

JAVA MYSTERY DUNGEON

INDICE

-ANALISI.....	2
1.1 Descrizione e requisiti.....	3
1.2 Modello del Dominio.....	3
-DESIGN.....	4
2.1 Architettura.....	5
2.2 Design dettagliato.....	9
Laura Bertozzi.....	9
Kelly Applebee.....	12
Alice Beccati.....	15
Matteo Monari.....	19
-SVILUPPO.....	22
3.1 Testing automatizzato.....	22
3.2 Note di sviluppo.....	24
Laura Bertozzi.....	24
Applebee Kelly.....	24
ALICE BECCATI.....	25
MATTEO MONARI.....	25
-COMMENTI FINALI.....	26
4.1 Autovalutazione e lavori futuri.....	26
Laura Bertozzi.....	26
Applebee Kelly.....	27
Alice Beccati.....	27
Monari Matteo.....	28
4.2 Difficoltà incontrate e commenti per i docenti.....	28
Alice Beccati.....	29
Kelly Applebee.....	29
Matteo Monari.....	29
Appendice A - Guida utente.....	30
Appendice B - Esercitazioni di laboratorio.....	32
B.0.1 alice.beccati@studio.unibo.it.....	33

CAPITOLO 1

-ANALISI

1.1 Descrizione e requisiti

Nel regno di Hyrule, l'eroe Link è incaricato di scendere nelle profondità di un dungeon multipiano, raggiungere il drago e sconfiggerlo per salvare il regno.

L'utente guiderà Link attraverso stanze e corridoi, affrontando nemici pronti all'attacco e raccogliendo pozioni dai diversi effetti benefici utili alla progressione.

Tutti i piani che compongono il dungeon differiscono uno dall'altro; si considera completato quando vengono trovate le scale che conducono al livello successivo senza possibilità di tornare indietro.

Durante i combattimenti con i nemici il giocatore può scegliere mosse con statistiche differenti, ciascuna caratterizzata da un costo in energia e da un ammontare di danno diverso.

Nel punto più remoto del dungeon attende il mostro più potente (boss finale): una volta sconfitto, Link recupererà la Triforza rubata, ristabilendo pace e armonia nel regno.

REQUISITI FUNZIONALI:

- Creazione di nemici diversificati.
- Creazione di oggetti di cura e power-up.
- Interfaccia grafica per mappa, combattimento, borsa/inventario.
- Generazione procedurale delle mappe su griglia.
- Mappa con avanzamento di piano (scale).
- Combattimento a turni con mosse a costo di energia e danni diversi.
- Game Over (schermata e gestione restart/uscita).
- Avanzamento livello del protagonista (aumento vita/stat).

REQUISITI NON FUNZIONALI:

- l'applicazione deve essere portatile, quindi funzionare su Windows, Linux e macOS.
- Il programma deve essere compressibile in un JAR con tutte le risorse necessarie.
- le finestre devono essere ridimensionabili
- generazione lazy dei piani del dungeon

1.2 Modello del Dominio

Il dominio prevede un mondo di gioco costituito da un singolo dungeon esplorabile, composto da diversi piani, ognuno dei quali si compone di stanze, tunnel e muri.

All'interno delle stanze possono essere presenti diverse entità: i nemici e gli oggetti, utili a facilitare la progressione.

I nemici possono appartenere a diversi tipi, differenziati dal loro comportamento (ad esempio fermi, in movimento regolare o inseguono il giocatore), in modo da introdurre varietà nell'esperienza di gioco.

Le pozioni sono oggetti raccoglibili che vengono conservate nell'inventario del giocatore e possono essere utilizzate durante il combattimento.

Il giocatore può muoversi sul campo soltanto nelle 4 direzioni consentite, attraverso le stanze e i tunnel. Insieme a lui, ad ogni movimento, effettueranno un passo anche i nemici.

Per ogni piano sarà presente solo una scala per avanzare al piano successivo, dopo il quale non sarà più possibile tornare indietro.

La stanza al piano finale ospita il boss: sconfiggerlo consente di completare l'avventura.

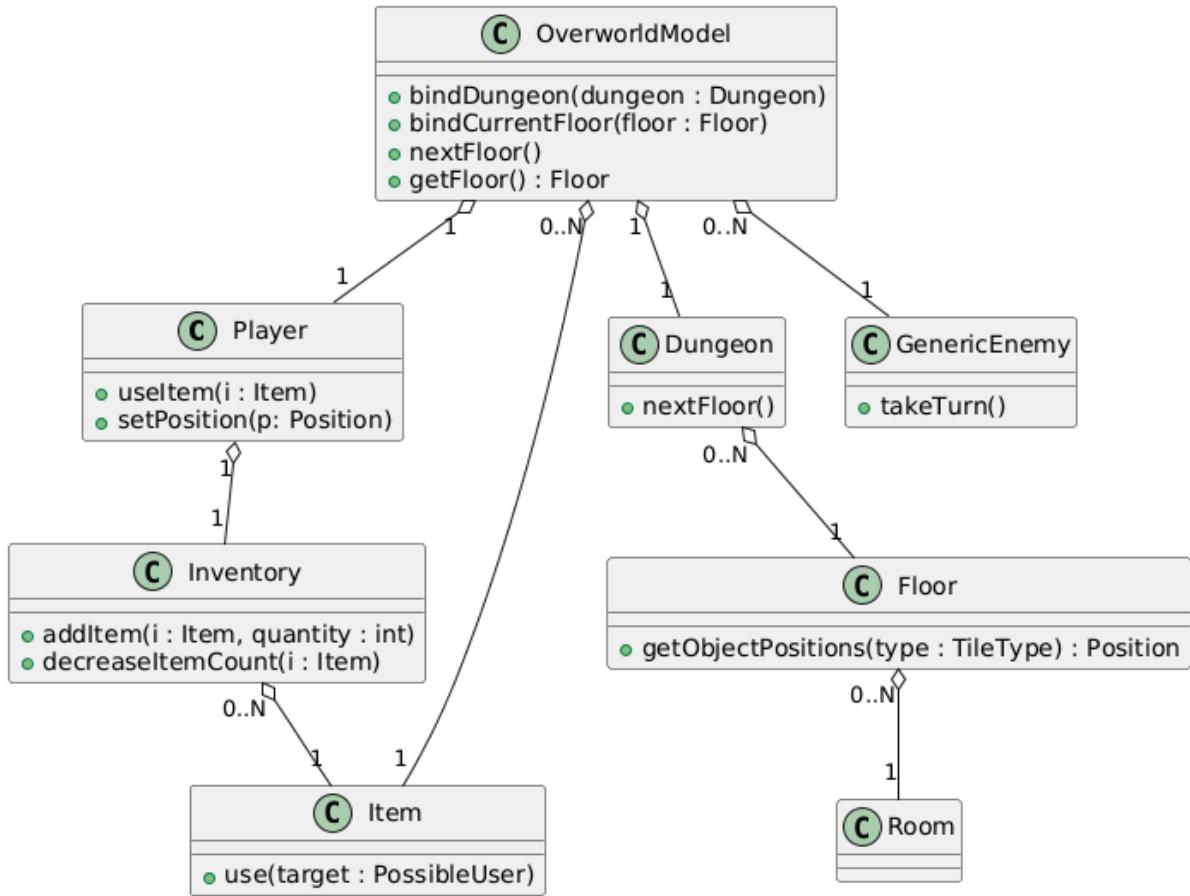


Figura 1.1: Schema UML dell’analisi del problema, con rappresentate le principali entità ed il rapporto tra loro

CAPITOLO 2

-DESIGN

2.1 Architettura

L’architettura di *JavaMysteryDungeon* utilizza il pattern architettonico **MVC**.

Più nello specifico, a livello architettonale, si è scelto di adottare **MVC in forma HMVC (Hierarchical Model–View–Controller)**: questa variante prevede la presenza di un **Manager** che ha il compito di gestire e indirizzare la vista corretta da presentare all’utente in base allo stato

corrente del gioco (schermata iniziale, mappa dell'overworld, inventario, combattimento, game over, vittoria).

Ogni vista principale è quindi organizzata come un sottosistema indipendente che adotta il pattern **MVC** o **MVP**, a seconda delle esigenze.

Overworld (MVC)

La parte di **Overworld** adotta il pattern MVC in forma “pura”:

- **Model**

- Rappresenta lo stato persistente dell'overworld, come griglia di base, entità presenti (giocatore, nemici, oggetti, scale), piani e stanze.
- Contiene anche la logica di dominio, come la gestione dei movimenti, della generazione dei piani e delle collisioni.
- Espone interfacce in sola lettura e un sistema di **notifica** per aggiornare la vista senza rivelare dettagli implementativi.

- **View**

- Si occupa unicamente della **rappresentazione grafica**.

- **Controller**

- Si occupa della **mediazione** fra input dell'utente e modello e tra modello e view (OverworldController e MapController).
- Riceve comandi (es. movimenti), aggiorna lo stato del modello (posizione del giocatore, cambio piano).

Combattimento (MVP)

La parte di **Combattimento** adotta il pattern **MVP** (**Model–View–Presenter**).

In questo approccio la **view** è concepita come componente **passiva**: mostra solo i dati che riceve e inoltra gli input utente al presenter, con una logica di presentazione molto ridotta (anche se attualmente vi sono alcuni costrutti minori per aspetti puramente visivi).

- **Model**

- Gestisce lo stato e la logica del combattimento:
caratteristiche dei personaggi, posizioni, mosse e attacchi.

- **View**

- Si limita a mostrare graficamente lo stato del combattimento
e a raccogliere input dall'utente.
- È rappresentata da CombatViewInterface (interfaccia) e
CombatView (implementazione concreta).

- **Presenter**

- Funziona da **mediatore**: riceve gli input dalla view, aggiorna
il modello e fornisce alla view i dati aggiornati da mostrare.
- In questo modo **model e view restano completamente
disaccoppiati**.

Entrambe le strutture assicurano un buon livello di **disaccoppiamento**:
è possibile sostituire la vista (ad esempio passando da Swing a JavaFX)
senza modificare il modello e con minime modifiche ai
controller/presenter.

