Progetto di Programmazione ad Oggetti
"The Exiled"

Luca Casadei, Francesco Pazzaglia, Marco Magnani, Manuel Baldoni

11 febbraio 2024

Indice

Capitolo 1

Analisi

1.1 Requisiti

L'applicazione è un gioco che presenta un personaggio controllato dal giocatore con la possibilità di muoversi nella mappa in 4 direzioni e di combattere contro dei nemici utilizzando magie elementali. I nemici sconfitti potrebbero rilasciare delle cure o dei potenziamenti che favoriscono il giocatore. Per concludere il gioco, il giocatore deve entrare in possesso di 4 cristalli che vengono consegnati una volta sconfitti i 4 boss del gioco che sono nemici più difficili da sconfiggere.

Requisiti funzionali

- Movimento del giocatore.
- Movimento dei nemici.
- Presenza di oggetti ottenibili dal giocatore (Cure, potenziamenti e cristalli).
- Terminazione del gioco una volta raccolti 4 cristalli o se il giocatore viene sconfitto.
- Posizionamento e distribuzione degli oggetti.
- Possibilità di fare battaglie tra giocatore e nemici.
- Nemici più forti (boss) che se sconfitti rilasciano i cristalli per terminare il gioco.

- Possibilità di utilizzo di magie in battaglia di diverso tipo (Fuoco, Acqua, Fulmine, Erba).
- Aumento di livello tramite guadagno di esperienza sconfiggendo i nemici.

Requisiti non funzionali

- Il gioco deve essere multipiattaforma.
- *TODO*

1.2 Analisi e modello del dominio

All'inizio al giocatore viene chiesto di che tipo elementale sarà il personaggio da controllare, in base a questa scelta gli verrà quindi assegnata una mossa di base di quel tipo e avrà le mosse di quel tipo potenziate fino alla fine del gioco.
Il giocatore può muoversi di una cella alla volta in una mappa a griglia in 4 direzioni (Nord, Est, Sud, Ovest) e una battaglia con i nemici (anch'essi di un certo tipo elementale) inizia quando la cella in cui è il giocatore combacia con quella di un nemico.
Per affrontare un nemico in battaglia il giocatore ha a disposizione massimo 4 mosse (non per forza dello stesso tipo del giocatore). Il combattimento avviene a turni alternati, il primo è il giocatore, poi il nemico e così via finché uno dei due viene sconfitto, se è il giocatore, il gioco termina.
Allo sconfiggere dei nemici viene conferita al giocatore una certa quantità di esperienza che serve ad aumentare di livello e un oggetto casuale, l'oggetto viene salvato nell'inventario del giocatore e potrà essere di diverso tipo, es. oggetto curativo(ripristina un tot di vita), booster di statistiche(per esempio aumenta l'attacco di un certo tipo di mosse) ecc.. L'aumento di livello comporta un incremento generale delle statistiche ovvero attacco, difesa e vita di un valore costante. Ogni 5 livelli verrà presentata al giocatore la possibilità di imparare una nuova mossa di un tipo casuale, se il giocatore ha già 4 mosse potrà decidere di scambiare quella nuova con una di quelle che conosce già. All'aumentare di livello sarà richiesta sempre più esperienza per passare a quello successivo.
Ci sono diverse classi di nemico, alcuni più deboli e altri più forti, come i boss, che se sconfitti consegnano uno dei 4 oggetti necessari per concludere il gioco (i cristalli).

1.2.1 Tipi elementali

I seguenti sono i tipi elementali presenti, ognuno è efficace rispetto ad un altro, quindi quando vengono usate mosse di un certo tipo che risulta essere efficace rispetto al tipo di un nemico, si avranno dei moltiplicatori del danno. Lo stesso vale per i nemici che usano mosse di un tipo efficace rispetto a quello scelto dal giocatore all'inizio.

Ad esempio, nel seguente schema si vede che Fulmine è efficace contro Acqua.

