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.

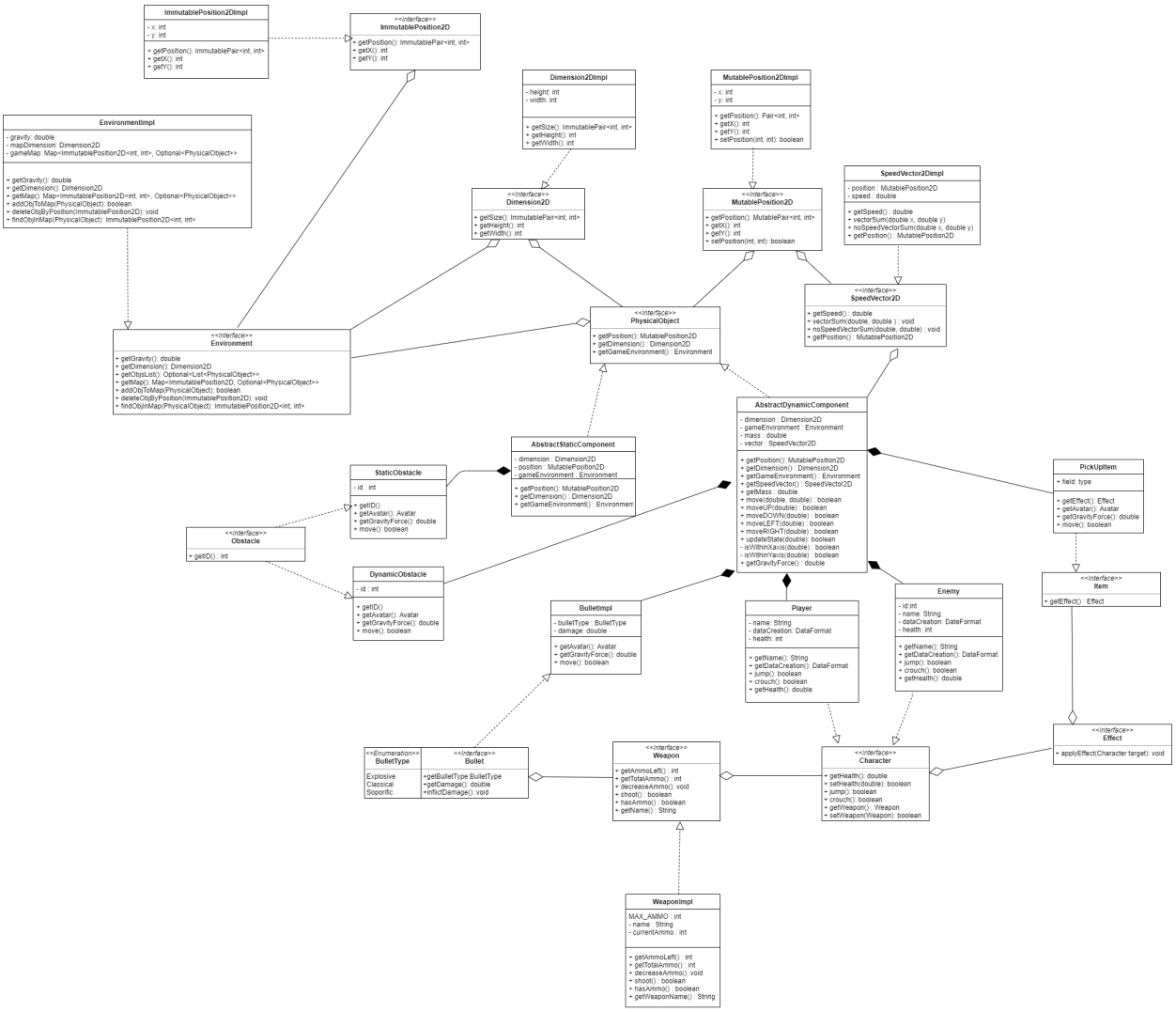

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro.

