1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

CORSO DI LAUREA IN
INFORMATICA - SCIENZA E TECNOLOGIA

Relazione per il Progetto del Corso di Programmazione e Modellazione a Oggetti a.a. 2024/2025

ESCAPE CHOICE

Luca Piovaticci
matricola: 328235

Settembre 2025

Indice

1 Analisi

L'obiettivo del progetto è la realizzazione di un'applicazione che simuli un videogioco narrativo interattivo a scelta multipla, con meccaniche di sopravvivenza basate su vite, inventario, nemici, infezione ed eventi condizionati dalle scelte del giocatore.

Il contesto del gioco è composto da:

- Un giocatore, che all'avvio della partita ha la possibilità di inserire il proprio nome e selezionare un livello di difficoltà. Durante la partita può raccogliere oggetti, perdere vite e affrontare situazioni di pericolo.
- Un insieme di stanze all'interno dell'ambiente, che rappresentano le scene della storia.
- Un gruppo di scelte possibili all'interno di ogni stanza, che permettono di decidere come proseguire la narrazione.
- Un inventario posseduto dal giocatore, che può contenere diversi oggetti (arma, codice, kit medico).
- Un meccanismo di infezione che può attivarsi sulla base di determinate scelte.

Il gioco prevede che il giocatore abbia la possibilità di selezionare una tra le scelte possibili in ogni scena: ogni decisione porta a conseguenze diverse, come lo spostamento in un'altra stanza, l'acquisizione di oggetti, l'incontro con nemici o l'attivazione di condizioni di rischio. Sequenze diverse di scelte fatte possono portare alla vittoria della partita o alla sconfitta.

1.1 Requisiti

1.1.1 Requisiti minimali

- Scelte multiple testuali ad ogni scena, in alcuni casi la possibilità di tornare indietro, con percorsi ramificati.
- Gestione delle vite: al loro esaurimento la partita termina con la sconfitta.
- Gestione dell'inventario: contiene gli oggetti raccolti nel corso della partita.
- Gestione dei nemici con logica di combattimento semplificata. L'esito dei combattimenti è determinato dalla presenza di arma e vite.
- Presenza di finali multipli: percorsi diversi possono portare alla vittoria o alla sconfitta.
- Game over per esaurimento vite, mancanza di arma in inventario o scelte errate.
- Presenza di una GUI per l'interazione con l'utente: pulsanti per le scelte e testi narrativi delle scene.
- Salvataggio e caricamento della partita: persistenza degli stati tramite serializzazione.

1.1.2 Requisiti opzionali

- Scelta tra tre possibili livelli di difficoltà (easy, medium, hard), in base alla quale vengono impostate le vite iniziali.
- Visualizzazione di una mappa tramite una scelta specifica nel gioco.
- Inventario e numero di vite (dinamici) visibili durante la partita.
- Gestione del rischio in alcune scene del gioco (gestite random).
- Scrittura e lettura della narrazione delle scene da file esterno in formato JSON. Questo permette: una separazione tra logica e contenuti, facilità di aggiornamento della storia e possibilità di inserirne di nuove, estendibilità del gioco con storie diverse.

1.2 Modello del dominio

Il gioco Escape Choice ha come obiettivo la fuga del giocatore da un laboratorio abbandonato. La narrazione è suddivisa in scene che corrispondono alle diverse stanze presenti nell'ambiente, ognuna caratterizzata da una descrizione testuale e da un insieme di scelte che possono portare ad altre scene, permettere di raccogliere oggetti, portare direttamente ad una condizione di vittoria o sconfitta.

Durante la partita il **giocatore (Player)**:

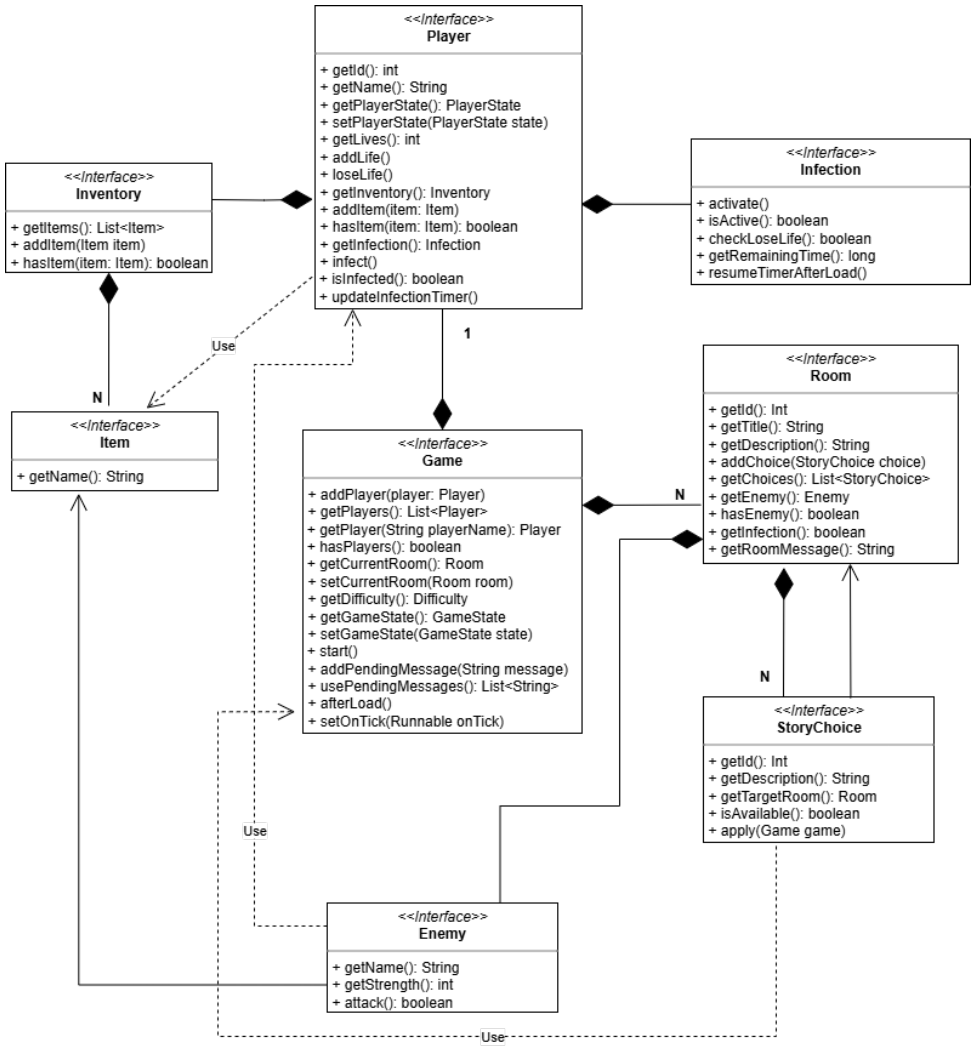
- può raccogliere **oggetti (Item)** da inserire nell'**inventario (Inventory)**, utili in situazioni specifiche;
- ha un numero limitato di vite (settate inizialmente in base alla difficoltà scelta), che possono ridursi durante lo scontro con **nemici (Enemy)** o in seguito a pericoli incontrati durante le diverse **stanze (Room)**;
- è influenzato dalle **scelte (StoryChoice)** e da condizioni particolari dello stato del **gioco (Game)**, come la comparsa di un'eventuale **infezione (Infection)**.