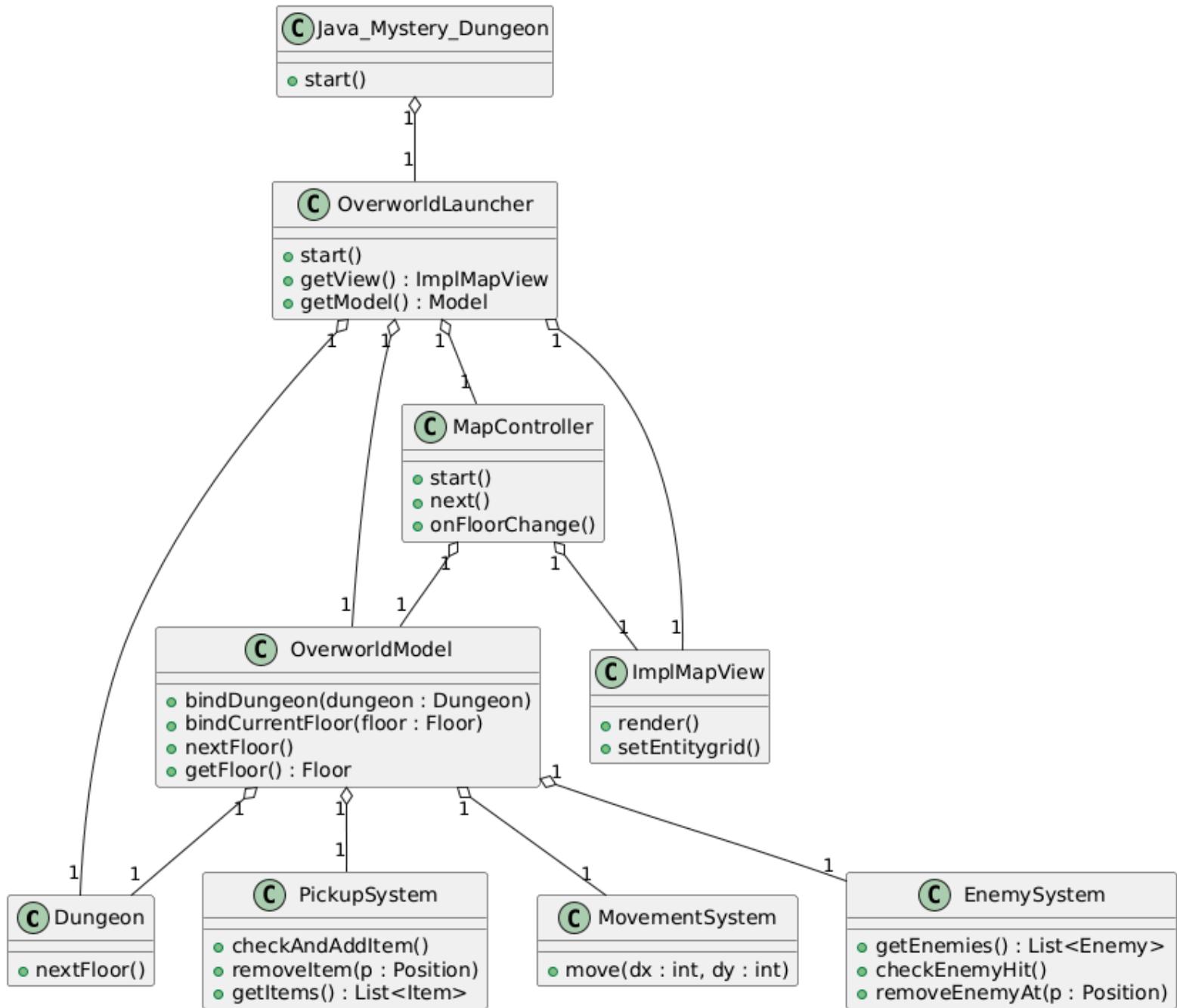


Figura 2.1: Schema UML dell'Overworld. OverworldLauncher è la classe che inizializza tutti i sistemi necessari per il overworld

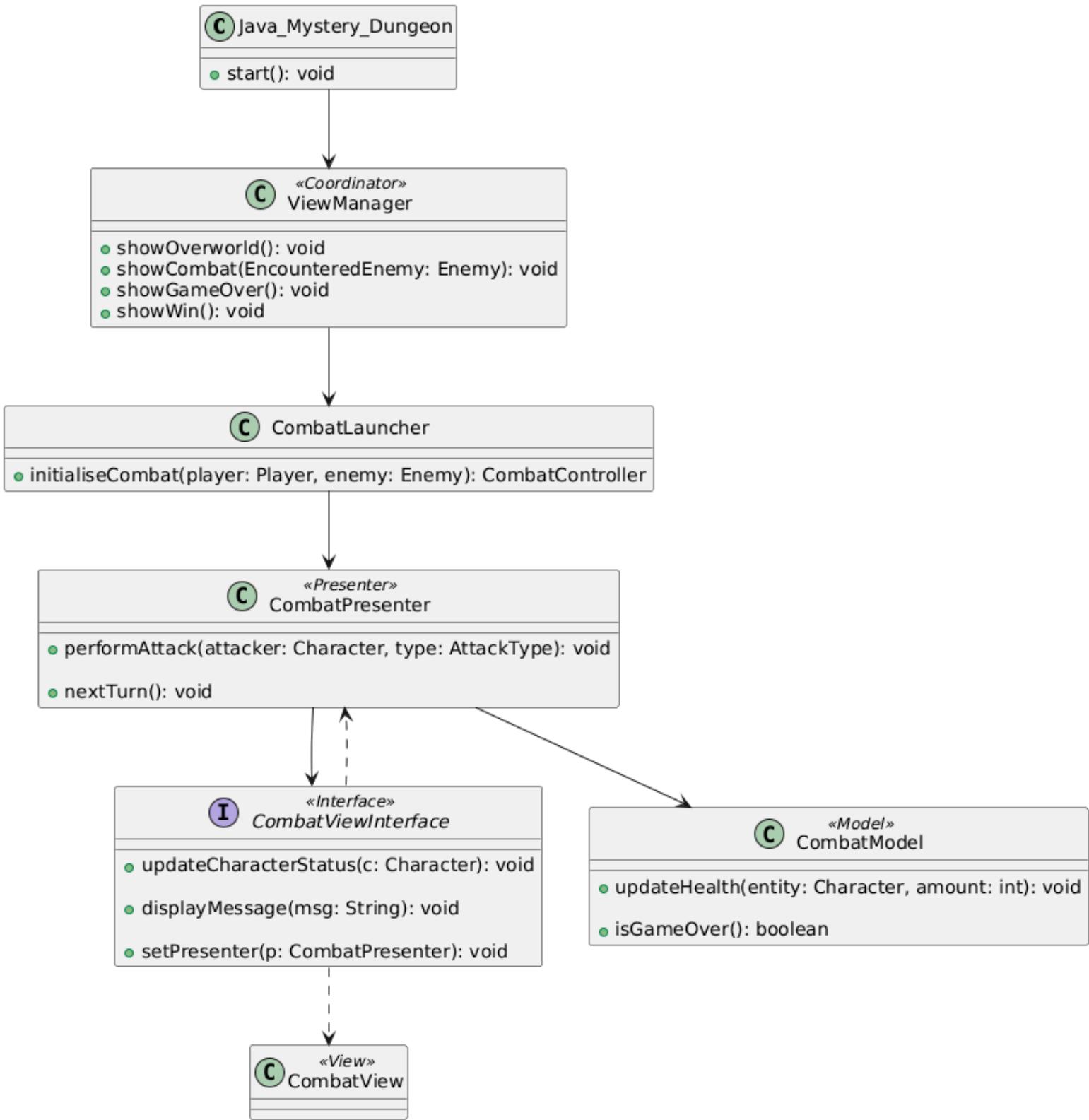


FIGURA 2.2: schema UML del combat System, ViewManager decide se mostrare CombatView oppure altre View

2.2 Design dettagliato

Laura Bertozzi

1) Strategie di movimento dei nemici

problema: Gestire la differenziazione dei movimenti del nemico: i nemici nel gioco possono muoversi seguendo un percorso stabilito (patroller), oppure seguire il giocatore.(follower)

soluzione: Lo state pattern, ossia creare un'interfaccia MovementStrategy che esponga in metodo executeMove, il quale verrà implementato in modo differente in base alle necessità.

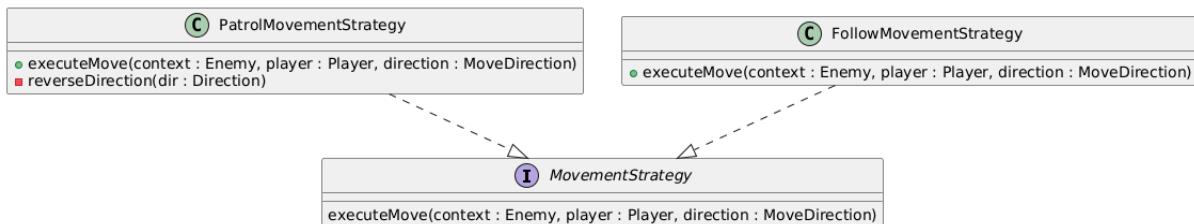


figura 2.3 Rappresentazione UML dello State Pattern il movimento dei nemici

2) Diverse tipologie di nemici

problema: differenziare i nemici: come accennato sopra sono presenti i patroller, i follower ed in aggiunta gli sleeper, i quali semplicemente restano immobili ed ingaggiano il combattimento solo se avvicinati.

soluzione: Ancora una volta lo state pattern. Alla base vi è l'interfaccia `GenericEnemyState`, implementata da `PatrollerState` e `SleeperState`, in aggiunta `FollowerState` espande `PatrollerState` aggiungendo al movimento iniziale di patrol, la possibilità di seguire in giocatore quando esso entra nel raggio visivo del nemico.

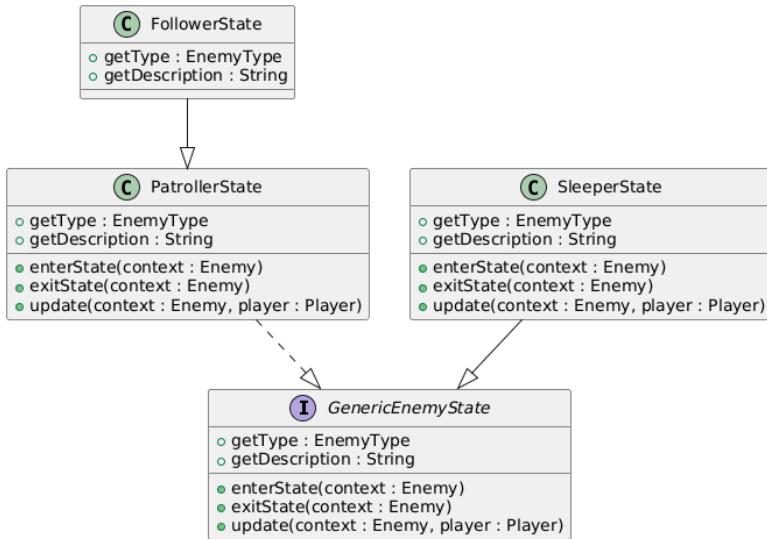


figura 2.4 Rappresentazione UML dello State Pattern le diverse tipologie di Nemici

3) Flessibilità nella gestione degli oggetti

problema: Avere flessibilità nella gestione dei diversi oggetti, ossia poterne creare di differenti ma con il medesimo metodo essenziale per l'utilizzo(use)

soluzione: Creare un'interfaccia item che esponga i metodi comuni, implementata da una classe astratta nella quale tali metodi sono definiti a sua volta estesa dai diversi tipi di oggetto, con le relative funzionalità aggiuntive. Nel nostro caso abbiamo creato solo l'oggetto pozione.