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:

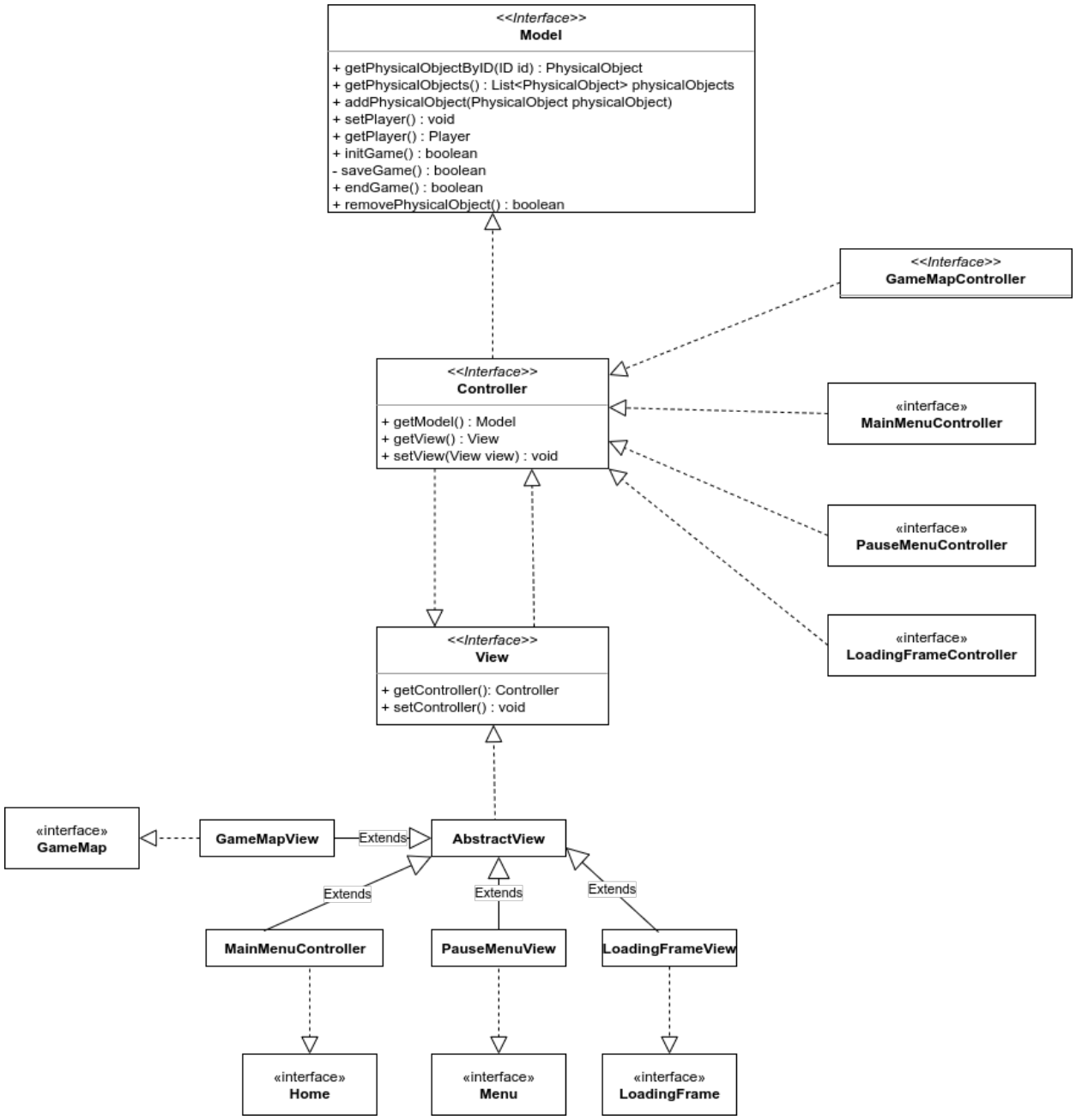

Figura 2.1: Schema UML del pattern architetturale MVC.

2.2 Design dettagliato

In questa sezione si possono approfondire alcuni elementi del design con maggior dettaglio. Mentre ci attendiamo principalmente (o solo) interfacce negli schemi UML delle sezioni precedenti, in questa sezione è necessario scendere in maggior dettaglio presentando la struttura di alcune sottoparti rilevanti dell'applicazione. È molto importante che, descrivendo un problema, quando possibile si mostri che non si è re-inventata la ruota ma si è applicato un design pattern noto. È assolutamente inutile, ed è anzi controproducente, descrivere classe-per-classe (o peggio ancora metodo-per-metodo) com'è fatto il vostro software: è un livello di dettaglio proprio della documentazione dell'API (deducibile dalla Javadoc).