Le entità principali individuate nel dominio sono:

- **Game**: rappresenta lo stato globale della partita e può includere informazioni sul nome del giocatore, sul livello di difficoltà, vite correnti, inventario, stato infezione, scena attuale.
- **Player**: rappresenta il giocatore protagonista del gioco.
- **Room**: scena narrativa con descrizione e gruppo di scelte.
- **Choice**: scelta che il giocatore può selezionare per avanzare nella partita.
- **Item**: oggetto che si può raccogliere (arma, codice, kit medico +1 vita) che può modificare l'esito di una scena e/o la partita.
- **Enemy**: ostacolo presente in alcune scene: l'esito del possibile scontro può dipendere dagli oggetti (arma) presenti nell'inventario.
- **Inventory**: insieme degli oggetti raccolti.

- **Infection**: condizione del giocatore permanente che si attiva durante una particolare scena, provocando la perdita di 1 vita ogni tot. minuti (3 minuti di default), fino al termine della partita.

Una partita può terminare con la vittoria del Player, in caso di fuga positiva, oppure con la sconfitta dovuta all'esaurimento vite o scelte narrative errate.



Schema UML del dominio, con le entità principali e le relazioni tra loro

2 Design

2.1 Architettura

Per lo sviluppo dell'applicativo è stato utilizzato il pattern architetturale **Model-View-Controller** (**MVC**), mantenendo separati la logica (**Model**), interfaccia (**View**) e gestione degli input (**Controller**).

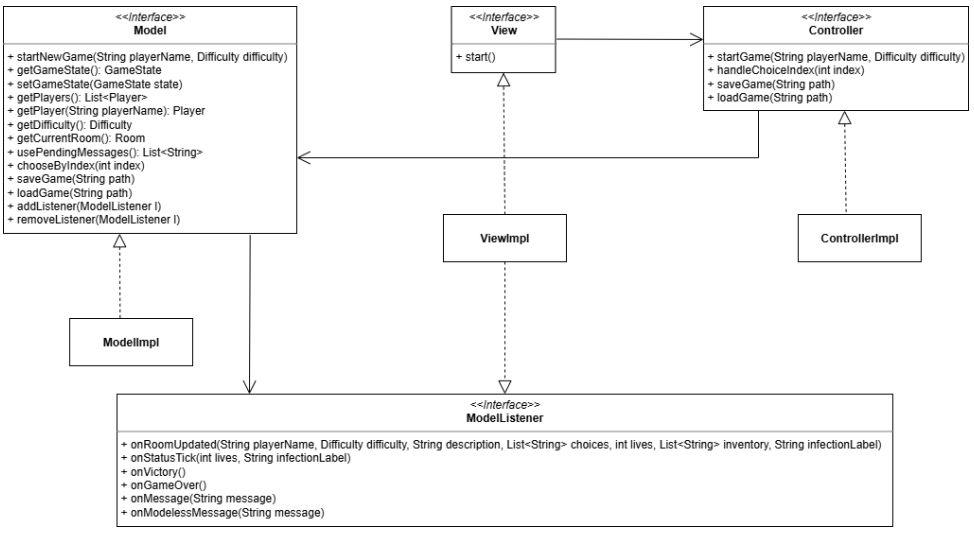
L'interazione è disaccoppiata grazie all'utilizzo del pattern **Observer**, implementato attraverso la realizzazione di un'interfaccia `ModelListener`: il Model non conosce la View concreta, ma solo l'interfaccia. In questo contesto:

- il Model è il Subject (soggetto osservato);
- la View è l'Observer o Listener (osservatore).

Nel progetto attuale viene registrata un'unica View come listener, ma l'architettura è estendibile a più osservatori.