Fulmine	$\rightarrow$	Acqua
Erba	$\rightarrow$	Fulmine
Acqua	$\rightarrow$	Fuoco
Fuoco	$\rightarrow$	Erba

1.2.2 UML

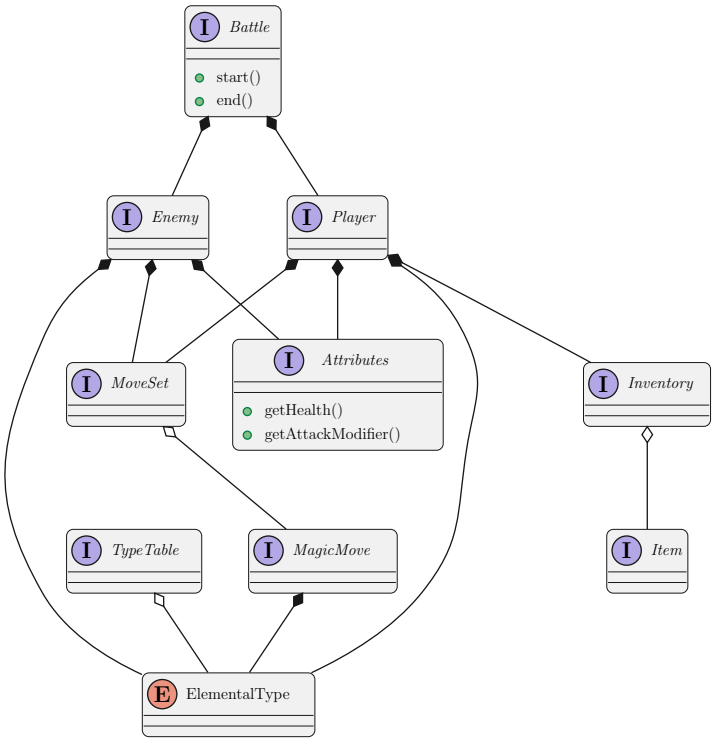

Figura 1.1: Schema UML del dominio.

Capitolo 2

Design

Qui entriamo nel dettaglio della struttura ad alto livello del gioco.

2.1 Architettura

2.2 Design dettagliato

Luca Casadei

La parte di creazione dei nemici è stata realizzata utilizzando il pattern creazionale "Abstract Factory"

Francesco Pazzaglia

La mia parte di progetto si concentra sull'implementazione dettagliata del giocatore, comprendendo la definizione delle sue caratteristiche quali attributi(la vita, l'esperienza, la relativa classe...) e le sue meccaniche in gioco. Per facilitare la comunicazione tra il Model del giocatore (*Player*) e la View è stato introdotto un Controller garantendo l'utilizzo del pattern *MVC*.

Mi sono anche occupato della gestione del movimento del giocatore e delle altre entità (i nemici) all'interno della mappa di gioco. Pertanto, anche in questo caso è stato fatto uso di Controller per consentire la visualizzazione dei nemici in movimento sulla mappa.

Infine, ho implementato la gestione delle condizioni di chiusura del gioco(vittoria e game over), che includevano la definizione delle regole e dei possibili eventi che portavano al termine del gioco, nonché la gestione delle rispettive schermate, come la chiusura del gioco o la possibilità di riavviare una nuova partita.

Player

Nell'ambito del progetto del gioco, ho osservato la presenza di caratteristiche comuni tra le diverse entità presenti. Le caratteristiche di base del giocatore sono condivise anche dai vari nemici. Pertanto, in collaborazione con *Luca Casadei*, responsabile della parte relativa ai nemici, abbiamo deciso di introdurre nel Model un'interfaccia denominata *GameCharacter*. Questa interfaccia contiene i metodi principali che definiscono le caratteristiche fondamentali comuni a tutte le entità nel gioco, come la vita, la posizione, il movimento e gli attributi. In aggiunta all'interfaccia, è stata implementata la classe astratta *GameCharacterImpl*, che concretizza l'interfaccia *GameCharacter*. Questa classe è stata progettata in modo che il giocatore, possa estenderla e acquisire le caratteristiche comuni menzionate in precedenza. Questa architettura ci ha consentito di gestire in modo efficiente le proprietà condivise tra il giocatore e i nemici, garantendo una maggiore coerenza e facilità di estensione del codice senza ripetizioni.

Nella gestione del giocatore, ho introdotto un'interfaccia denominata *Player*, che estende l'interfaccia preesistente *GameCharacter*. Successivamente, ho realizzato l'implementazione di questa interfaccia attraverso una classe concreta chiamata *PlayerImpl*. Tale procedura mi ha consentito di definire un comportamento specifico, mantenendo al contempo la flessibilità per estendere e personalizzare le funzionalità di base fornite dall'interfaccia *GameCharacter*.