È necessario che ciascun membro del gruppo abbia una propria sezione di design dettagliato, di cui sarà il solo responsabile. Ciascun autore dovrà spiegare in modo corretto e giustamente approfondito (non troppo in dettaglio, non superficialmente) il proprio contributo. È importante focalizzarsi sulle scelte che hanno un impatto positivo sul riuso, sull'estensibilità, e sulla chiarezza dell'applicazione. Esattamente come nessun ingegnere meccanico presenta un solo foglio con l'intero progetto di una vettura di Formula 1, ma molteplici fogli di progetto che mostrano a livelli di dettaglio differenti le varie parti della vettura e le modalità di connessione fra le parti, così ci aspettiamo che voi, futuri ingegneri informatici, ci presentiate prima una visione globale del progetto, e via via siate in grado di dettagliare le singole parti, scartando i componenti che non interessano quella in esame. Per continuare il parallelo con la vettura di Formula 1, se nei fogli di progetto che mostrano il design delle sospensioni anteriori appaiono pezzi che appartengono al volante o al turbo, c'è una chiara indicazione di qualche problema di design.

Usare correttamente i design pattern in questa sezione è molto importante: se vengono utilizzati correttamente, è molto probabile riuscire a progettare il software in modo corretto, estensibile, e riusabile. Per ogni pattern utilizzato si presenti:

- almeno un paragrafo che spieghi come è reificato nel progetto (ad esempio: nel caso di Template Method, qual è il metodo template; nel caso di Strategy, quale interfaccia del progetto rappresenta la strategia, e quali sono le sue implementazioni; nel caso di Decorator, qual è la classe astratta che fa da Decorator e quali sono le sue implementazioni concrete; eccetera);
- almeno uno schema UML che grafichi quanto sopra descritto.

La presenza di pattern di progettazione *correttamente utilizzati* è valutata molto positivamente. L'uso inappropriato è invece valutato negativamente:

a tal proposito, si raccomanda di porre particolare attenzione all'abuso di Singleton, che, se usato in modo inappropriato, è di fatto un anti-pattern.

Elementi positivi

- Ogni membro del gruppo discute le proprie decisioni di progettazione, ed in particolare le azioni volte ad anticipare possibili cambiamenti futuri (ad esempio l'aggiunta di una nuova funzionalità, o il miglioramento di una esistente).
- Si identificano, utilizzano *appropriatamente*, e descrivono come suggerito diversi design pattern.
- Ogni membro del gruppo identifica i pattern utilizzati nella sua sotto-parte.
- Si mostrano gli aspetti di design più rilevanti dell'applicazione, mettendo in luce la maniera in cui si è costruita la soluzione ai problemi descritti nell'analisi.
- Si tralasciano aspetti strettamente implementativi e quelli non rilevanti, non mostrandoli negli schemi UML (ad esempio, campi privati) e non descrivendoli.
- Si mostrano le principali interazioni fra le varie componenti che collaborano alla soluzione di un determinato problema.
- Ciascun design pattern identificato presenta una piccola descrizione del problema calato nell'applicazione, uno schema UML che ne mostra la concretizzazione nelle classi del progetto, ed una breve descrizione della motivazione per cui tale pattern è stato scelto. Ad esempio, se si dichiara di aver usato Observer, è necessario specificare chi sia l'observable e chi l'observer; se si usa Template Method, è necessario indicare quale sia il metodo template; se si usa Strategy, è necessario identificare l'interfaccia che rappresenta la strategia; e via dicendo.

Elementi negativi

- Il design del modello risulta scorrelato dal problema descritto in analisi.
- Si tratta in modo prolisso, classe per classe, il software realizzato.
- Non si presentano schemi UML esemplificativi.

- Non si individuano design pattern, o si individuano in modo errato (si spaccia per design pattern qualcosa che non lo è).
- Si utilizzano design pattern in modo inopportuno. Un esempio classico è l'abuso di Singleton per entità che possono essere univoche ma non devono necessariamente esserlo. Si rammenta che Singleton ha senso nel secondo caso (ad esempio `System` e `Runtime` sono singleton), mentre rischia di essere un problema nel secondo. Ad esempio, se si rendesse singleton il motore di un videogioco, sarebbe impossibile riusarlo per costruire un server per partite online (dove, presumibilmente, si gestiscono parallelamente più partite).
- Si producono schemi UML caotici e difficili da leggere, che comprendono inutili elementi di dettaglio.
- Si presentano schemi UML con classi (nel senso UML del termine) che "galleggiano" nello schema, non connesse, ossia senza relazioni con il resto degli elementi inseriti.
- Si tratta in modo inutilmente prolisso la divisione in package, elencando ad esempio le classi una per una.

Esempio minimale (e quindi parziale) di sezione di progetto con UML ben realizzati

In questa sezione ci si concentrerà sugli aspetti di personalità e sul funzionamento del reporting di GLaDOS.