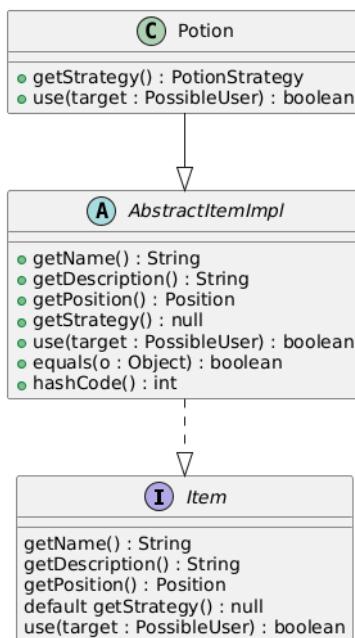


figura 2.5 Rappresentazione UML dell'interfaccia Potion e la classe astratta

4) Adattare il player dell'overworld al combattimento

problema: adattare il player dell'overworld a quello del combattimento, in quanto mentre nell'overworld il giocatore deve semplicemente muoversi e raccogliere oggetti, nel contesto del combat sono necessari metodi che tengano conto di stamina, hp e potenza.

soluzione: Adapter pattern, l'interfaccia PossibleUser definisce i metodi utili al combattimento e la classe OverworldPlayerAdapter la implementa ed incapsula il Player

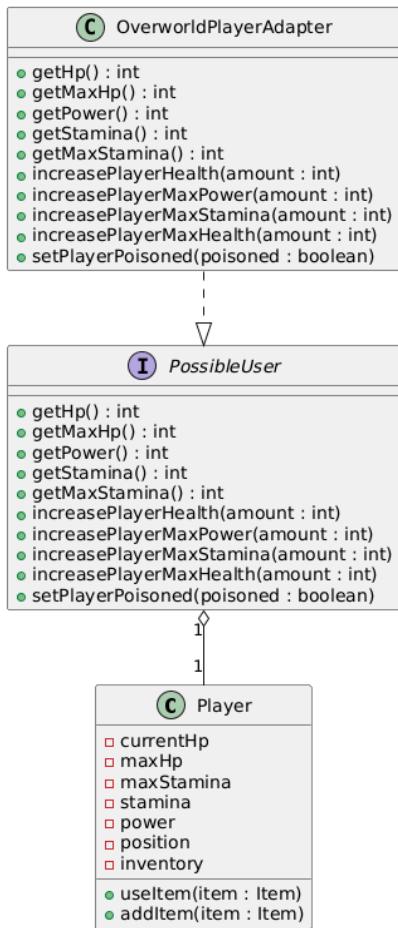


figura 2.6 Rappresentazione UML dell'Adapter Pattern

5) OverworldModel “godClass”

problema: OverworldModel gestiva sia il movimento del giocatore, che le sue interazioni con oggetti e nemici

soluzione: Pattern facade per delegare a ognuno di tre sistemi distinti una singola funzione.

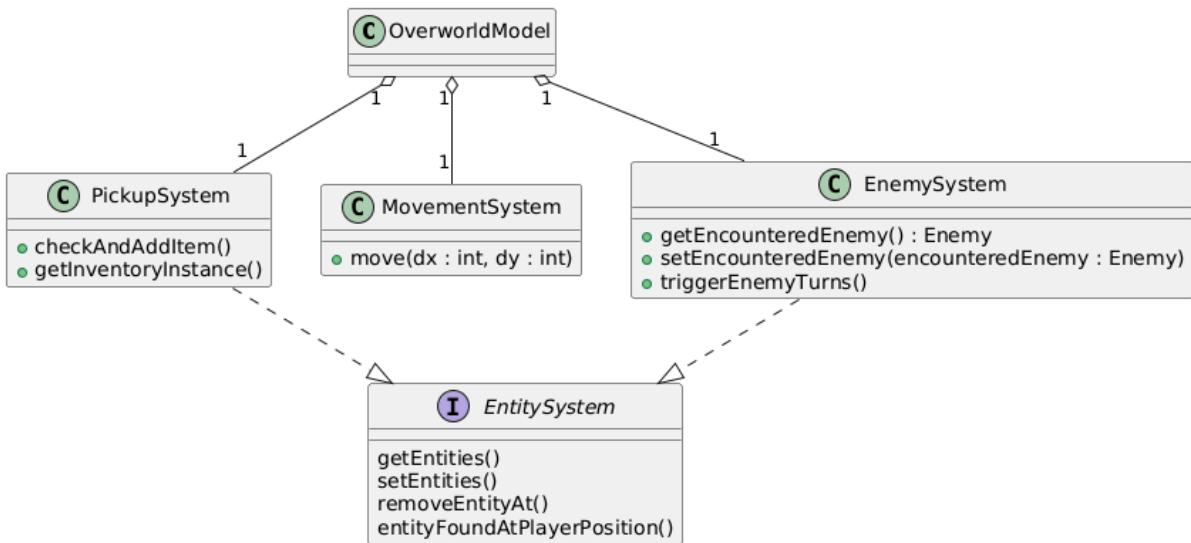


figura 2.7 Rappresentazione UML del pattern facade

Kelly Applebee

Problema:

Il combattimento si svolge a turni, quindi era necessario implementare un meccanismo di transizione tra il turno del giocatore, quello del nemico ed eventualmente diverse animazioni.

La prima soluzione ipotizzata era quella di usare un singolo metodo con numerosi *if/else*; questa idea è stata scartata molto velocemente in quanto risulta molto complessa in fase di debug e avrebbe reso il controller difficile da leggere, poco gestibile e complicato da estendere.

Soluzione: Per la gestione dei turni è stato adottato lo **State Pattern**

Questo approccio permette di rappresentare ogni fase del combattimento come uno stato indipendente, semplificando la logica del controller e al contempo facilitando l'aggiunta di nuovi stati senza dover apportare sostanziali modifiche al controller.

Gli stati implementati sono:

- **CombatState**: Interfaccia i cui metodi vengono implementati e sovrascritti dalle classi concrete.

- **AnimatingState:** Gestisce le animazioni al termine di un turno, disabilitando i comandi sia del giocatore che del nemico, passando da un turno all'altro al termine.
- **BossTurnState:** Gestisce il turno del boss, caratterizzato da una vita aumentata e un insieme di mosse e logiche diverse rispetto a un nemico normale.
- **EnemyTurnState:** Gestisce il turno di un nemico semplice.
- **GameOverState:** Gestisce la fine del combattimento; questo stato determina se il giocatore torna all'overworld (in caso di vittoria contro un nemico semplice), viene reindirizzato a una schermata di vittoria (in caso di vittoria contro un boss), oppure alla schermata di sconfitta.
- **InfoDisplayState:** Mostra informazioni sul nemico che si sta affrontando, disabilita tutti i bottoni eccetto quello necessario per tornare ai menù precedenti.
- **ItemSelectionState:** Gestisce la scelta e il successivo utilizzo delle pozioni durante il combattimento.
- **PlayerTurnState:** Gestisce le azioni disponibili al giocatore durante il suo turno.

Questo approccio ha permesso di evitare l'utilizzo di numerose condizioni annidate, rendendo il controller più leggibile e modulare. Inoltre lo State Pattern garantisce maggiore **estensibilità**, permettendo di, eventualmente, aggiungere facilmente altri stati (Pause State, PlayerSwapState ecc.) senza modificare la logica esistente.

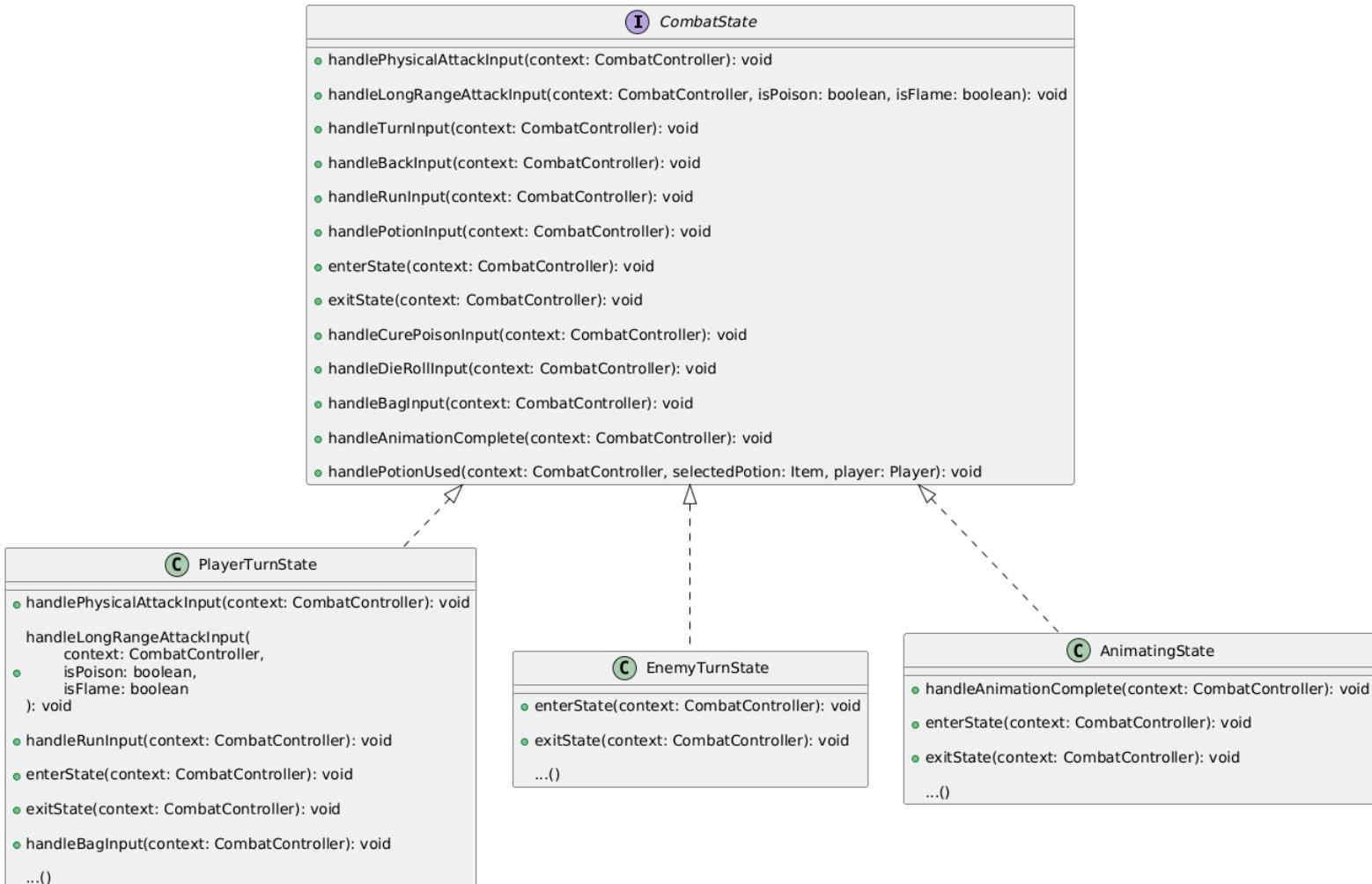


figura 2.8 Rappresentazione UML dello state pattern

Problema:

Durante il combattimento, sia il giocatore che il nemico devono poter eseguire azioni come attacchi fisici o a lungo raggio. Implementare direttamente la logica all'interno del controller avrebbe aumentato il numero di controlli e condizioni da gestire, rendendo il codice difficile da leggere, mantenere e debuggare.