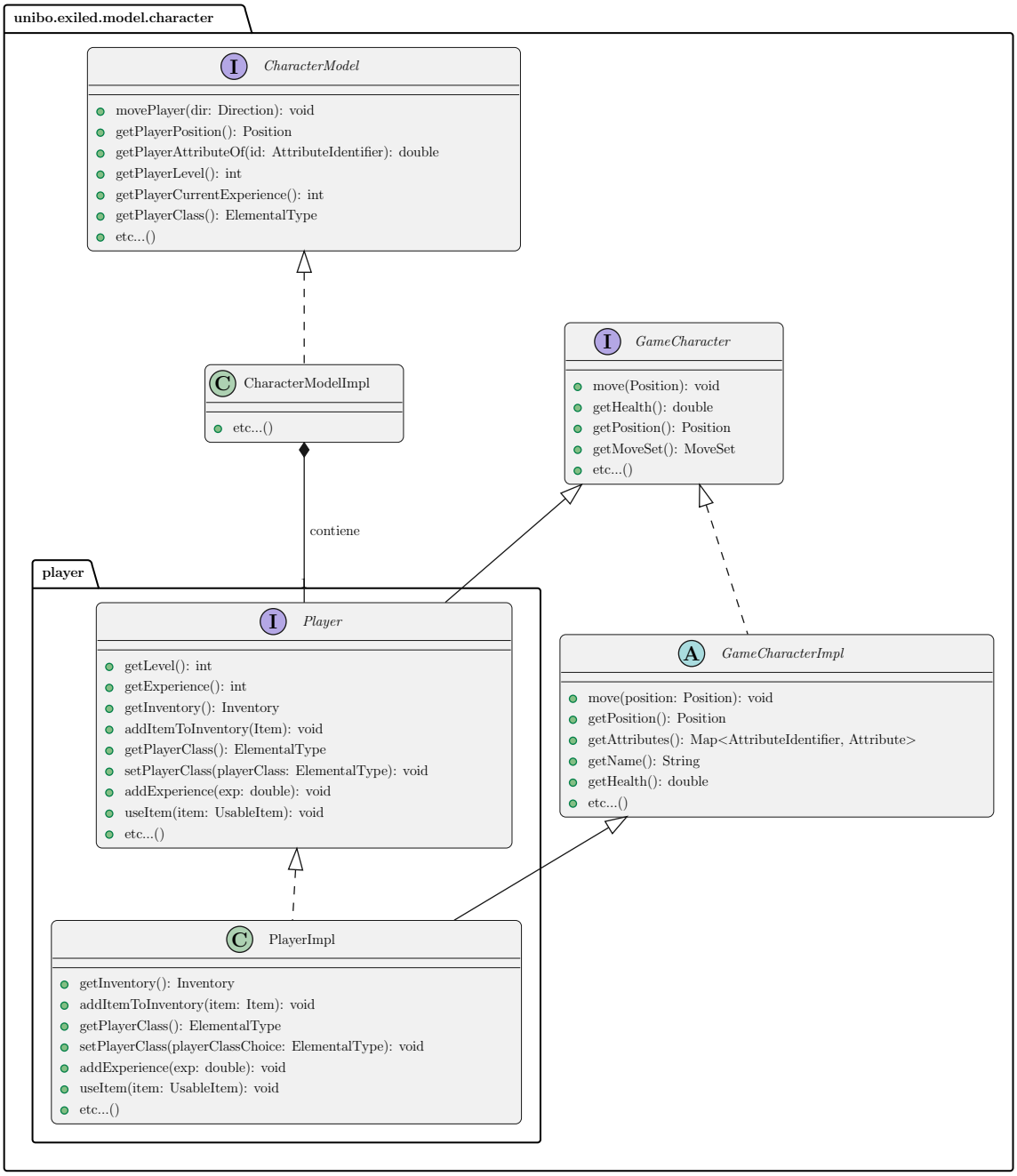

Figura 2.1: Schema UML del Model del Player e del GameCharacter

Nella parte inerente alla View sia il giocatore che i nemici sono stati gestiti attraverso una classe chiamata *CharacterView*. Questa classe è stata progettata per utilizzare immagini al fine di rappresentare graficamente le

entità del gioco. In particolare, sono state gestite delle animazioni mediante l'utilizzo di due immagini per ciascuna direzione (NORD, SUD, EST e OVEST). Questo approccio è stato adottato per migliorare l'aspetto visivo del gioco e rendere più coinvolgente l'esperienza di gioco.