Il sistema per la gestione della personalità utilizza il pattern Strategy, come da Figura 2.2: le implementazioni di `Personality` possono essere modificate, e la modifica impatta direttamente sul comportamento di GLaDOS.

Sono state attualmente implementate due personalità, una buona ed una cattiva. Quella buona restituisce sempre una torta vera, mentre quella cattiva restituisce sempre la promessa di una torta che verrà in realtà disattesa. Dato che le due personalità differiscono solo per il comportamento da effettuarsi in caso di percorso completato con successo, è stato utilizzato il pattern template method per massimizzare il riuso, come da Figura 2.3. Il metodo template è `onSuccess()`, che chiama un metodo astratto e protetto `makeCake()`.

Per quanto riguarda il reporting, è stato utilizzato il pattern Observer per consentire la comunicazione uno-a-molti fra `GLaDOS` ed i sistemi di output. `GLaDOS` è observable, e le istanze di `Input` sono observer. Il suo utilizzo è esemplificato in Figura 2.4

Contro-esempio: pessimo diagramma UML

In Figura 2.5 è mostrato il modo **sbagliato** di fare le cose. Questo schema è fatto male perché:

- È caotico.
- È difficile da leggere e capire.
- Vi sono troppe classi, e non si capisce bene quali siano i rapporti che intercorrono fra loro.
- Si mostrano elementi implementativi irrilevanti, come i campi e i metodi privati nella classe `AbstractEnvironment`.
- Se l'intenzione era quella di costruire un diagramma architetturale, allora lo schema è ancora più sbagliato, perché mostra pezzi di implementazione.
- Una delle classi, in alto al centro, galleggia nello schema, non connessa a nessuna altra classe, e di fatto costituisce da sola un secondo schema UML scorrelato al resto
- Le interfacce presentano tutti i metodi e non una selezione che aiuti il lettore a capire quale parte del sistema si vuol mostrare.

Alessandro Pioggia

Leon Baiocchi

Federico Brunelli

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.

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.

Mappa di gioco

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

E' presente l'interfaccia *GameView* che viene implementata dalla classe astratta *AbstractScene* che ha il compito di rappresentare una generica scena del gioco, come la mappa o i crediti del gioco. Questa è ereditata 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.

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 trova nella classe *LevelData*.

Ad ogni entità del gioco è associato un carattere che si trova nell'array di stringhe in *LevelData*.

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 *Map* che tiene traccia di questi elementi della View.

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.

Per fare questo, usufruisco di un *BufferedWriter* per creare il file qualora questo non esista già. Altrimenti, mi limito ad aggiungere i dati in coda al file per evitare di eliminare ciò che era stato precedentemente memorizzato.

Poi mi avvalgo di un *BufferedReader* per leggere dal file e memorizzare i dati e restituire un *HashMap<String, Integer>* con il nome del player e il suo relativo punteggio.

Infine, ho un metodo per resettare il file e cancellare tutti i dati che erano stati immagazzinati.