Soluzione:

Per risolvere questo problema è stato adottato il **Command Pattern**. È stata definita un'interfaccia comune, **GameButton**, con un metodo **execute()**. Le classi concrete, che rappresentano i bottoni che il giocatore può cliccare, implementano questa interfaccia e sovrascrivono

il metodo `execute()`, incapsulando così la logica specifica del comando associato.

In questo modo il controller non deve conoscere i dettagli dell'azione da eseguire: si limita a chiamare il metodo `execute()` sull'interfaccia, delegando la logica alle classi concrete.

I comandi implementati sono:

- **GameButton:** Interfaccia il cui metodo `execute` verrà sovrascritto
- **LongRangeButton:** esegue un attacco a lungo raggio con la possibilità di applicare veleno
- **MeleeButton:** esegue un attacco fisico

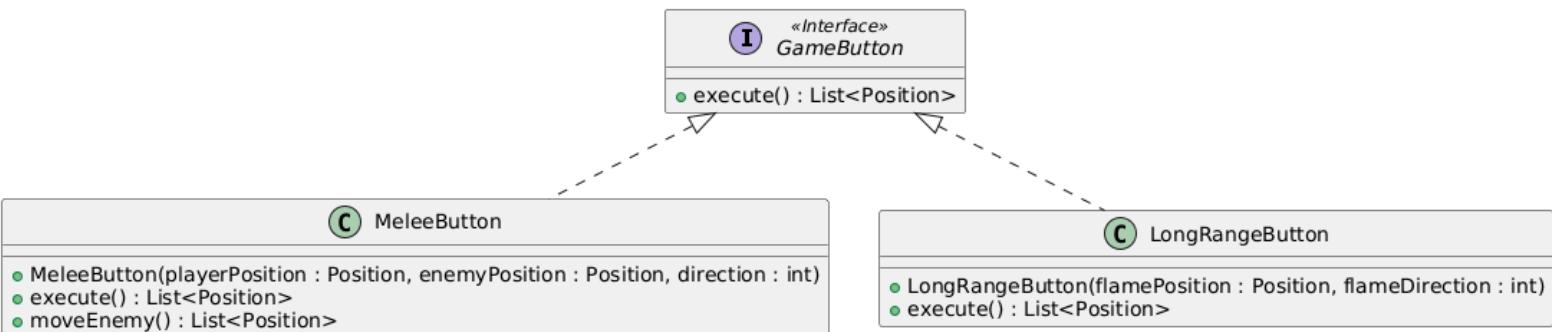


figura 2.9 Rappresentazione UML Command Pattern

Problema:

Durante lo sviluppo delle diverse classi *View* abbiamo optato per l'utilizzo della classe *CardLayout* di Java Swing per organizzare in maniera dinamica le schede di gioco.

Per mantenere la gestione e la manipolazione di queste *View* centralizzata è stata introdotta la classe *ViewManager*, il suo compito è quello di incapsulare la logica del *CardLayout* assieme alla transizione delle varie schermate. Un problema importante è sorto al momento di integrare la *View* dedicata al combattimento.

Dovendo transizionare dall'overworld allo stato di combattimento, la creazione di una nuova istanza di *ViewManager* (la prima soluzione testata) avrebbe comportato la perdita dell'istanza di *JFrame* e *CardLayout* già in uso dall'overworld.

Soluzione:

Per risolvere il problema sopra descritto abbiamo implementato il pattern *Observer*. Questa soluzione ci ha permesso di disaccoppiare la logica di rilevamento del combattimento dalla gestione effettiva del cambio delle *View*.

Gli elementi principali di questo Pattern sono:

- **ViewManagerObserver**: Interfaccia Observer
- **OverworldController**: Observer Concreto
- **CombatCollision**: Soggetto Notificatore

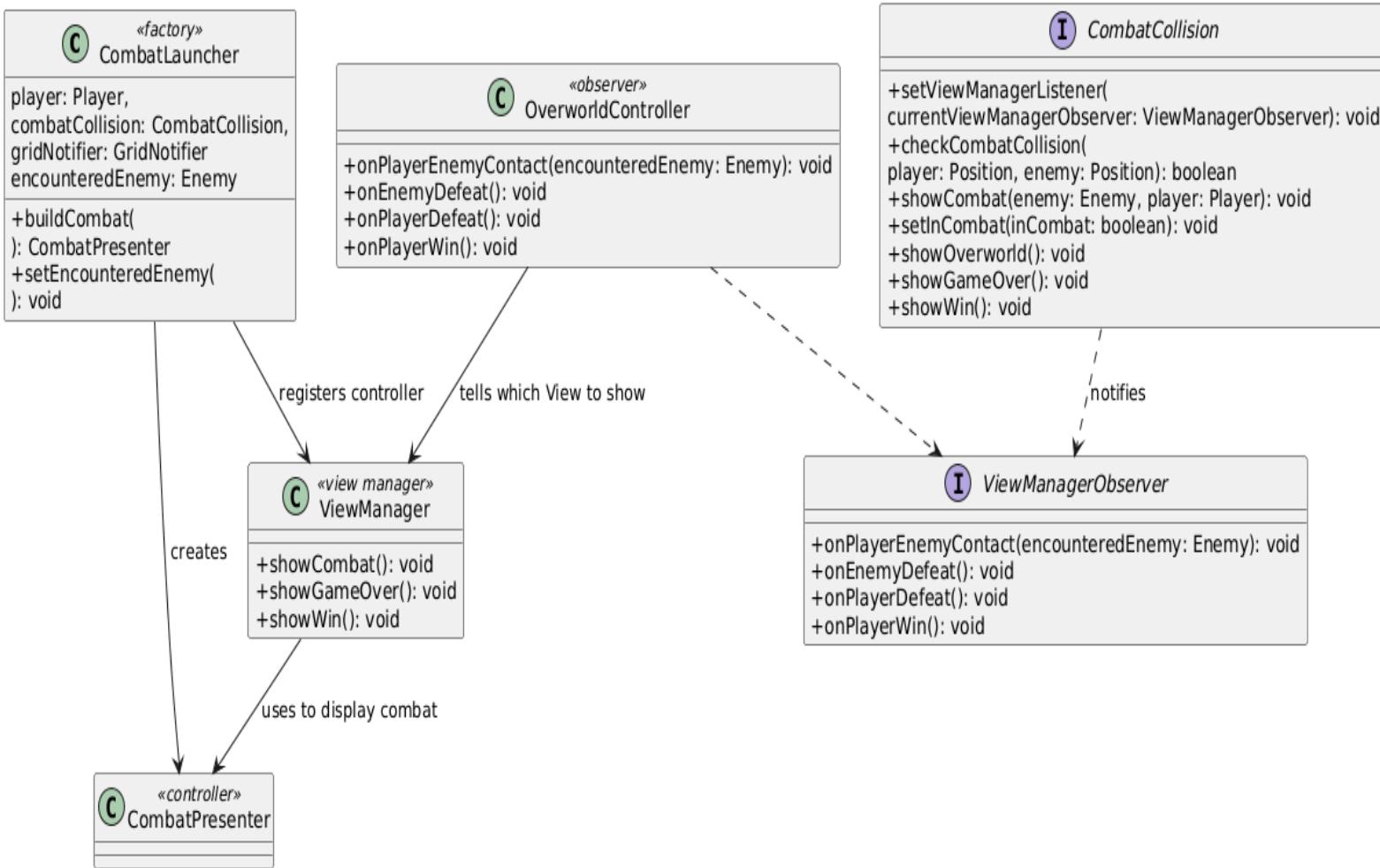


figura 2.10 Rappresentazione UML del pattern observer

Alice Beccati

Generazione delle stanze e dei corridoi

Problema:

Generare il layout dei piani che formano il dungeon permettendo di variare facilmente l'algoritmo di creazione (che in questo caso è

procedurale) di stanze, tunnel e piazzamento dei vari elementi; senza toccare il resto del sistema.

Soluzione:

Per permettere il disaccoppiamento degli algoritmi di posizionamento dal resto del sistema ho usato il pattern strategy.

Da OverworldLauncher viene iniettata la strategia desiderata e FloorGenerator si occupa della gestione, invocando i metodi opportuni per la corretta generazione del campo di gioco.

In figura 2.11 l'esempio del pattern applicato al posizionamento degli oggetti (scale, item, enemy...).

Strategy è stato usato allo stesso modo, come si evince dagli attributi in FloorGenerator, anche per gli algoritmi di posizionamento delle stanze (RoomPlacementStrategy) e dei tunnel (TunnelPlacementStrategy).

In questo modo, per cambiare tecnica di generazione di alcune parti del dungeon, basta soltanto fornire una nuova implementazione delle interfacce.

Il design resta estendibile, testabile e disaccoppiato da Model e View.

Nota: Algoritmo di generazione dei tunnel

Per collegare le stanze ho preso come esempio il Manhattan routing (collegamento ortogonale "a L").

Le stanze vengono considerate in sequenza e unite prendendo i loro centri e tracciando corridoi allineati agli assi: per ogni coppia, il percorso è a L (prima orizzontale e poi verticale, o viceversa, scelto casualmente).

Lo scavo interviene solo sulle celle di muro, lasciando inalterati i pavimenti delle stanze, così da garantire connettività semplice e una mappa leggibile.