Le componenti interagiscono tra loro in questo modo:

- Il Model gestisce lo stato e la logica di gioco, restando indipendente da View e Controller, e notifica la view tramite `ModelListener`;
- La View realizza la GUI con Swing, riceve input dall'utente e li inoltra al Controller; come osservatore del Model, riceve notifiche tramite `ModelListener` e aggiorna l'interfaccia grafica.
- Il Controller riceve gli input dell'utente dalla View e li elabora in comandi per il Model, senza accedere direttamente alle classi concrete del dominio;



Schema UML dell'architettura MVC, con le relazioni tra Model, View, Controller e l'interfaccia `ModelListener`

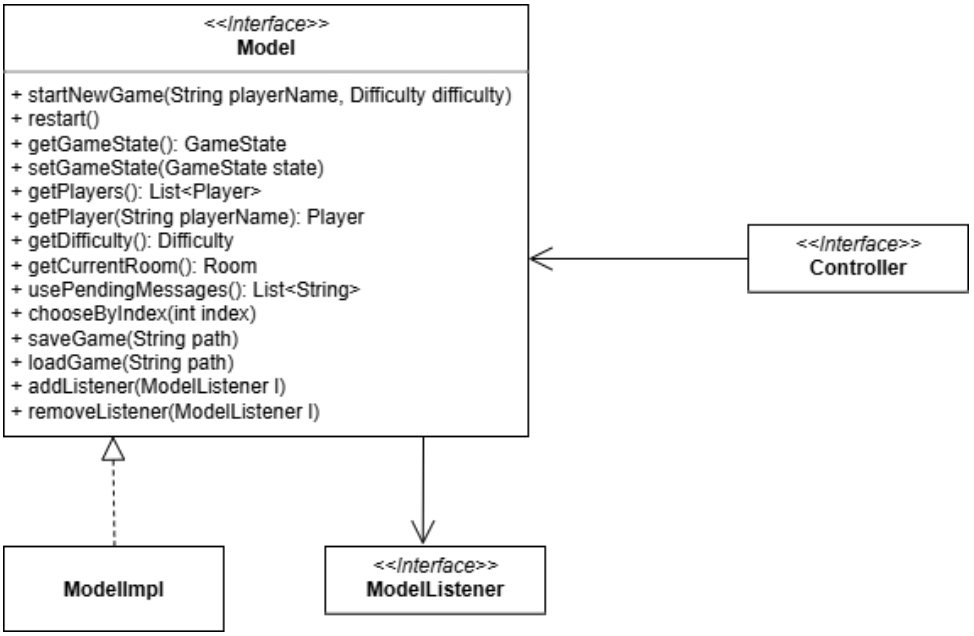
2.1.1 Model

Il Model è responsabile della gestione della logica e dello stato del gioco. La sua funzione principale è quella di determinare in modo logico come le azioni e le scelte del giocatore influenzino il sistema, garantendo indipendenza completa da View e Controller.

Questa funzionalità è stata realizzata definendo l'interfaccia `Model`, che espone metodi per avviare una nuova partita, elaborare le scelte, salvare e caricare lo stato di gioco e registrare eventuali listener.

L'implementazione concreta del Model è fornita dalla classe `ModelImpl`, che mantiene internamente l'istanza di `Game` e coordina gli aggiornamenti verso la View tramite il pattern **Observer**. Nello specifico, il Model notifica i cambiamenti attraverso l'interfaccia ModelListener, permettendo alla View di aggiornarsi senza conoscere i dettagli interni della logica di gioco.

`ModelImpl` implementa un sistema di salvataggio e caricamento della partita, serializzando lo stato interno su file binario e ripristinandolo in seguito. Questa funzionalità sfrutta il fatto che le classi del dominio sono state rese serializzabili.



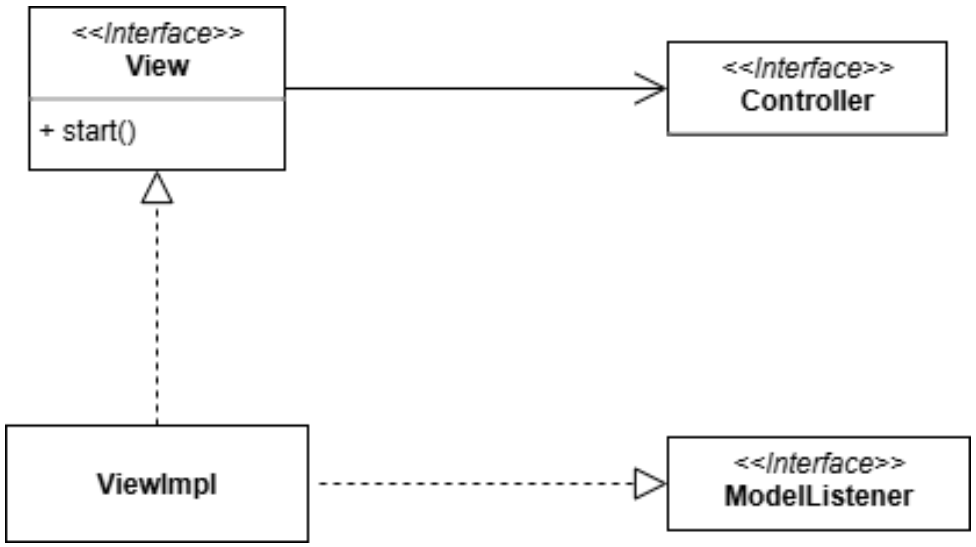
Schema UML del Model, con la sua implementazione ModelImpl e il collegamento a Controller e ModelListener

2.1.2 View

La View, realizzata tramite Swing, rappresenta l'interfaccia grafica dell'applicativo. Il suo compito è consentire l'interazione con l'utente, mostrando lo stato e l'avanzamento del gioco.

La View non contiene la logica del gioco: si limita a ricevere notifiche dal Model tramite l'interfaccia `ModelListener`, aggiornando di conseguenza la GUI. Quando l'utente compie un'azione (ad esempio selezionare una scelta o avviare la partita), la View notifica l'evento al Controller, che si occupa di gestire gli input.