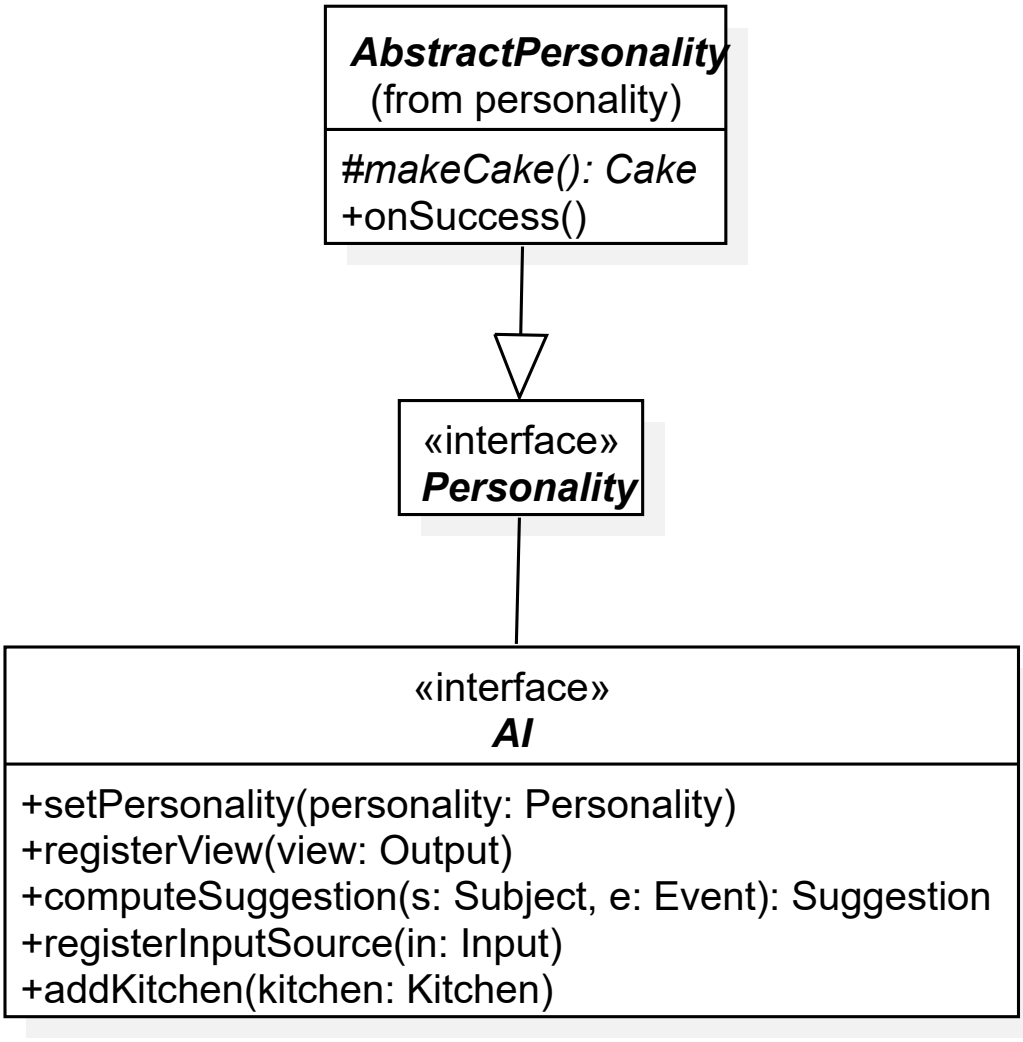

Figura 2.2: Rappresentazione UML del pattern Strategy per la personalità di GLaDOS

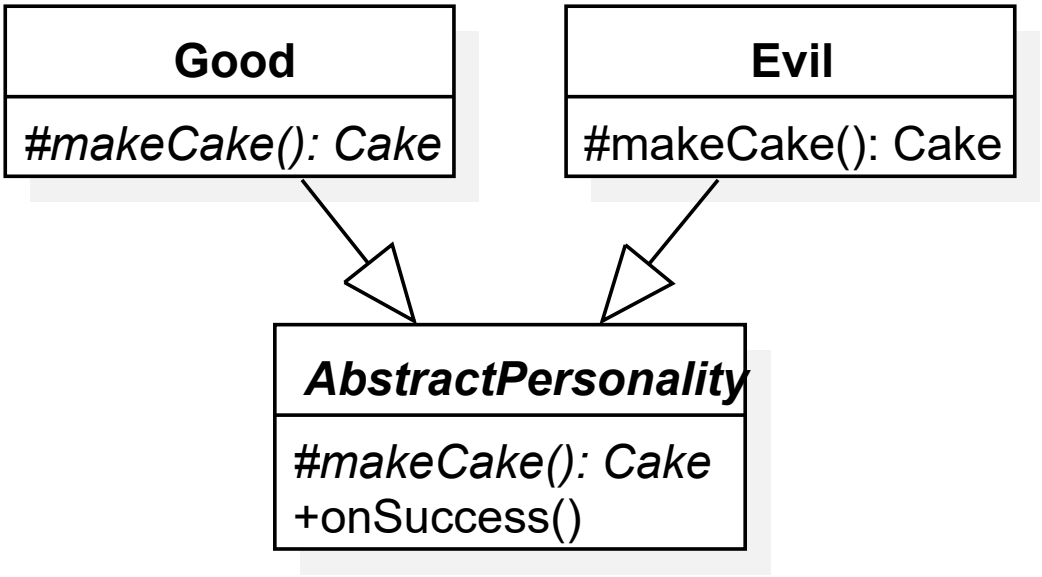

Figura 2.3: Rappresentazione UML dell'applicazione del pattern Template Method alla gerarchia delle Personalità