La *CharacterView* è una estensione di *JLabel* che gestisce l'aspetto visivo dei personaggi nel gioco. All'interno di questa classe, viene impostata l'immagine che la *JLabel* deve rappresentare, selezionata tra quelle disponibili nelle risorse del gioco. Inizialmente, per il player, viene scelta un'immagine in base alla sua classe, e lo stesso principio si applica ai nemici. Questa classe ausiliaria ci ha permesso di garantire una gestione ottimale della scalabilità della griglia di gioco durante il ridimensionamento della finestra e ci ha permesso inoltre effettuare efficacemente il cambio di immagine per le animazioni.

Per garantire la comunicazione tra il Model e la View del *Player* è stato introdotto un Controller, sotto forma di interfaccia, denominato *CharacterController*. La relativa classe di implementazione invece è stata chiamata *CharacterControllerImpl* dove, al suo interno sono stati definiti i metodi per il controllo del *Player*, tra cui quelli relativi al movimento, all'assegnamento della classe(del tipo), all'uso di modificatori di esperienza e di livello, nonché i relativi metodi Getter necessari per l'interfaccia grafica.

Nel *CharacterController* sono stati definiti metodi generali applicabili a qualsiasi entità presente nel gioco, sia un giocatore o nemico. Per tale motivo, è stata introdotta un'ulteriore interfaccia denominata *CharacterModel*, accompagnata dalla relativa classe *CharacterModelImpl*. Questa struttura consente al controller di gestire entrambe le tipologie di entità in modo uniforme e coerente.

Il design pattern che viene utilizzato per gestire la parte del *Player* è Template Method. Questo pattern consente di definire uno scheletro di comportamento comune per il *Player*. La classe astratta *GameCharacterImpl* costituisce la base per tutte le entità di gioco, ad esempio definisce il metodo *move(Position)*, ossia la struttura generica per il movimento di un'entità. L'interfaccia *Player*, che estende *GameCharacter*, definisce ulteriori metodi specifici del giocatore per il livello, l'esperienza, l'inventario ecc. La classe *PlayerImpl* implementa l'interfaccia *Player* e fornisce le implementazioni specifiche dei metodi definiti nell'interfaccia. Questo approccio favorisce la riutilizzabilità del codice e facilita l'estensione delle funzionalità del *Player* senza dover modificare la struttura di base definita nella classe astratta *GameCharacterImpl*.

Gestione del movimento

Nell'ambito dell'implementazione del movimento delle entità all'interno della mappa di gioco, ho adottato due distinzioni principali. Per il movimento del giocatore ho progettato un sistema basato sull'input dell'utente tramite tastiera. D'altra parte, per i nemici, ho sviluppato un meccanismo autonomo di movimento. In pratica, i nemici si spostano in maniera pseudo-casuale all'interno di un'area definita dalla loro tipologia(Fire, Bolt, Water e Grass) e qualora il giocatore si dovesse avvicinare, entro un certo raggio, ad un nemico, quest'ultimo inizierà a inseguirlo per combattere. Questo comportamento è stato realizzato attraverso due metodi: uno per calcolare la distanza tra il giocatore e un nemico (il metodo privato *calculateDistance*), e un altro per determinare la direzione in cui il nemico deve muoversi per inseguire il giocatore (il metodo privato *calculateChaseDirection*). Questo approccio consente un comportamento dinamico e coinvolgente per i nemici, rendendo il gioco più interessante e stimolante per il giocatore, dato che per poter completare il gioco bisogna sconfiggere i quattro Boss che sono situati agli estremi della mappa di gioco. Considerando la decisione di non impiegare i thread nel nostro progetto, i nemici si sposteranno esclusivamente in risposta all'input dell'utente. Di conseguenza, i nemici si muoveranno solamente quando l'utente eseguirà uno spostamento nella mappa.