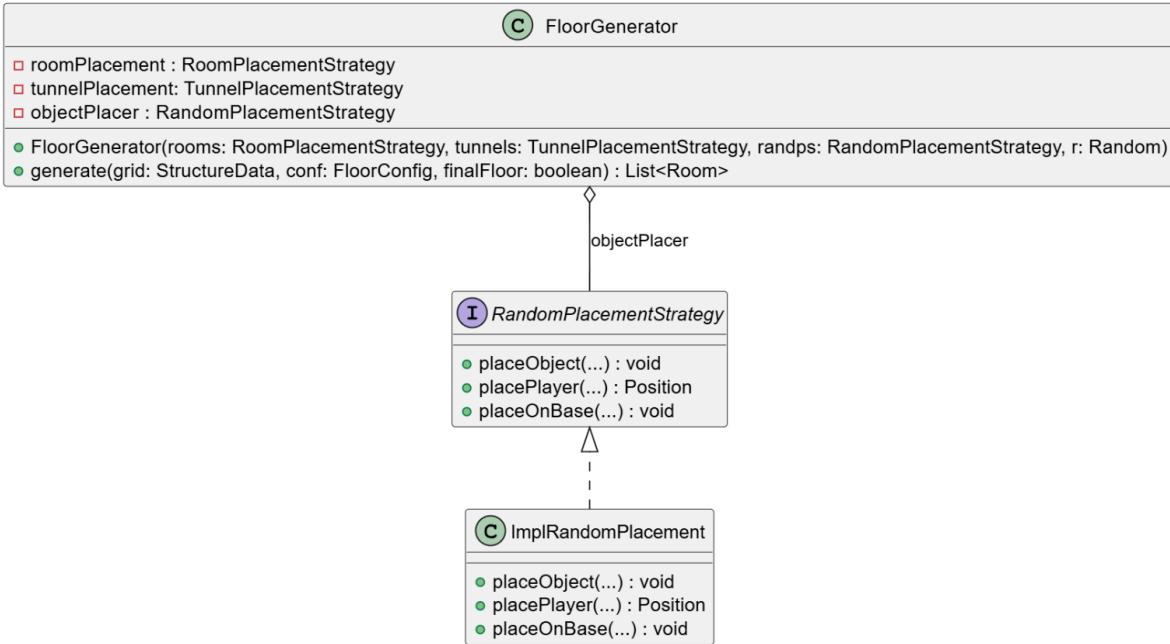


figura 2.11 Rappresentazione UML del pattern strategy

Astrazione della griglia e viste read-only

Problema:

Gestire due griglie distinte, una per la superficie calpestabile con layout del piano ed una per il posizionamento delle entità (nemici/oggetti/player), garantendo che View e Controller non possano modificarle.

Si vuole inoltre che l'applicazione sia indipendente dall'implementazione concreta delle griglie.

Soluzione:

Separazione dell'interfaccia mutabile da quella immutabile, grazie al pattern Adapter.

StructureData è l'interfaccia mutabile, utilizzabile solo all'interno del model per la scrittura su griglia.

ReadOnlyGrid è l'interfaccia di sola lettura, che viene vista da View e Controller.

Mentre ReaOnlyGridAdapter wrappa una StructureData e la espone come ReadOnlyGrid.

L'utilizzo di interfacce permette inoltre di disaccoppiare l'implementazione concreta dal suo utilizzo nel sistema.

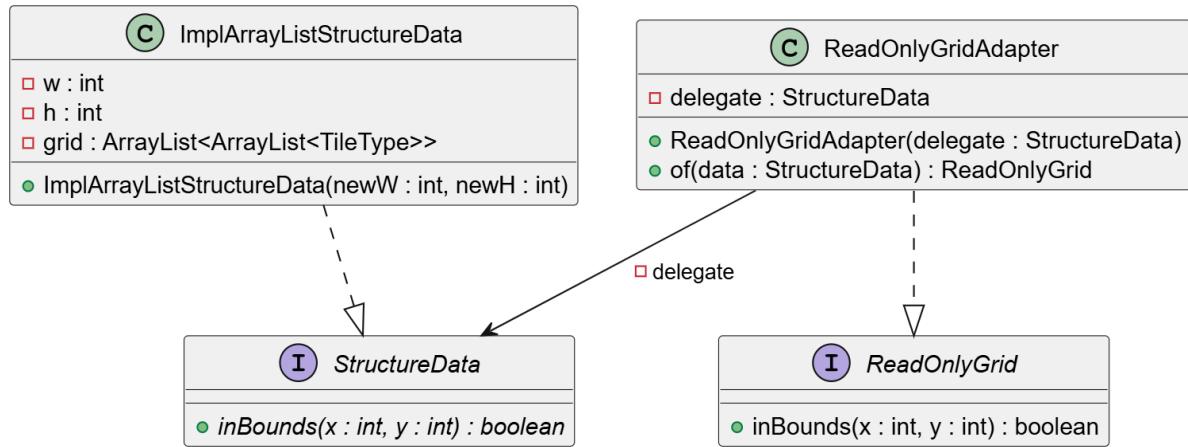


figura 2.12 Rappresentazione UML del pattern Adapter

Progressione dei piani e generazione lazy

Problema:

- Gestire la progressione tra più piani del dungeon senza pre-generare tutto. Evitando costi di tempo e memoria elevati all'avvio dell'applicazione.
 - Configurare il dungeon (es: dimensioni, numero stanze per piano, numero di piani...), senza avere ridondanza di dati e metodi con molti argomenti.

Soluzione:

Ho adottato una generazione lazy dei piani: il Dungeon non pre-genera nulla all'avvio; il Model invoca “il cambio piano” e solo in quel momento viene creato il nuovo Floor tramite FloorGenerator.

La configurazione per piano è costruita con il pattern Builder in FloorConfig, evitando costruttori con molti argomenti e ridondanza di parametri.

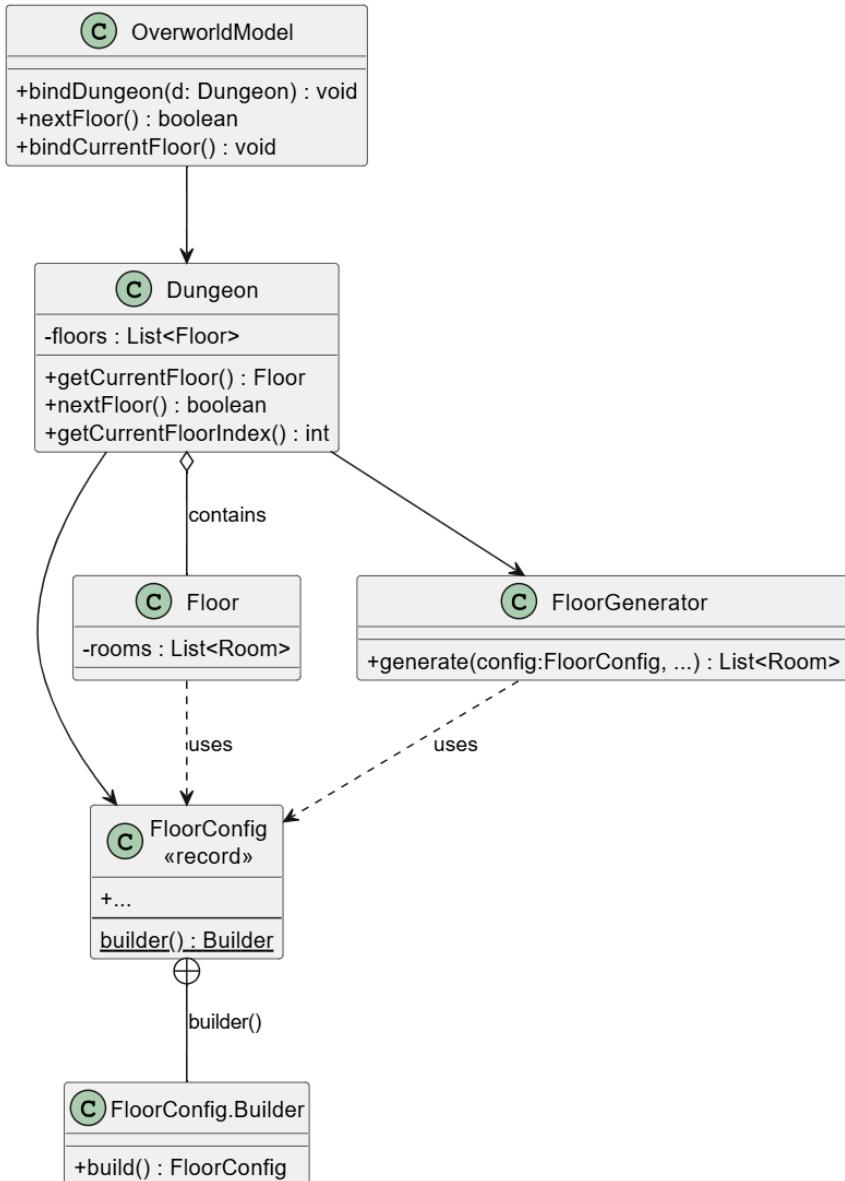


figura 2.13 Rappresentazione UML del Builder Pattern e Generazione Lazy

Eventi di aggiornamento griglia e cambio piano

Problema:

Notificare la View dei cambiamenti del mondo di gioco: movimenti (player e nemici), spawn/despawn di entità e cambio piano; senza accoppiare la View al Model.

Soluzione:

Per permettere l'ascolto e l'invio di notifiche, riguardo ai cambiamenti del mondo di gioco, ho implementato il pattern Observer.

OverworldModel funge da publisher, aggiornando lo stato dell'applicazione ed emettendo gli eventi via GridNotifier, che fa da

dispatcher, incapsulando i riferimenti agli observer e invocandoli quando il Model cambia stato.

GridUpdater e ChangeFloorListener sono le interfacce observer, che vengono concretamente implementate da EntityGridUpdater e MapController.

Tramite la gestione di notifiche è possibile garantire il disaccoppiamento, rispettando l'architettura mvc, inoltre vengono consentite più reazioni in parallelo permettendo di aggiungere in futuro nuove feature (es: gestione dell'audio).