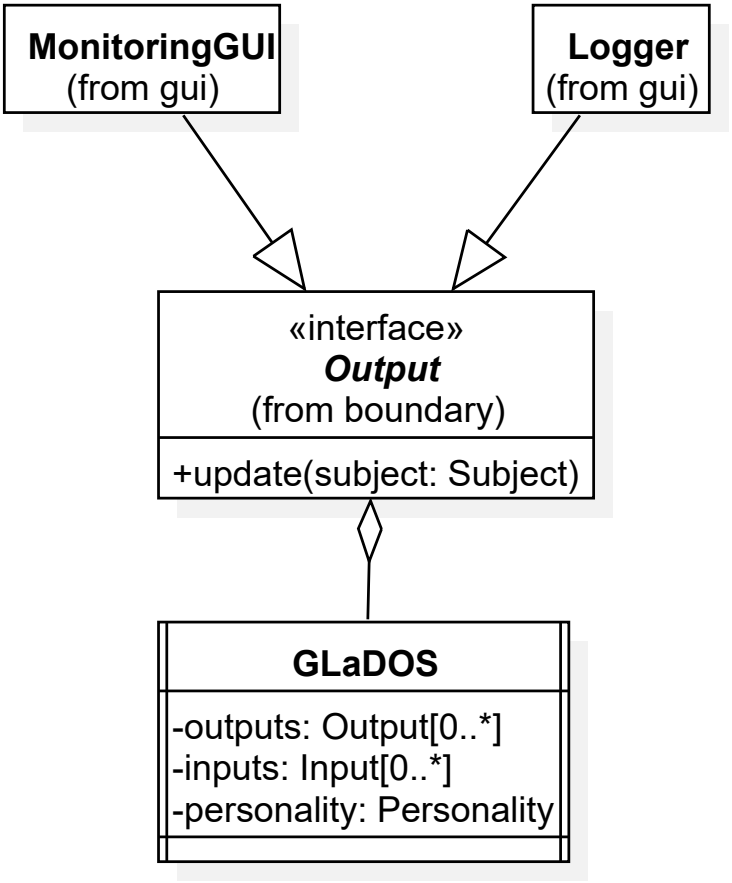

Figura 2.4: Il pattern Observer è usato per consentire a GLaDOS di informare tutti i sistemi di output in ascolto

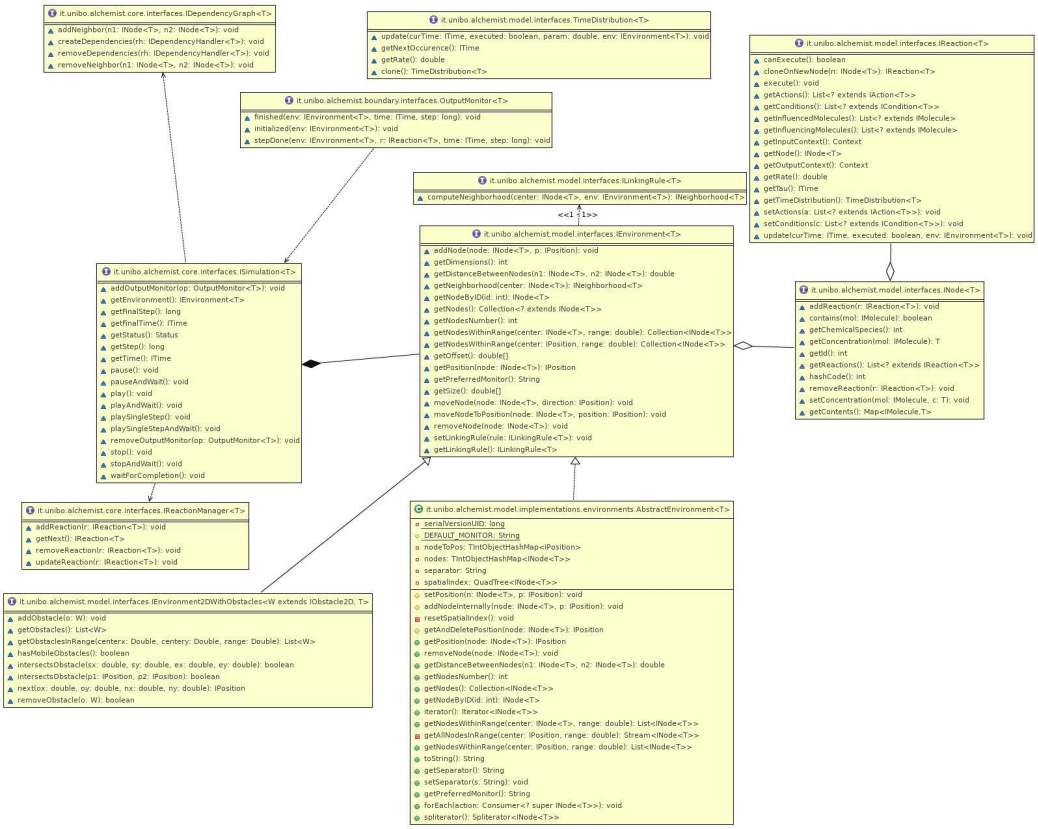

Figura 2.5: Schema UML mal fatto e con una pessima descrizione, che non aiuta a capire. Don't try this at home.