In particolare, ho utilizzato il *CharacterModel* per definire il meccanismo di movimento sia per il personaggio che per i nemici. Successivamente, all'interno del *CharacterController*, ho implementato un metodo di movimento che ha richiamato le funzioni precedentemente definite nel Model. Questa scelta è stata fatta per mantenere la separazione delle responsabilità e per organizzare il movimento applicando il pattern *MVC*. Inoltre, nella classe *GameKeyListener* della View, ho gestito l'input dell'utente associando i movimenti alle direzioni corrispondenti ai tasti premuti (ad esempio, W per su, S per giù, A per sinistra e D per destra). La *GameView* rappresenta la schermata principale dov'è inserita la mappa e l'HUD che visualizza gli attributi principali del personaggio. Quando l'utente preme un tasto specifico sulla tastiera tra quelli elencati prima, all'interno della *GameView*, il *CharacterView* associato al giocatore si sposterà in una cella corrispondente. Di conseguenza, come precedentemente descritto, tutti i nemici, a meno che non siano vicini al giocatore, si sposteranno anch'essi, in una direzione scelta in modo pseudo-casuale.

Oltre ai KeyListener di base per il movimento del giocatore, ho integrato la funzionalità che permette di utilizzare il tasto E per aprire l'inventario quando si è in gioco e il tasto ESC per aprire/chiudere il menu di gioco. Questa scelta è stata fatta per migliorare l'esperienza di gioco e rendere

l'interfaccia più intuitiva e user-friendly, consentendo ai giocatori di navigare facilmente tra le diverse funzionalità senza interruzioni.

Avendo separato la gestione del movimento e dell'ascolto dei *KeyListener* in una classe dedicata, è stato adottato il design pattern Observer. Questo pattern ha consentito di definire un meccanismo di notifica personalizzato attraverso la classe *GameKeyListener*, che funge da Observer. *GameKeyListener* implementa l'interfaccia *KeyListener* e si occupa di ricevere notifiche sugli eventi di pressione dei tasti. In questo modo, sono stato in grado di gestire dinamicamente il movimento delle entità di gioco in risposta all'input dell'utente.

Meccanismo di fine gioco

Mi sono occupato di gestire la conclusione del gioco attraverso due scenari distinti: la vittoria(tramite la *GameVictoryView*) e la sconfitta(tramite *GameOverView*).

Nella parte della sconfitta ho realizzato la gestione del game over attraverso il *GameController*. Utilizzando il metodo *isOver()*, viene deciso quando il gioco è terminato e di conseguenza quando deve essere visualizzata la *GameOverView* dedicata. Avendo un unico metodo per determinare il game over, ho potuto facilmente definire il suo criterio. Essendo un gioco centrato su un singolo giocatore principale, è stato naturale impostare il game over nel caso in cui la sua vita dovesse azzerarsi o scendere sotto lo zero, a seguito di un combattimento con un nemico.

Lo stesso principio si applica alla GameVictoryView, la quale rappresenta la schermata che compare quando il gioco è completato con successo. Questa View viene attivata quando il metodo *isWon()* restituisce un valore che indica se il giocatore ha vinto il gioco. Per poter vincere il gioco, il personaggio deve avere nel suo inventario i quattro cristalli della redenzione ottenuti sconfiggendo i Boss finali.

All'interno delle due schermate, sono stati inclusi due *JButton* con relativi *ActionListener*: uno permette di iniziare una nuova partita e l'altro serve per chiudere definitivamente il gioco.

La *GameOverView* e la *GameVictoryView* sono state integrate all'interno della *GameView*, la schermata principale definita in precedenza. All'interno della sua classe è contenuto il campo del *GameController*, che facilita il collegamento tra il Model e la View stessa. Grazie a questo Controller, è possibile determinare quando la schermata principale deve essere sostituita dalla schermate di vincita o perdita del gioco.

Questo approccio ha semplificato la gestione delle schermate di fine gioco, utilizzando una logica chiara e intuitiva nella scrittura del codice.