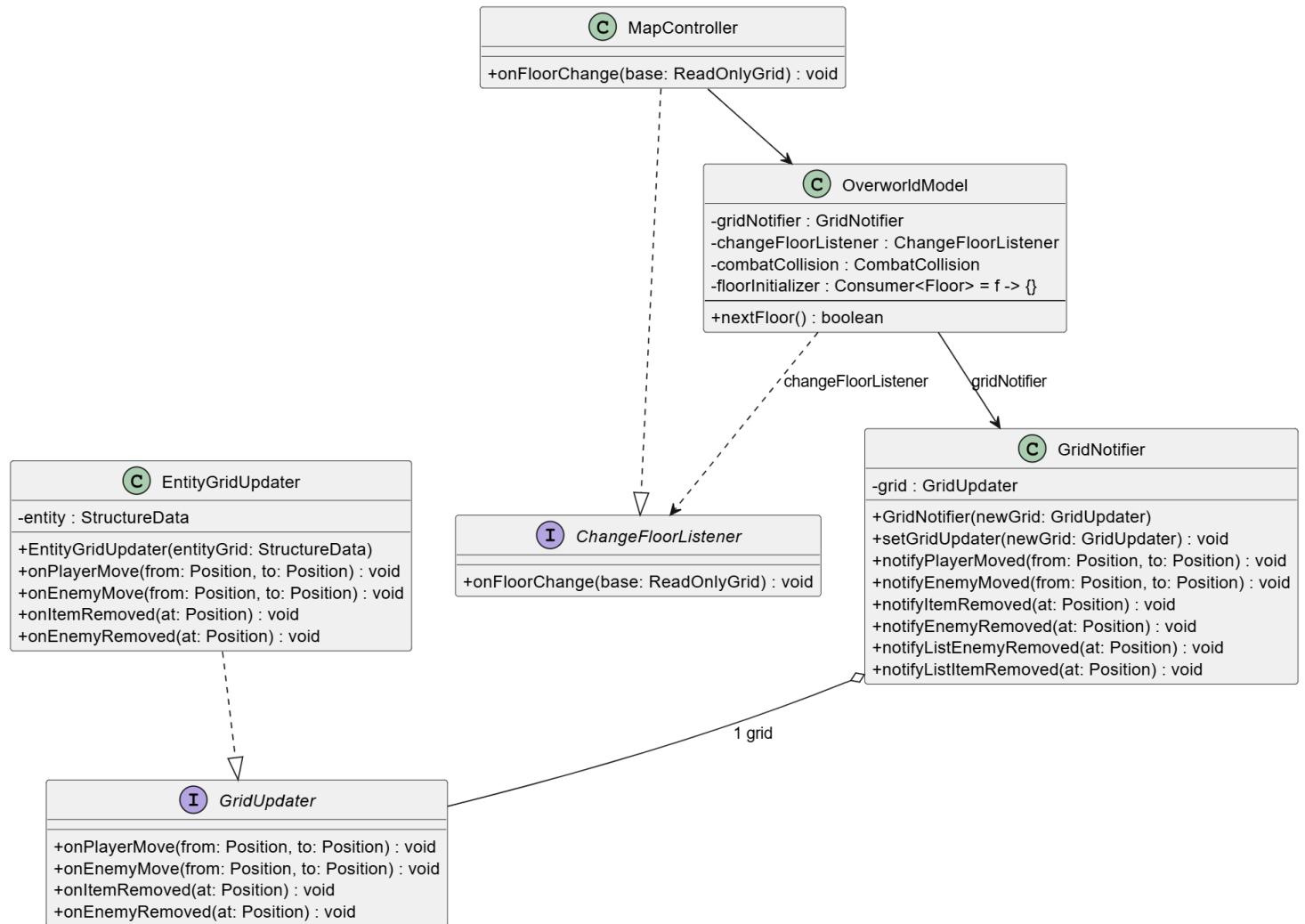


figura 2.14 Rappresentazione UML dell Observer Pattern

Matteo Monari

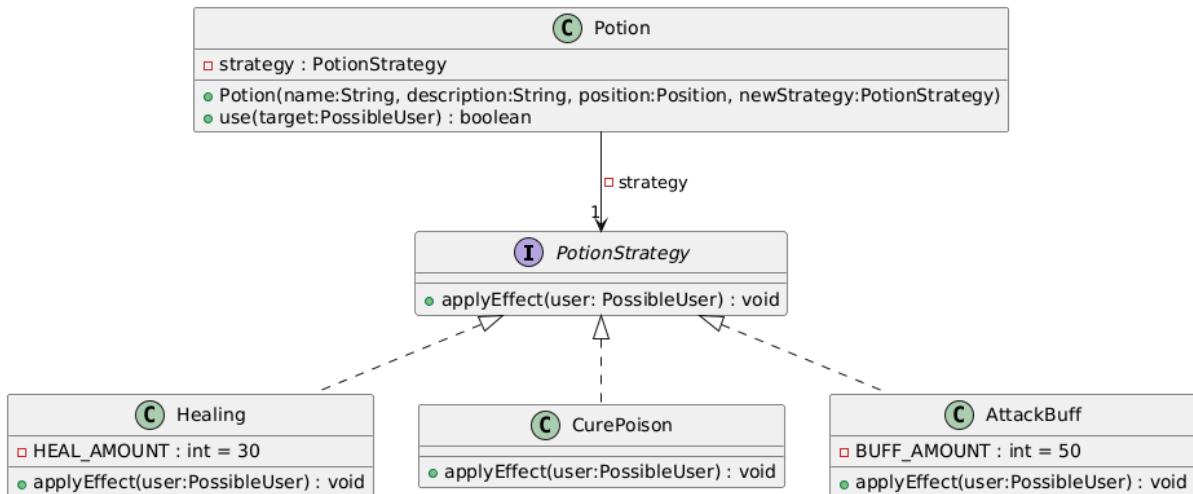


figura 2.15 Rappresentazione UML del Potion_Strategy

Problema:

Il videogioco include diversi Item raccoglibili nel mondo di gioco, chiamate **Potion**, che possono essere usate in combattimento. E' necessario implementare un sistema per creare o rimuovere, ed eventualmente aggiungere in futuro, nuove tipologie di pozioni. Ciò deve avvenire in modo facilitato senza dover modificare il codice di base. I vari tipi di **Potion** sono:

- **CurePoison** = agisce guarendo il giocatore dall'avvelenamento causato da un attacco nemico
- **AttackBuff** = aumenta la potenza del prossimo attacco del giocatore
- **Healing** = cura restituendo vita al giocatore

Soluzione:

Per l'implementazione delle **Potion** è stato deciso di utilizzare il pattern **Strategy (Potion_Strategy)** in cui è stata definita l' interfacchia

PotionStrategy con il metodo applyEffect a cui viene passato il player a cui applicare l'effetto.

La classe Potion delega l'effetto della pozione alla strategy associata, cioè, ogni pozione è rappresentabile semplicemente creando una classe che implementi Potion_startegy e questo garantisce che tutte le pozioni abbiano la stessa struttura esterna, ma un comportamento diverso a seconda della startegy associata.

Ciò permette al codice duttilità, semplicità e manutenibilità.

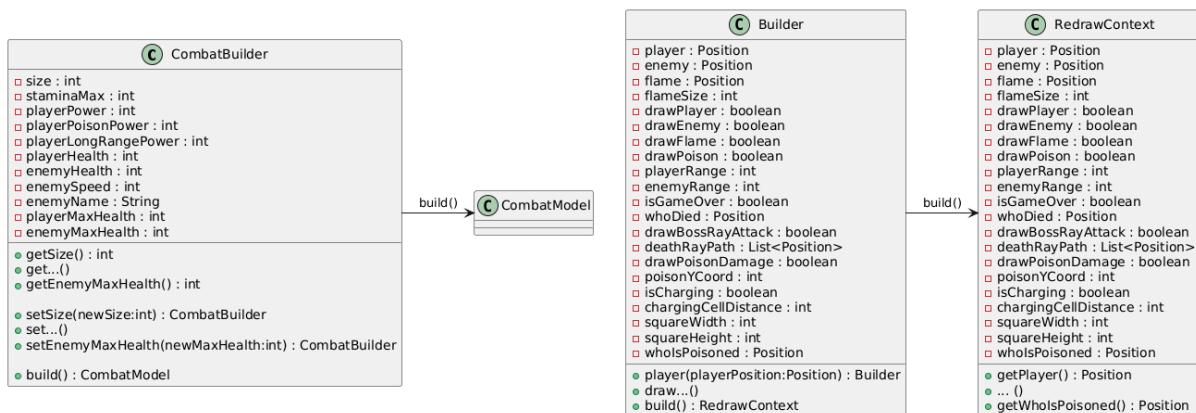


figura 2.16 Rappresentazione UML dei Builder Pattern
...() sostituisce i troppi metodi

Problema:

La creazione delle schermate di combattimento comprende numerosi parametri, sia dettagli grafici (posizioni, elementi da visualizzare, ecc.) sia statistici (vita, stamina, ecc.) e lo stanziamento di tutti questi oggetti complessi risulterebbe ostica, disordinata e poco flessibile.

Soluzione:

Per risolvere il problema è stato adottato il Build Pattern rappresentato in due classi:

- CombatBuilder permette di impostare parametri del giocatore e del nemico tramite gli appositi metodi set...().
Una volta configurati tutti gli attributi crea l'oggetto tramite il metodo build().

- RedrawContext(che fa uso di una Inner Class Builder) imposta i dettagli grafici tramite metodi descrittivi e creando gli oggetti usando build().

Questo garantisce chiarezza nella costruzione degli oggetti, flessibilità impostando solo i parametri necessari e estendibilità permettendo la creazione di nuovi oggetti senza scrivere codice.

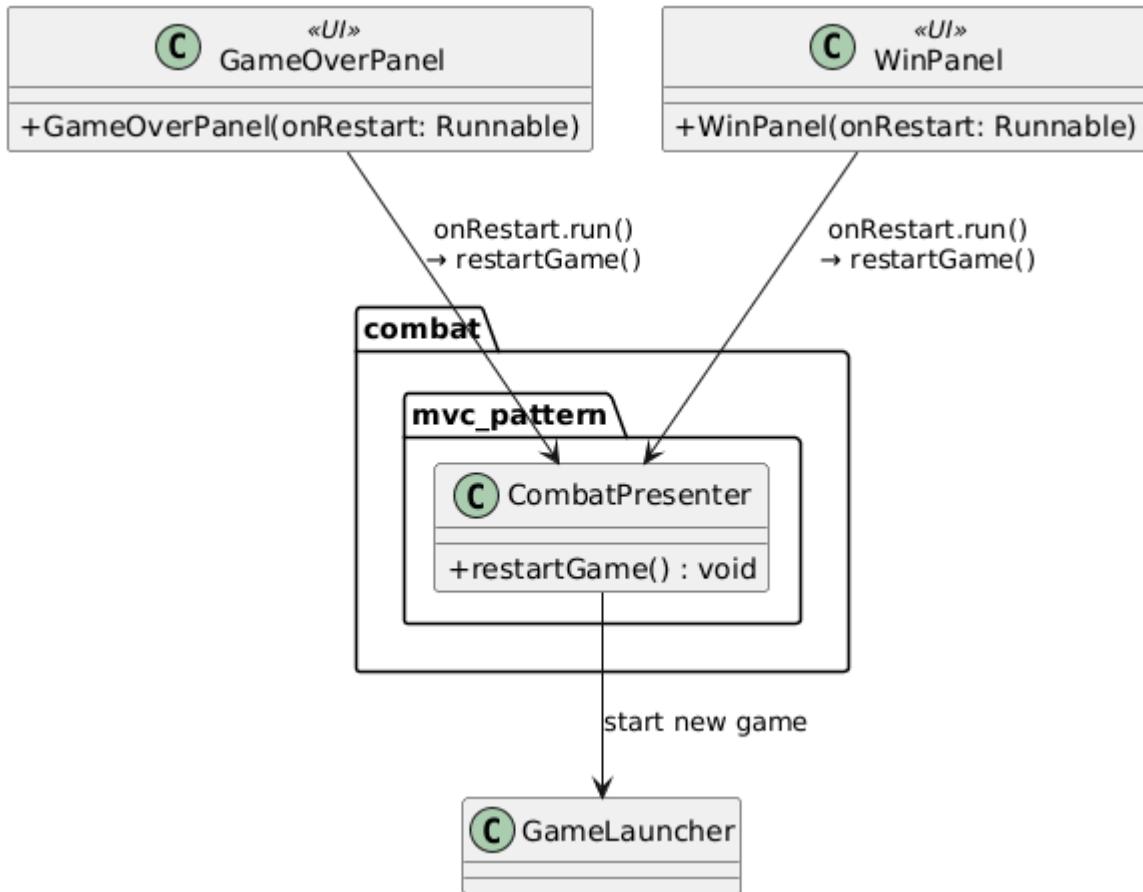


figura 2.17 Rappresentazione UML dell'implementazione del pulsante restart

Problema:

Lo stato finale del gioco comprende due possibili scenari, il Game Over o la vittoria del gioco. In entrambi è necessario sviluppare un sistema per far ripartire il programma senza che l'utente debba chiudere e riaprire l'applicativo.