Questa scelta di progettazione rende l'interfaccia `View` minimale, esponendo un unico metodo utilizzato per avviare la GUI, e assicura che rimanga una componente "passiva", responsabile unicamente dell'interfaccia grafica, e che le decisioni logiche riguardo al gioco vengano prese dal Model.



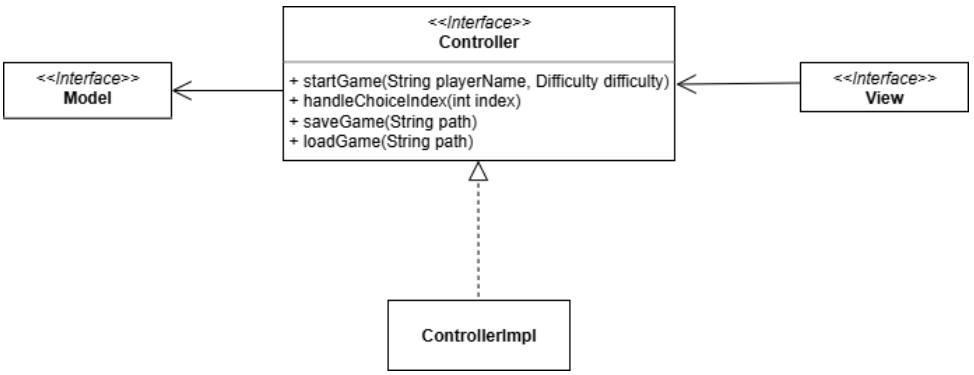
Schema UML della View, con la sua implementazione ViewImpl e il collegamento a Controller e ModelListener

2.1.3 Controller

Il Controller ha il compito di gestire le interazioni dell'utente provenienti dalla View, traducendole in comandi che il Model dovrà eseguire.

Non contiene logica di gioco né aggiorna direttamente la View. In questo progetto la sua implementazione concreta, `ControllerImpl`, si limita a invocare i metodi del Model (`startNewGame`, `chooseByIndex`, `saveGame`, `loadGame`) sulla base degli input ricevuti.

Questa scelta progettuale è volta a garantire che il Controller rimanga un componente minimale e "sottile", responsabile solo del collegamento fra input utente (View) e Model, senza dover accedere alle classi concrete del dominio o influenzare il flusso delle notifiche tra Model e View, e rendere il sistema più flessibile e facile da mantenere.



Schema UML del Controller, con la sua implementazione ControllerImpl e il collegamento a Model e View

2.2 Design dettagliato

Uno dei problemi principali che ho dovuto affrontare è stata la gestione delle scelte narrative e degli stati di gioco, poiché l'evoluzione della partita dipende sia dalle decisioni del giocatore sia dalle condizioni in cui si trova (vite rimanenti, inventario, infezione). Per questo motivo, una parte significativa dello sviluppo è stata dedicata alla definizione di un sistema con più componenti indipendenti, ciascuno con una responsabilità chiara e separata, che potessero rappresentare stanze, scelte ed effetti sul giocatore in maniera flessibile.

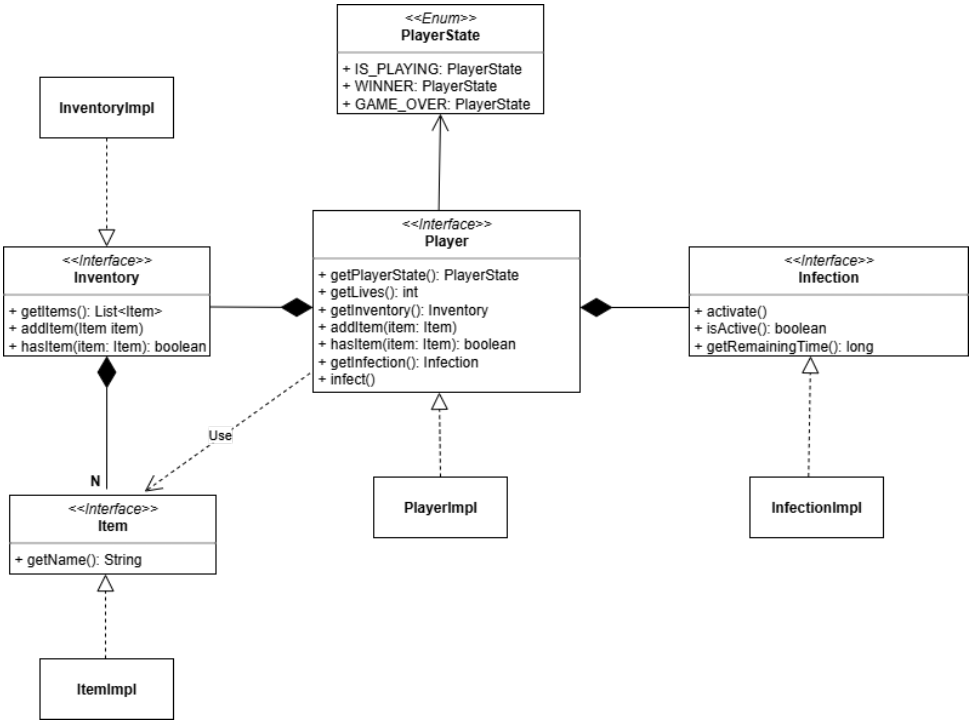
2.2.1 Player, Inventory, Item, Infection

Le interfacce `Player`, `Inventory`, `Item` e `Infection` sono state progettate per rappresentare le principali caratteristiche del giocatore: vite, oggetti raccolti e stato di infezione.