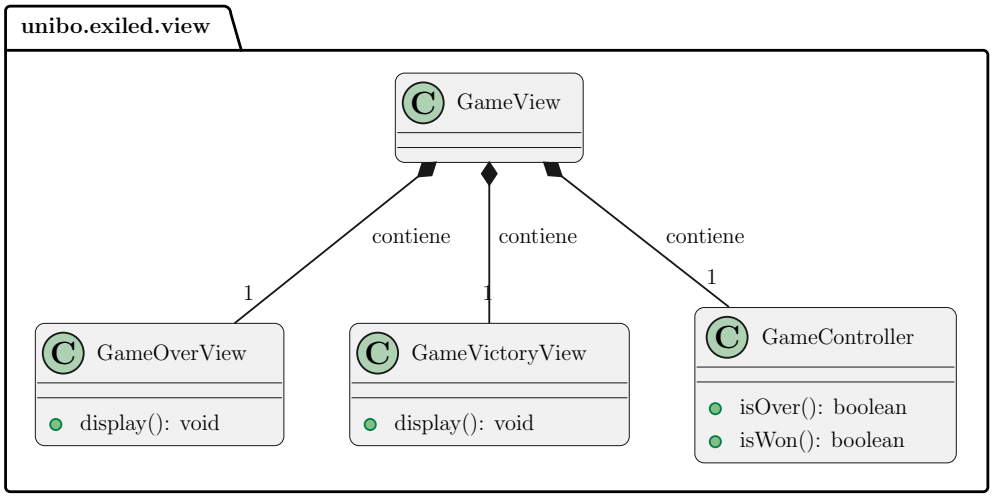

Figura 2.2: Schema UML del GameOver con relativa vista sulle View

Il design pattern applicato alla parte di *GameOverView* e *GameVictoryView*, è il pattern comportamentale Strategy. Le due View rappresentano due strategie diverse per gestire la visualizzazione dell'esito del gioco (game over o vittoria). La classe *GameView* utilizza queste strategie decidendo quale delle due visualizzare in base alle condizioni definite nei metodi *isOver()* e *isWon()* del *GameController*. Utilizzando lo Strategy Pattern, è possibile cambiare dinamicamente il comportamento di *GameView* sostituendo una strategia con un'altra senza modificare il suo codice.

Marco Magnani

Manuel Baldoni

Capitolo 3

Sviluppo

3.1 Testing automatizzato

3.2 Note di sviluppo

Luca Casadei

Francesco Pazzaglia

- **Espressioni Lambda**: Utilizzate nei bottoni che hanno *ActionListener* nella *GameOverView*.
- **Optional**: Utilizzati per sostituire i *null value*, usati nel *CharacterModel*(metodo *getPlayer()*) e anche per il passaggio della *GameView* tra la differenti viste.
- **Stream**: Utilizzati nei Controller per controllare che il *Player* abbia vinto il gioco.

Marco Magnani

Manuel Baldoni

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Luca Casadei

Francesco Pazzaglia

Nonostante le parti complesse del progetto e il fatto di sviluppare un progetto in team mi avesse inizialmente intimorito, sono soddisfatto di come siamo riusciti a completare e consegnare il gioco. Di solito sono abituato a lavorare da solo, ma ho trovato la collaborazione in team piuttosto soddisfacente. Lo scambio di idee e il confronto ci hanno aiutato a coordinarci efficacemente, individuando i metodi e le classi necessarie per varie parti del progetto. Lavorare in team è stato molto formativo e mi ha permesso di apprezzare come si possa realizzare un progetto insieme, nonostante stili di programmazione talvolta diversi.

Marco Magnani

Manuel Baldoni

4.2 Difficoltà incontrate e commenti per i docenti

Appendice A

Guida utente

Appendice B

Esercitazioni di laboratorio

Elenco dei Permalink delle consegne nel forum studenti del modulo di laboratorio.

Consegne

B.0.1 luca.casadei27@studio.unibo.it

- Laboratorio 06: `https://virtuale.unibo.it/mod/forum/discuss.php?d=146511#p208401`
- Laboratorio 07: `https://virtuale.unibo.it/mod/forum/discuss.php?d=147598#p209323`
- Laboratorio 08: `https://virtuale.unibo.it/mod/forum/discuss.php?d=148025#p209782`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=149231#p211336`
- Laboratorio 10: `https://virtuale.unibo.it/mod/forum/discuss.php?d=150252#p212475`