Figura 2.6: Schema UML del *Model* relativo ai *Characters*.

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

- StaticObstacleTest
- DynamicObstacleTest
- ObstacleFactoryTest
- SpeedVector2DTest

- Dimension2DTest
- PickupItemTest
- ItemFactoryTest
- MainMenuControllerTest

Leon Baiocchi

- todo : scrivere le classi di test utilizzate

Federico Brunelli

- todo : scrivere le classi di test utilizzate

Luca Rengo

- todo : scrivere le classi di test utilizzate

3.2 Metodologia di lavoro

Ci aspettiamo, leggendo questa sezione, di trovare conferma alla divisione operata nella sezione del design di dettaglio, e di capire come è stato svolto il lavoro di integrazione. **Andrà realizzata una sotto-sezione separata per ciascuno studente** che identifichi le porzioni di progetto sviluppate, separando quelle svolte in autonomia da quelle sviluppate in collaborazione. Diversamente dalla sezione di design, in questa è consentito elencare package/classi, se lo studente ritiene sia il modo più efficace di convogliare l'informazione. Si ricorda che l'impegno deve giustificare circa 40-50 ore di sviluppo (è normale e fisiologico che approssimativamente la metà del tempo sia impiegata in analisi e progettazione).

Elementi positivi

- Si identifica con precisione il ruolo di ciascuno all'interno del gruppo, ossia su quale parte del progetto ciascuno dei componenti si è concentrato maggiormente.
- La divisione dei compiti è equa, ossia non vi sono membri del gruppo che hanno svolto molto più lavoro di altri.

- La divisione dei compiti è coerente con quanto descritto nelle parti precedenti della relazione.
- La divisione dei compiti è realistica, ossia le dipendenze fra le parti sviluppate sono minime.
- Si identifica quale parte del software è stato sviluppato da tutti i componenti insieme.
- Si spiega in che modo si sono integrate le parti di codice sviluppate separatamente, evidenziando eventuali problemi. Ad esempio, una strategia è convenire sulle interfacce da usare (ossia, occuparsi insieme di stabilire l'architettura) e quindi procedere indipendentemente allo sviluppo di parti differenti. Una possibile problematica potrebbe essere una dimenticanza in fase di design architetturale che ha costretto ad un cambio e a modifiche in fase di integrazione. Una situazione simile è la norma nell'ingegneria di un sistema software non banale, ed il processo di progettazione top-down con raffinamento successivo è il così detto processo "a spirale".
- Si descrive in che modo è stato impiegato il DVCS.

Elementi negativi

- Non si chiarisce chi ha fatto cosa.
- C'è discrepanza fra questa sezione e le sezioni che descrivono il design dettagliato.
- Tutto il progetto è stato svolto lavorando insieme invece che assegnando una parte a ciascuno.
- Non viene descritta la metodologia di integrazione delle parti sviluppate indipendentemente.
- Uso superficiale del DVCS.

3.3 Note di sviluppo

Questa sezione, come quella riguardante il design dettagliato va svolta **singolarmente da ogni membro del gruppo**.

Ciascuno dovrà mettere in evidenza eventuali particolarità del suo metodo di sviluppo, ed in particolare:

- **Elencare** (fare un semplice elenco per punti, non un testo!) le feature *avanzate* del linguaggio e dell'ecosistema Java che sono state utilizzate. Le feature di interesse sono:

  - Progettazione con generici, ad esempio costruzione di nuovi tipi generici, e uso di generici bounded. Uso di classi generiche di libreria non è considerato avanzato.
  - Uso di lambda expressions
  - Uso di `Stream`, di `Optional` o di altri costrutti funzionali
  - Uso della reflection
  - Definizione ed uso di nuove annotazioni
  - Uso del Java Platform Module System
  - Uso di parti di libreria non spiegate a lezione (networking, compressione, parsing XML, eccetera...)
  - Uso di librerie di terze parti (incluso JavaFX): Google Guava, Apache Commons...
  - Uso di build systems

  Si faccia molta attenzione a non scrivere banalità, elencando qui features di tipo "core", come le eccezioni, le enumerazioni, o le inner class: nessuna di queste è considerata avanzata.