Il giocatore doveva possedere un inventario, con oggetti raccolti durante la partita, e una possibile infezione, contratta in una determinata stanza. Attraverso l'interfaccia `Player` si può accedere alle altre componenti (`Inventory`, `Item`, `Infection`), delegando a `Inventory` la gestione degli oggetti (istanze di Item) e a `Infection` la gestione della perdita periodica di vite solo quando il giocatore è infetto. Questa scelta ha permesso di evitare di concentrare troppa logica in un'unica classe, mantenendo il codice più leggibile e facilmente estendibile.

L'interfaccia `Player` è stata realizzata per offrire un'interfaccia semplificata, in quanto contiene `Inventory` e `Infection` e reindirizza metodi come `addItem()`, `hasItem()`, `infect()`.

Infine l'uso dell'enumerazione `PlayerState` permette un comportamento variabile del gioco a seconda dello stato corrente del giocatore (in partita, vittoria o game over).



Schema UML di Player, Inventory, Item e Infection, con le relative implementazioni e relazioni fra loro

2.2.2 Enemy

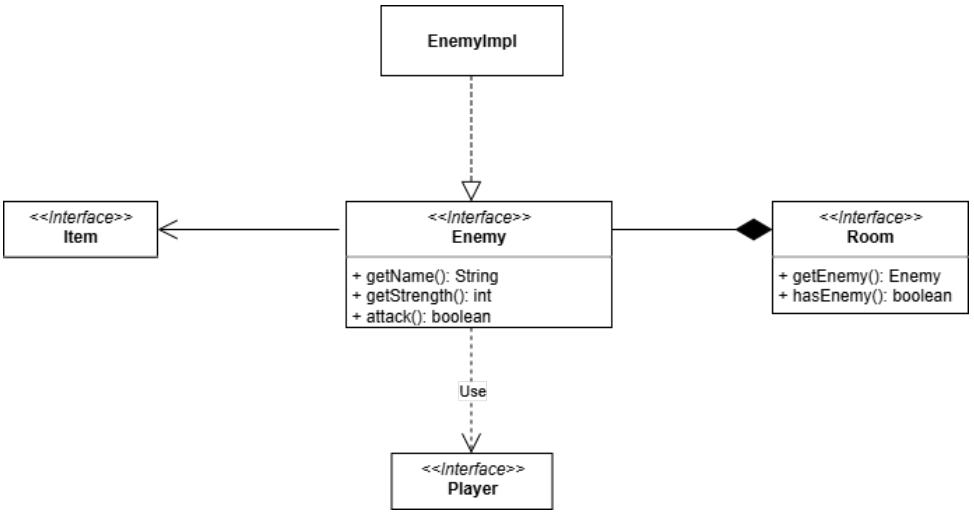
Per creare il nemico è stato necessario avere una logica di combattimento che dipendesse dalla sua forza (rappresentata dal numero di nemici) e dalla presenza di un oggetto (arma) utile al giocatore per difendersi.

Per questo motivo è stata creata l'interfaccia `Enemy`, che attraverso la sua implementazione riceve dal costruttore il livello di forza e l'oggetto richiesto.

Il nemico applica l'attacco tramite il metodo `attack(Player player)` che incapsula la logica di combattimento: l'attacco non avviene automaticamente all'ingresso di una stanza che possiede un nemico, ma solo in base alla scelta narrativa del giocatore. In questo modo è il giocatore a decidere se affrontare o meno un nemico, mantenendo il controllo sull'evoluzione della partita.

In particolare, il metodo viene invocato quando una `StoryChoice` contiene l'effetto `attack_enemy` (il giocatore sceglie se affrontare il nemico). L'esito dell'attacco dipende dalla forza del nemico e dall'eventuale possesso dell'oggetto richiesto.

L'interfaccia `Enemy` separa la responsabilità del combattimento dal resto della logica, incapsulando il comportamento di attacco nel metodo `attack(Player player)`. Questo rende il codice più chiaro ed estendibile, permettendo in futuro di introdurre nemici con logiche differenti.



Schema UML di Enemy, con la sua implementazione EnemyImpl e le relazioni con Item, Room e Player

2.2.3 StoryChoice

L'interfaccia `StoryChoice` rappresenta una scelta disponibile all'interno di una scena.

Per l'implementazione `StoryChoiceImpl` c'è stata la necessità di utilizzare due interfacce funzionali: `Supplier<Boolean>` per stabilire se la scelta è disponibile sulla base delle condizioni (ad esempio, in base agli oggetti posseduti dal giocatore), e `Consumer<Game>` per definire gli effetti da applicare sul gioco. Poiché `Supplier` e `Consumer` non sono serializzabili, la soluzione è stata quella di creare due interfacce, `SerializableSupplier` e `SerializableConsumer`, che estendono rispettivamente `Supplier` e `Consumer`, aggiungendo anche l'interfaccia `Serializable`.

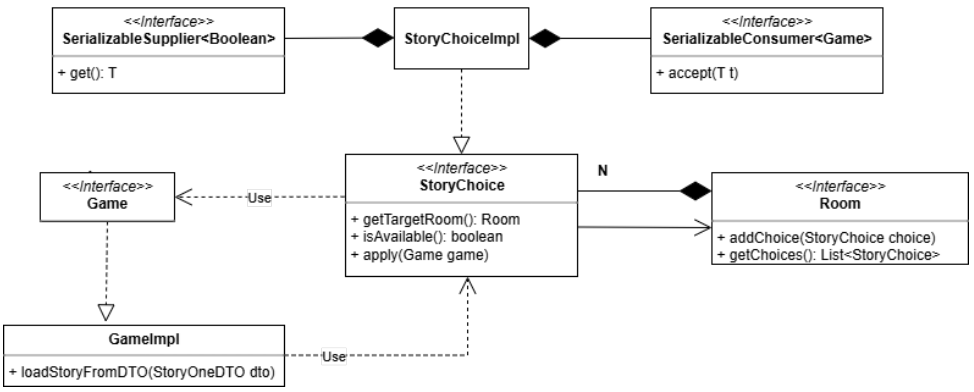
In questo modo, le lambda utilizzate rimangono serializzabili e compatibili con il salvataggio della partita.

