le tempistiche, non ho considerato il tempo necessario per il setup del progetto e per eventuali errori(esempio : errori nella gradle build).

Punti di forza:

- flessibilità, capacità di mettersi in discussione
- comunicazione con i compagni

Punti deboli :

- gestione delle tempistiche

- mancato uso di programmazione funzionale
- ridotta complessità generale

Leon Baiocchi

Punti di forza:

-
-
-

Punti deboli:

-
-
-

Federico Brunelli

Punti di forza:

-
-
-

Punti deboli:

-
-
-

Luca Rengo

Questo primo progetto mi ha sicuramente fatto comprendere meglio l'importanza di una buona organizzazione, gestione delle tempistiche e di quanto sia essenziale una corretta coordinazione e comunicazione del lavoro.
Alla fine di tutto, ciò che è fondamentale per una produttiva esecuzione del progetto è un effettivo teamwork e workflow.

Punti di forza:

- Esaminare e discutere i vari aspetti del progetto assieme ai colleghi e cercare di trovare soluzioni comuni sul come affrontarli.
- Complementarietà del team, ogni membro ha la sua parte specifica del progetto.
- Risoluzione dei problemi in maniera unita.
- Scambio di opinioni oneste.

Punti deboli:

- Gestione del tempo a disposizione.
- Poca chiarezza sul da farsi in modo pratico e preciso, dovuto anche al fatto che fosse la prima volta che facevamo un progetto così grande.
- Realizzazione del lavoro e della sua comunicazione.

4.2 Difficoltà incontrate e commenti per i docenti

Questa sezione, **opzionale**, può essere utilizzata per segnalare ai docenti eventuali problemi o difficoltà incontrate nel corso o nello svolgimento del progetto, può essere vista come una seconda possibilità di valutare il corso (dopo quella offerta dalle rilevazioni della didattica) avendo anche conoscenza delle modalità e delle difficoltà collegate all'esame, cosa impossibile da fare usando le valutazioni in aula per ovvie ragioni. È possibile che alcuni dei commenti forniti vengano utilizzati per migliorare il corso in futuro: sebbene non andrà a vostro beneficio, potreste fare un favore ai vostri futuri colleghi. Ovviamente *il contenuto della sezione non impatterà il voto finale.*

Appendice A

Guida utente

Capitolo in cui si spiega come utilizzare il software. Nel caso in cui il suo uso sia del tutto banale, tale capitolo può essere omesso. A tal riguardo, si fa presente agli studenti che i docenti non hanno mai utilizzato il software prima, per cui aspetti che sembrano del tutto banali a chi ha sviluppato l'applicazione possono non esserlo per chi la usa per la prima volta. Se, ad esempio, per cominciare una partita con un videogioco è necessario premere la barra spaziatrice, o il tasto "P", è necessario che gli studenti lo segnalino.

Elementi positivi

- Si istruisce in modo semplice l'utente sull'uso dell'applicazione, eventualmente facendo uso di schermate e descrizioni.

Elementi negativi

- Si descrivono in modo eccessivamente minuzioso tutte le caratteristiche, anche minori, del software in oggetto.
- Manca una descrizione che consenta ad un utente qualunque di utilizzare almeno le funzionalità primarie dell'applicativo.

Appendice B

Esercitazioni di laboratorio

In questo capitolo ciascuno studente elenca gli esercizi di laboratorio che ha svolto (se ne ha svolti), elencando i permalink dei post sul forum dove è avvenuta la consegna.

Esempio

B.0.1 Alessandro Pioggia

- Laboratorio 04: https://virtuale.unibo.it/mod/forum/discuss.php?d=62685#p101507
- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=62684#p101217
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=62579#p100880
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p100893
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p107314
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p103992
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=66753#p106887

B.0.2 Paperon De Paperoni

- Laboratorio 04: https://virtuale.unibo.it/mod/forum/discuss.php?d=12345#p123456
- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=99999#p999999
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=99999#p999999
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=99999#p999999
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=22222#p222222