Soluzione:

Si è pensato quindi di implementare nelle view di GameOverPanel e WinPanel un pulsante di Restart in modo tale da consentire all'utente di riavviare il programma con facilità e rapidità.

I pulsanti Restart presenti nelle due view richiamano il metodo restartGame() presente nel CombatPresenter che avvia una nuova sessione di gioco.

CAPITOLO 3

-Sviluppo

3.1 Testing automatizzato

GameOverState: classe dello state pattern che gestisce la logica di fine combattimento, distinguendo tra sconfitta del giocatore, vittoria contro un nemico normale o contro un boss.

È stata verificata la corretta esecuzione delle transazioni alle facciate di fine combattimento e degli effetti collaterali (messaggi, aggiornamento statistiche, disabilitazione e rimozione nemico, cambio schermata).

CombatModel: modello logico del sistema di combattimento. È stato testato per la corretta inizializzazione degli attributi tramite builder (vita, stamina, potenza), la gestione dei limiti (HP e stamina non oltre massimo/minimo) e la logica di attacco e game over.

Movement System, Pickup System, Enemy System: testati per garantire il corretto movimento del player, la possibilità di raccogliere oggetti e ingaggiare combattimenti con i nemici

Inventory: Testati i metodi di aggiunta e rimozione degli oggetti

WallCollision: Essenziale per gestire le collisioni delle entità mobili con i muri, ossia per evitare che i personaggi “fluttuino” al di fuori dell’area percorribile

CombatCollision: Testata per verificare che il combattimento venisse innescato correttamente

VisibilityUtil: Testata la visuale dei nemici

MovementUtil: Testata per verificare che i personaggi si muovessero solo nei casi a loro consentiti.

Per quanto riguarda la generazione del dugeon e del mondo di gioco sono state testate le seguenti classi e i seguenti aspetti:

- **Dungeon:** creazione del primo piano, avanzamento di piano fino al limite massimo con controllo delle istanze.
- **FloorGenerator:** corretto piazzamento delle stanze, all'interno della griglia e non sovrapposte, ognuna con almeno un tunnel collegato. Posizionamento di scale (una sola) e oggetti su celle valide.
- **Room:** avanzamento corretto dell'iteratore per scorrere le celle appartenenti alla stanza.
- **RandomPlacement:** piazzamento del numero corretto di oggetti e nelle celle valide (lontano da player e tunnel), piazzamento corretto del player.
- **TunnelPlacement:** corretta generazione in presenza di un diverso numero di stanze.
- **RoomPlacement:** generazione molteplici stanze in una griglia piccola, caso con 0 rooms.

¹

Neighbours: Testato per assicurarsi che l'algoritmo funzioni correttamente anche date distanze diverse da calcolare

CombatPresenter: Testato per assicurarsi che gli stati vengano scambiati correttamente, le animazioni partissero quando chiamate, ad ogni combattimento il primo turno sia **FATTO** dal giocatore e non dal nemico

CombatView: Testata con lo scopo di assicurarsi che tutte le componenti all'interno della view rispondessero correttamente agli input dell'utente e ai diversi stati del gioco

¹ Classi appartenenti al combatState non sono state testate esplicitamente perché testate abbastanza nel file "CombatPresenterTest" - Kelly Applebee

3.2 Note di sviluppo

Laura Bertozzi

Utilizzo di Optional, Stream e Lambda:

Un esempio che le racchiude tutte.

https://github.com/k3lly200413/Java-Mystery-Dungeon/blob/6038ae7c6c14fc8bc21e4d55e0762e0b67d042ef/src/main/java/it/unibo/progetto_oop/overworld/enemy/movement_strategy/wall_collision/WallCollisionImpl.java#L132C9-L143C16

Un altro esempio dello stesso tipo.

https://github.com/k3lly200413/Java-Mystery-Dungeon/blob/1a82e2baa1313a4a98edb67c9f12646ae28b7702/src/main/java/it/unibo/progetto_oop/overworld/mvc/model_system/EnemySystem.java#L104C5-L109C6

Bresenham Line Algorithm

Ideato da Jack Elton Bresenham, utilizzato per tracciare la “linea di visuale” dei nemici.

https://github.com/k3lly200413/Java-Mystery-Dungeon/blob/6038ae7c6c14fc8bc21e4d55e0762e0b67d042ef/src/main/java/it/unibo/progetto_oop/overworld/enemy/movement_strategy/VisibilityUtil.java#L97C5-L139C6

Applebee Kelly

Utilizzo della classe Timer

Utilizzato in diversi punti. Il seguente è un singolo esempio:

https://github.com/k3lly200413/OOP24-JMD-JavaMysteryDungeon/blob/65af8ec43d4f5da9f41a95e421f1673ccb70e72/src/main/java/it/unibo/progetto_oop/combat/mvc_pattern/CombatPresenter.java#L474-L498

Utilizzo della Classe Graphics2D

Utilizzato per rappresentare immagini personalizzate

https://github.com/k3lly200413/OOP24-JMD-JavaMysteryDungeon/blob/1b8ac349c01489fac88e29e502abfe1c5f3bb2ff/src/main/java/it/unibo/progetto_oop/combat/mvc_pattern/CombatView.java#L884-L892

Utilizzo di Runnable

Utilizzato in diversi punti. Il seguente è un singolo esempio:

https://github.com/k3lly200413/OOP24-JMD-JavaMysteryDungeon/blob/1b8ac349c01489fac88e29e502abfe1c5f3bb2ff/src/main/java/it/unibo/progetto_oop/combat/mvc_pattern/CombatPresenter.java#L354-L368

Utilizzo di Lambda

Utilizzato in diversi punti. Il seguente è un singolo esempio:

https://github.com/k3lly200413/OOP24-JMD-JavaMysteryDungeon/blob/1b8ac349c01489fac88e29e502abfe1c5f3bb2ff/src/main/java/it/unibo/progetto_oop/combat/mvc_pattern/CombatPresenter.java#L531-L541

ALICE BECCATI

Utilizzo di Stream

[https://github.com/k3lly200413/Java-Mystery-Dungeon/blob/f70c849ff835b54e22915200da4788671d04fd90/src/main/java/it/unibo/progetto_oop/overworld/playground/data/StructureData_strategy/ImplArrayListStructureData.java#L63C5-L66C6]

Utilizzo di Generici per test

https://github.com/k3lly200413/Java-Mystery-Dungeon/blob/f70c849ff835b54e22915200da4788671d04fd90/src/test/java/it/unibo/progetto_oop/overworld/playground/FloorGeneratorTest.java#L162-L169

Utilizzo di Method references, libreria SwingUtilities per il metodo invokeLater() e lambda

https://github.com/k3lly200413/Java-Mystery-Dungeon/blob/86edae1cb928c157665613cad0429eefa1ed3f22/src/main/java/it/unibo/progetto_oop/overworld/playground/MapController.java

Utilizzo di Runnable

[https://github.com/k3lly200413/Java-Mystery-Dungeon/blob/4d14650872bd8d65d902223f33b283cf9712bc59/src/main/java/it/unibo/progetto_oop/overworld/playground/view/game_start/GameStartView.java#L74]

MATTEO MONARI

In questo metodo del CombatPresenter vengono utilizzate le lambda expressions, interfacce funzionali(Runnable) e parti di JDK come javax.swing.Timer

https://github.com/k3lly200413/OOP24-JMD-JavaMysteryDungeon/blob/a69129845cae30a117eabe904fc22637ca40aa66/src/main/java/it/unibo/progetto_oop/combat/mvc_pattern/CombatPresenter.java#L808-L842

Analogo per questo metodo

https://github.com/k3lly200413/OOP24-JMD-JavaMysteryDungeon/blob/a69129845cae30a117eabe904fc22637ca40aa66/src/main/java/it/unibo/progetto_oop/combat/mvc_pattern/CombatPresenter.java#L769-L806

In questo metodo di GameOverState vengono usati javax.swing.Timer e le lambda expressions

https://github.com/k3lly200413/OOP24-JMD-JavaMysteryDungeon/blob/a69129845cae30a117eabe904fc22637ca40aa66/src/main/java/it/unibo/progetto_oop/combat/state_pattern/GameOverState.java#L65-L117

In questo metodo di CombatPresenter vengono usati javax.swing.Timer, le lambda expressions e uno stream

https://github.com/k3lly200413/OOP24-JMD-JavaMysteryDungeon/blob/a69129845cae30a117eabe904fc22637ca40aa66/src/main/java/it/unibo/progetto_oop/combat/mvc_pattern/CombatPresenter.java#L543-L594

CAPITOLO 4

-COMMENTI FINALI

4.1 Autovalutazione e lavori futuri

Laura Bertozzi

Il mio ruolo è consistito nello sviluppo della logica riguardante i nemici, l'inventario (con relativa view) e il player nel contesto dell'overworld, con focus particolare nei confronti della gestione del movimento e delle interazioni tra entità.

Punti di forza

I punti di forza del mio prodotto sono, a mio avviso, l'estensibilità e la modulabilità del codice, oltre alla presenza di test automatizzati.

Debolezze

Le debolezze sono invece che la struttura del codice non è particolarmente raffinata e la connessione tra i vari componenti sviluppati da diversi membri è limitata, il che rappresenta la principale debolezza del sistema; in particolare per quanto riguarda OverworldModel, utilizzata da tutti per ottenere istanze di oggetti utili, ritengo che abbia responsabilità eccessive.

Applebee Kelly

Ruolo

Come descritto nella proposta di progetto, il mio ruolo consisteva nello sviluppo della *View* del combattimento assieme al parziale sviluppo del *Controller*, o nel nostro caso, *Presenter* del combattimento.

Punti di forza

Un punto di forza rilevante del progetto è la corretta adozione dei design pattern, che ha contribuito a una struttura estensibile e facilmente debuggabile in generale.

Punti di debolezza

La debolezza principale riguarda la classe CombatPresenter.

Nonostante i metodi implementati risultino coerenti con le responsabilità assegnate, la dimensione della classe riduce la chiarezza complessiva e rischia di compromettere la facilità di debuggare a lungo termine. Una suddivisione ulteriore delle logiche interne, in conformità con il *Single Responsibility Principle* e con il principio di *separation of concerns*, avrebbe permesso di ottenere una struttura più snella e maggiormente allineata alle buone pratiche di progettazione.

Alice Beccati

Il mio ruolo all'interno del gruppo è in linea con quanto dichiarato nel momento della proposta del progetto. Mi sono occupata della parte di generazione del dungeon e dei suoi piani in maniera procedurale, sia per quel che riguarda la GUI, che dell'implementazione effettiva e dello sviluppo della schermata iniziale del gioco.

In aggiunta mi sono occupata della generazione effettiva delle istanze delle entità di gioco.

Punti di forza

Penso di essere riuscita a rispettare i pattern che mi ero proposta di utilizzare al momento della progettazione, garantendo estensibilità per aggiunte future.