Le condizioni di disponibilità di una scelta sono incapsulate tramite funzioni (lambda), che permettono di esprimere logiche diverse (ad esempio "sempre disponibile", "richiede un oggetto") in modo flessibile. Analogamente, anche gli effetti sono rappresentati come funzioni (lambda), e incapsulano le azioni da applicare sul gioco (ad esempio perdita di vite, aggiunta di un oggetto, attivazione infezione).

Le condizioni di disponibilità (`SerializableSupplier<Boolean>`) e gli effetti narrativi (`SerializableCons`
sono implementati come funzioni **lambda**, che rappresentano strategie di condizione o di effetto. Questa applicazione rappresenta il **pattern Strategy**, che permette di definire una famiglia di algoritmi o comportamenti e incapsularli dietro un'interfaccia comune. In questo modo, la logica di ogni scelta può variare dinamicamente (es. "sempre disponibile", "richiede un oggetto") senza cambiare la classe `StoryChoiceImpl`.

In particolare, nel progetto:

- le interfacce `SerializableSupplier<Boolean>` e `SerializableConsumer<Game>` svolgono il ruolo di **interfaccia strategia**;
- le diverse condizioni ed effetti delle scelte narrative sono le **implementazioni concrete**, espresse tramite **lambda expressions** (es. condizione "sempre disponibile", condizione "richiede un oggetto");
- la classe `StoryChoiceImpl` funge da *contesto*, perché utilizza le strategie invocando semplicemente `get()` o `accept(Game)` senza conoscerne l'implementazione concreta.



Schema UML di StoryChoice, con la sua implementazione StoryChoiceImpl, le relazioni con Game, GameImpl, Room, SerializableSupplier e SerializableConsumer

2.2.4 Room

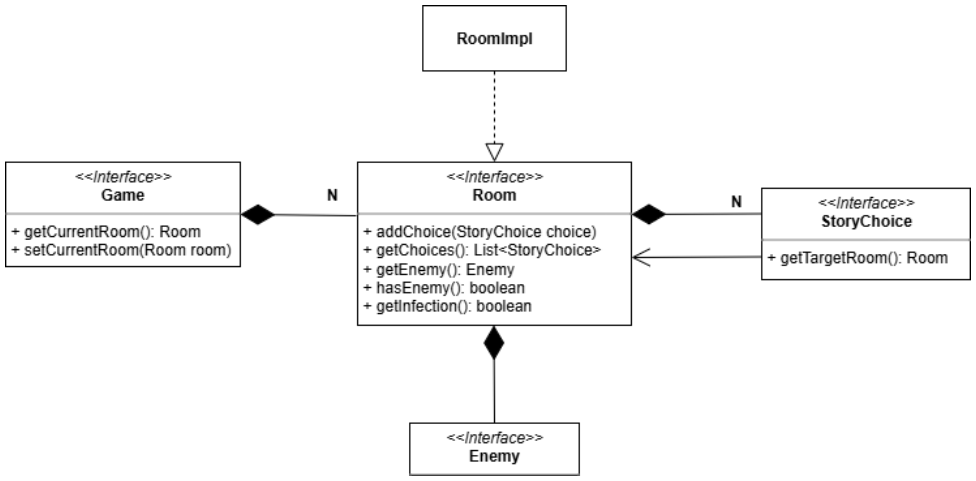
L'esigenza era quella di creare una scena narrativa che rappresentasse una stanza, contenente descrizione, una lista di scelte disponibili per l'utente e ulteriori elementi narrativi come nemici o aria infetta.

L'interfaccia `Room` è stata progettata per questo. L'implementazione `RoomImpl` evita duplicati di scelte disponibili, garantendo coerenza con la narrazione. Inoltre permette di mostrare alcuni

messaggi particolari subito all'ingresso della stanza, non dipendenti dalla scelta narrativa fatta dal giocatore.

La relazione tra `Room` e `StoryChoice` è di composizione, poiché una stanza contiene più scelte che la definiscono come unità narrativa completa.

In aggiunta, `Room` funge da punto di accesso a tutte le informazioni di una scena e raccoglie al suo interno gli elementi essenziali (scelte, nemico, infezione, messaggi).



Schema UML di Room, con la sua implementazione RoomImpl e le relazioni con Game, StoryChoice e Enemy

2.2.5 Game

L'interfaccia `Game` rappresenta lo stato globale della partita, includendo il giocatore (singolo in questa prima versione del gioco), il livello di difficoltà, stanza corrente, messaggi pendenti e gestione dell'infezione.