- Descrivere *molto brevemente* le librerie utilizzate nella propria parte di progetto, se non trattate a lezione (ossia, se librerie di terze parti e/o se componenti del JDK non visti, come le socket). Si ricorda che l'utilizzo di librerie è valutato *positivamente*.

- Sviluppo di algoritmi particolarmente interessanti *non forniti da alcuna libreria* (spesso può convenirvi chiedere sul forum se ci sia una libreria per fare una certa cosa, prima di gettarvi a capofitto per scriverla voi stessi).

In questa sezione, *dopo l'elenco*, è anche bene evidenziare eventuali pezzi di codice "riadattati" (o scopiazzati...) da Internet o da altri progetti, pratica che tolleriamo ma che non raccomandiamo. I pattern di design, invece **non** vanno messi qui. L'uso di pattern di design (come suggerisce il nome) è un aspetto avanzato di design, non di implementazione, e non va in questa sezione.

Elementi positivi

- Si elencano gli aspetti avanzati di linguaggio che sono stati impiegati
- Si elencano le librerie che sono state utilizzate
- Si descrivono aspetti particolarmente complicati o rilevanti relativi all'implementazione, ad esempio, in un'applicazione performance critical, un uso particolarmente avanzato di meccanismi di caching, oppure l'implementazione di uno specifico algoritmo.
- Se si è utilizzato un particolare algoritmo, se ne cita la fonte originale. Ad esempio, se si è usato Mersenne Twister per la generazione dei numeri pseudo-random, si cita [**?**].
- Si identificano parti di codice prese da altri progetti, dal web, o comunque scritte in forma originale da altre persone. In tal senso, si ricorda che agli ingegneri non è richiesto di re-inventare la ruota continuamente: se si cita debitamente la sorgente è tollerato fare uso di di snippet di codice per risolvere velocemente problemi non banali. Nel caso in cui si usino snippet di codice di qualità discutibile, oltre a menzionarne l'autore originale si invitano gli studenti ad adeguare tali parti di codice agli standard e allo stile del progetto. Contestualmente, si fa presente che è largamente meglio fare uso di una libreria che copiarsi pezzi di codice: qualora vi sia scelta (e tipicamente c'è), si preferisca la prima via.

Elementi negativi

- Si elencano feature core del linguaggio invece di quelle segnalate. Esempi di feature core da non menzionare sono:
  - eccezioni;
  - classi innestate;
  - enumerazioni;
  - interfacce.
- Si elencano applicazioni di terze parti (peggio se per usarle occorre licenza, e lo studente ne è sprovvisto) che non c'entrano nulla con lo sviluppo, ad esempio:
  - Editor di grafica vettoriale come Inkscape o Adobe Illustrator;

  - Editor di grafica scalare come GIMP o Adobe Photoshop;
  - Editor di audio come Audacity;
  - Strumenti di design dell'interfaccia grafica come SceneBuilder: il codice è in ogni caso inteso come sviluppato da voi.

- Si descrivono aspetti di scarsa rilevanza, o si scende in dettagli inutili.

- Sono presenti parti di codice sviluppate originalmente da altri che non vengono debitamente segnalate. In tal senso, si ricorda agli studenti che i docenti hanno accesso a tutti i progetti degli anni passati, a Stack Overflow, ai principali blog di sviluppatori ed esperti Java (o sedicenti tali), ai blog dedicati allo sviluppo di soluzioni e applicazioni (inclusi blog dedicati ad Android e allo sviluppo di videogame), nonché ai social network. Conseguentemente, è *molto* conveniente *citare* una fonte ed usarla invece di tentare di spacciare per proprio il lavoro di altri.

- Si elencano design pattern

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

È richiesta una sezione per ciascun membro del gruppo, obbligatoriamente. Ciascuno dovrà autovalutare il proprio lavoro, elencando i punti di forza e di debolezza in quanto prodotto. Si dovrà anche cercare di descrivere *in modo quanto più obiettivo possibile* il proprio ruolo all'interno del gruppo. Si ricorda, a tal proposito, che ciascuno studente è responsabile solo della propria sezione: non è un problema se ci sono opinioni contrastanti, a patto che rispecchino effettivamente l'opinione di chi le scrive. Nel caso in cui si pensasse di portare avanti il progetto, ad esempio perché effettivamente impiegato, o perché sufficientemente ben riuscito da poter esser usato come dimostrazione di esser capaci progettisti, si descriva brevemente verso che direzione portarlo.

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