Mi sono inoltre impegnata a rispettare sempre il design mvc, senza esporre elementi del model a view e controller.

Punti di debolezza

I commenti del codice risultano molto dettagliati in alcune classi e invece molto scarsi in altre, rendendo alle volte difficile l'interpretazione di alcuni algoritmi.

Inoltre potrebbe sicuramente essere migliorata la connessione delle mie classi che si espongono verso le parti dei miei compagni e di conseguenza alcune classi comuni che potrebbero risultare un po' prolisse.

Monari Matteo

Il mio ruolo nel gruppo è rimasto fedele a quello dichiarato nella proposta di progetto ovvero, gestione dei dati riguardanti il combat-system come model e controller, o presenter nel nostro caso; creazione e gestione schermata di fine gioco (sia di vittoria che di sconfitta)

Punti di forza

Ritengo che i punti di forza del mio codice siano:

L'uso di pattern utili per la corretta implementazione e per facilitare la comprensione del combattimento.

L'utilizzo di javadoc e di checkstyle per rendere il codice chiaro e comprensibile.

L'estensibilità, in quanto fornisce la possibilità di rimuovere o aggiungere oggetti senza riscrivere codice.

Testabilità del codice, l'utilizzo di interfacce e di classi separate favorisce la creazione di test per il corretto funzionamento del codice.

Inoltre ritengo che il mio codice risulti in linea con l'impiego specificato precedentemente nella richiesta di progetto in quanto sviluppa parte della logica del combattimento e delle schermate di fine gioco.

Punti di debolezza

Ritengo che il punto di debolezza principale del mio codice sia lo scarso utilizzo di costrutti avanzati utili contro lo spreco di memoria o di efficienza del sistema.

La connessione delle mie classi e oggetti a quelle del progetto risulta leggermente macchinosa e lenta in fase di debug.

Inoltre in certi metodi sono presenti alcuni blocchi ripetuti che potrebbero essere rielaborati e ottimizzati in funzioni comuni.

4.2 Difficoltà incontrate e commenti per i docenti

Alice Beccati

Difficoltà principali

La sfida più grande è stata l'esame scritto in laboratorio, soprattutto la parte con JUnit. Alla fine del corso non mi sono sentita pienamente pronta al livello

di ragionamento richiesto, in particolare sugli esercizi di ottimizzazione del codice (riduzione delle ripetizioni, uso di Stream e pattern non banali) da svolgere in tempi molto stretti. Anche il fatto di avere poche possibilità di rifiuto ha aumentato l'ansia e spinto talvolta ad accontentarsi di un voto inferiore alle aspettative.

Cosa ho imparato

Il progetto è stato un'ottima palestra per il lavoro di gruppo, l'uso di Git e la capacità di organizzarsi con scadenze autoimposte per restare allineata con il team.

Kelly Applebee

Difficoltà principali

In conclusione, l'esame scritto ha rappresentato la principale difficoltà del corso. Al tempo non ero in grado di ragionare correttamente con le stream, anche perché non ritengo che sia stato trasmesso in modo chiaro quanto fossero strumenti importanti e potenti. Inoltre, il limite di tempo per il primo esercizio ha aggiunto ulteriore pressione, rendendo difficile mantenere la lucidità necessaria per affrontare problemi che richiedevano l'utilizzo delle stream, già di per sé per me complesso.

Matteo Monari

Difficoltà Principali

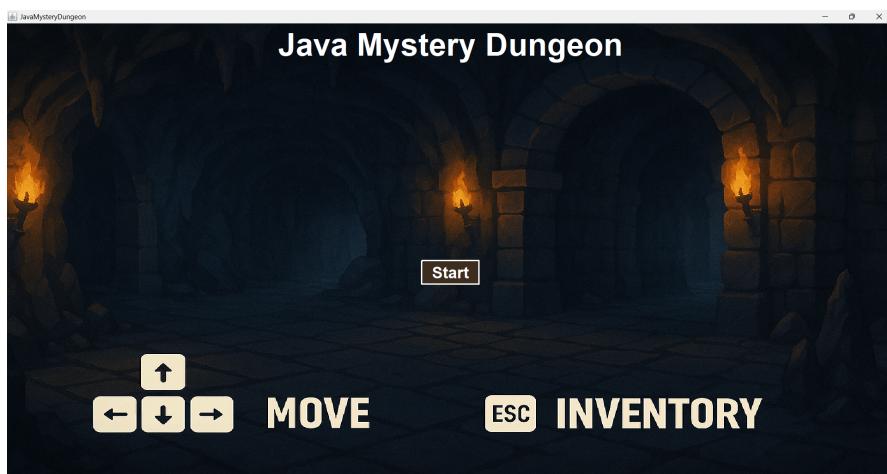
La difficoltà principali di questo corso riguarda principalmente l'esame pratico in laboratorio che ho trovato molto ostico. Ho preferito molto di più cimentarmi nella sfida del progetto che ho trovato più coinvolgente, stuzzicante e a tratti anche divertente.

Cosa Ho Imparato

Ho imparato a programmare usando classi, oggetti e interfacce costruite dai miei compagni e quindi ad adattare il mio stile di programmazione a quello del team.

Ho imparato a relazionarmi e a socializzare con i miei compagni per risolvere problemi comuni e a lavorare in squadra specialmente con Kelly con cui ho condiviso l'implementazione del controller (CombatPresenter).

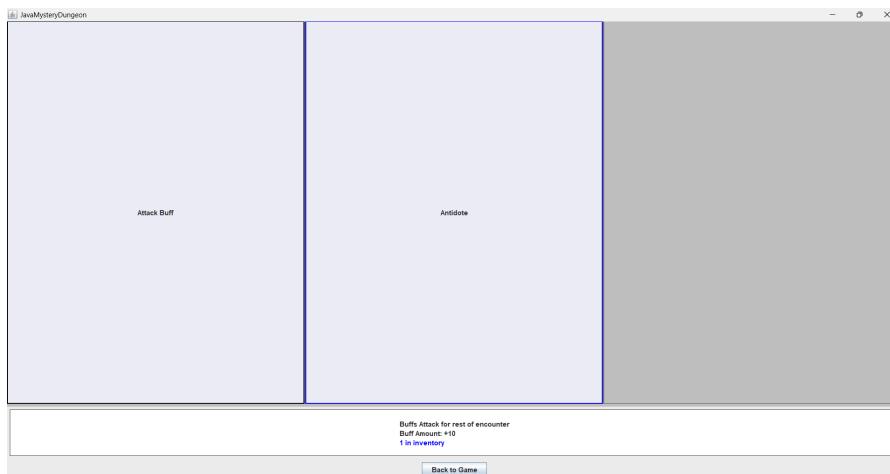
Appendice A - Guida utente



Il gioco si apre con una schermata che esplicita i comandi.



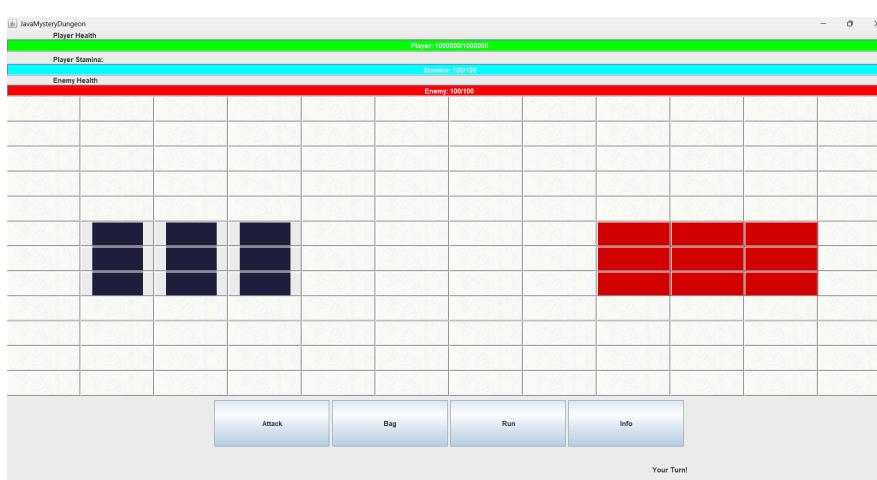
Nell'area esplorabile sarà possibile raccogliere pozioni, camminandoci sopra.



Tali pozioni sono visibili nell'inventario(aperto con **Esc**); cliccando sopra ad uno slot appare la descrizione della pozione e la quantità disponibile.



Si incontrano poi i nemici, fissi o mobili, con i quali si ingaggia il combattimento entrando nell'area 3x3 ad esso circostante.



Si aprirà quindi la schermata di combattimento, con i relativi bottoni: **attack**, per selezionare il tipo di attacco da eseguire, **bag**, per utilizzare le pozioni raccolte, **info**, per conoscere dettagli riguardanti il nemico da sconfiggere ed infine **run**, per sottrarsi allo scontro.

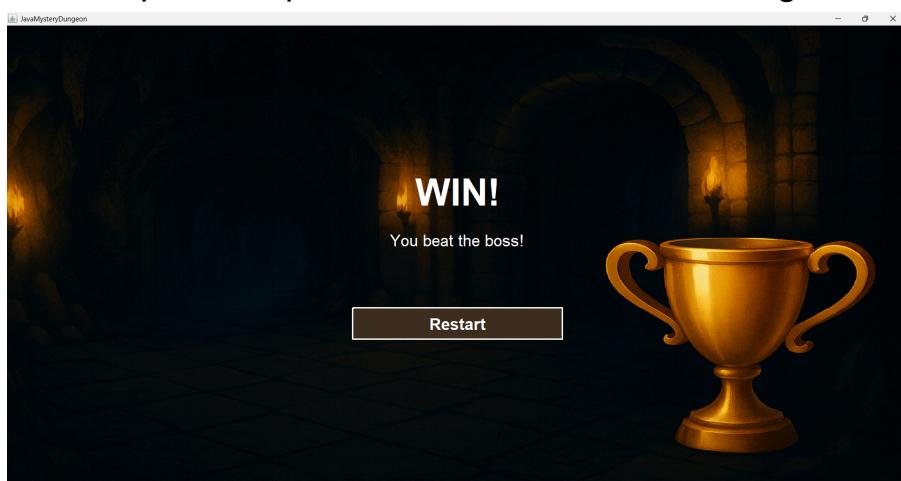
Una volta vinto lo scontro si ritorna alla mappa, se invece si viene sconfitti si apre una schermata di game over e sarà necessario ricominciare la partita.



Sparse per i piani si trovano inoltre le scale, che permettono di accedere al piano successivo.



L'ultimo piano è quello del boss, sconfitto lui, il gioco è terminato:



Appendice B - Esercitazioni di laboratorio

B.0.1 alice.beccati@studio.unibo.it

- **Laboratorio 07:** <https://github.com/AliceBeccati/lab07.git>
- **Laboratorio 08:** <https://github.com/AliceBeccati/lab08.git>
- **Laboratorio 12:** <https://github.com/AliceBeccati/lab12.git>