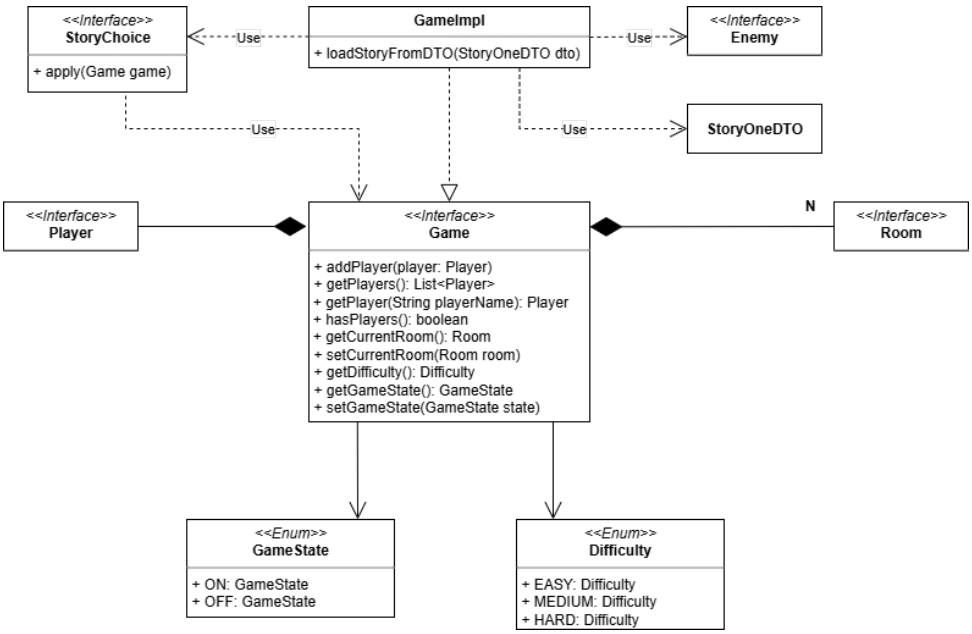
L'implementazione `GameImpl` incapsula la logica principale del gioco, gestendo l'avvio della partita, la gestione dell'infezione tramite un `Timer` periodico e la costruzione dinamica della storia a partire da un file JSON esterno.

Il `GameImpl` utilizza un `Timer` che ogni secondo controlla lo stato dell'infezione e, quando necessario, riduce le vite del giocatore. A ogni tick i cambiamenti vengono notificati ai `ModelListener`, permettendo alla View di aggiornarsi: questo realizza l'applicazione del **pattern Observer**. Poiché il `Timer` non è serializzabile, dopo il caricamento della partita viene ricreato a partire dal tempo residuo salvato.

L'enumerazione `GameState` rappresenta i possibili stati globali del gioco: `ON`, `OFF`, `PAUSE` (quest'ultimo non utilizzato in questa prima versione del gioco).

Infine, il metodo `loadStoryFromDTO()` si occupa della creazione di stanze e scelte a partire dai dati narrativi esterni (DTO), e viene utilizzato all'avvio di una nuova partita. Questo riflette il **pattern Factory**: un pattern "creazionale" che centralizza la responsabilità di creare oggetti, delegando la logica di istanziazione a un'unica classe, così che il resto del sistema utilizzi gli oggetti già pronti senza preoccuparsi di come vengano costruiti. Nel progetto questa

responsabilità è svolta da `GameImpl`, in particolare dal metodo `loadStoryFromDTO`, che funge da **Simple Factory**: a partire dai dati del file JSON, esso crea e collega tutte le istanze concrete del dominio (`RoomImpl`, `StoryChoiceImpl`, `EnemyImpl`, `ItemImpl`) e costruisce le relative condizioni ed effetti tramite `SerializableSupplier` e `SerializableConsumer`. In questo modo, il processo di creazione degli oggetti (es. stanze con nemici, scelte con condizioni ed effetti) rimane nascosto dentro la **factory** (`loadStoryFromDTO`), mentre Controller e View possono limitarsi a usare il `Game` già pronto.



Schema UML di Game, con la sua implementazione GameImpl, le relazioni con Player, Room, StoryChoice, Enemy, GameState e Difficulty

2.2.6 Salvataggio e Caricamento della Partita

L'applicativo realizza un sistema di salvataggio e caricamento in formato binario per il gioco. Tramite questa implementazione, l'intero stato del gioco viene serializzato su file e in seguito deserializzato e ricreato tramite `ObjectOutputStream` e `ObjectInputStream`.

Per implementare questo meccanismo, le interfacce del dominio sono state progettate per estendere direttamente `Serializable`. In questo modo ogni implementazione concreta risulta automaticamente serializzabile, semplificando la gestione del salvataggio e del caricamento. Da notare come alcuni campi della classe `GameImpl` (`Timer infectionTimer` e `Runnable onTick` di callback) sono stati dichiarati `transient`: il `Timer` perché non serializzabile (contiene thread interni alla JVM), e il `Runnable onTick` perché rappresenta un callback verso la View, che non fa parte dello stato da salvare. Dopo il caricamento, il metodo `afterLoad()` di `GameImpl` riavvia il `Timer` tramite `startInfectionTimer()`, mentre la callback `onTick` viene nuovamente collegata dal Model alla View, così da aggiornare periodicamente le informazioni mostrate.

La scelta di questa soluzione offre diversi vantaggi:

- Completezza: tutto lo stato viene salvato e ricostruito senza dover gestire manualmente ogni singolo attributo.
- Semplicità di implementazione: Java fornisce già pronti meccanismi standard di serializzazione come `Serializable`, che rendono il processo immediato.
- Coerenza: quando si ricarica il file corrispondente ad un determinato salvataggio, la partita riprende esattamente dal punto in cui era stata interrotta.

2.2.7 File JSON per la narrazione

La storia del gioco (stanze, descrizioni, scelte, condizioni, effetti) è scritta direttamente in un file esterno JSON. Al momento dell'avvio di una nuova partita, il metodo `loadStoryFromDTO()` carica questo file e costruisce dinamicamente tutte le stanze e le scelte (`RoomImpl`, `StoryChoiceImpl`), senza scrivere a mano l'intera narrazione nel codice.

Per gestire il parsing del file JSON è stata utilizzata la classe `StoryOneDTO` (Data Transfer Object), che funge da "contenitore intermedio": raccoglie le informazioni prese dal file JSON e le passa poi al gioco per creare le stanze e le scelte. Questo approccio consente di mappare direttamente la struttura del file in oggetti Java, e trasformarli nelle entità reali del gioco.

Questa logica è stata adottata per:

- Separare logica e contenuto narrativo: la logica rimane nel codice, la narrazione nei file esterni.
- Rendere più flessibile il sistema: la storia può essere aggiornata o ampliata modificando solo il file JSON.
- Manutenibilità: rende il progetto più ordinato e modulare (codice da una parte, dati narrativi dall'altra).

3 Sviluppo

3.1 Testing automatizzato

Per garantire che le funzionalità core restino corrette nel tempo e verificare il corretto comportamento delle componenti principali dell'applicativo, il progetto include test automatizzati implementati tramite `JUnit 5`.

È stato seguito un principio ispirato al test-driven development (TDD): nel caso un test non passasse, lo sviluppo non proseguiva fino alla correzione del problema e al superamento di tutti i test.

In questa sezione per ogni test sono riportati: obiettivo del test (scopo e comportamento atteso), scenario di test (i passaggi chiave eseguiti), implementazioni tecniche (classi e metodi coinvolti), risultati attesi (asserzioni e criteri di successo).

3.1.1 GameTest

L'**obiettivo** di questo test è verificare la logica dell'infezione (timer che scala le vite e terminazione partita a vite finite) e l'attacco dei nemici al giocatore.

Scenario di test (casi principali):

- Inizializzazione della partita con `initGame(initialLives)` (crea `GameImpl`, `Player`, `Room`, avvia la partita).
- Infezione: attivazione con `infect()`, simulazione dell'avanzare del tempo e verifica tramite asserzioni che le vite diminuiscano correttamente.
- Attacco del nemico con verifica di tre diversi casi:
  1. forza del nemico pari a 1, senza `Item` (arma) richiesta: perdita di 1 vita.
  2. forza del nemico pari a 1, con `Item` (arma) richiesta: nessuna perdita di vita.
  3. forza del nemico maggiore di 1, senza `Item` (arma) richiesta: immediato game over.

Implementazioni tecniche:

- Classi utilizzate: `GameImpl`, `PlayerImpl`, `InfectionImpl`, `EnemyImpl`, `RoomImpl`.
- Metodi principali invocati: `start()`, `infect()`, `getLives()`, `getPlayerState()`, `getGameState()`, `attack(Player)`.

Risultati attesi:

- Le vite scendono periodicamente quando l'infezione sul giocatore è attiva, grazie al controllo ciclico del `Timer`.
- Se il giocatore ha 1 vita residua, stato del `Player` impostato a `GAME_OVER`, e stato del gioco a `OFF`.
- Attacco del nemico coerente con forza e presenza/assenza dell'`Item` (arma) difensivo.

3.1.2 PlayerTest

L'**obiettivo** di questo test è verificare le funzionalità base del giocatore: vite, inventario, infezione del `Player`.

Scenario di test (casi principali):

- Inizializzazione del `Player` con costruttore: stato iniziale `IS_PLAYING`, id, nome, vite, inventario vuoto, infezione non attiva.
- Vite: aggiunta di vita tramite `addLife()` (con rispetto del limite massimo 6), perdita di 1 vita con `loseLife()`, perdita di tutte le vite (senza cambiare stato globale del gioco).
- Inventario: `ItemImpl`, verifica presenza di `Item` con `hasItem()` e aggiunta con `addItem(oggetto)`.
- Infezione del giocatore: `infect()` attiva l'infezione e `updateInfectionTimer()` aggiorna il tempo rimanente, con eventuale decremento di vite.

Implementazioni tecniche:

- Classi utilizzate: `PlayerImpl`, `InventoryImpl`, `ItemImpl`, `InfectionImpl`.
- Metodi principali invocati: `addLife()`, `loseLife()`, `getLives()`, `hasItem()`, `getInventory()`, `infect()`, `updateInfectionTimer()`.

Risultati attesi:

- Limite massimo di vite rispettato (6).
- Perdita e guadagno di vite corretti.
- Inventario aggiornato correttamente.
- Infezione attivabile dal `Player`, con gestione corretta del tempo rimanente e perdita di vite al termine del countdown.

3.1.3 SaveLoadGameTest

L'**obiettivo** di questo test è verificare e garantire che il salvataggio e caricamento binario preservi lo stato della partita e che il timer infezione riparta in modo corretto dopo il caricamento.

Scenario di test (casi principali):

- Inizializzazione del Model e avvio della partita.
- Salvataggio e caricamento di base: verifica che dopo il caricamento i giocatori siano stati correttamente ricostruiti (lista non nulla e non vuota).
- Stato del giocatore: salva la partita e verifica che il numero vite e l'inventario salvati, coincidano dopo il caricamento.
- Stanza corrente: salva la partita e verifica che la stanza corrente sia la stessa anche dopo il caricamento (identificata dall'id).
- Infezione post caricamento: infetta il `Player` con `infect()`, salva la partita e verifica che dopo il caricamento il giocatore risulti ancora infetto, e che il timer sia stato ricreato correttamente riprendendo dal tempo residuo salvato.

Implementazioni tecniche:

- Classi/Interfacce utilizzate: `ModelImpl`, `GameImpl`, `PlayerImpl`, `Room`.
- Metodi principali invocati: `startNewGame()`, `saveGame(path)`, `loadGame(path)`, `getPlayers()`, `getCurrentRoom()`.

Risultati attesi:

- Stato del giocatore (vite e inventario) rimasto invariato dopo il caricamento.
- Stanza corrente rimasta invariata dopo il caricamento.
- Infezione ancora attiva il timer ricreato dopo il caricamento (`remainingTime` ¿ 0), con gestione corretta del tempo residuo.

3.1.4 StoryChoiceConditionsTest

L'obiettivo di questo test è verificare le condizioni di disponibilità delle scelte (`StoryChoice`): `always` (sempre vera), `has_item` (vera se l'oggetto è presente nell'inventario), `not_has_item` (vera se l'oggetto non è presente nell'inventario).

NOTA: la condizione `not_has_item` è testata, ma non utilizzata in questa versione della narrazione.

Scenario di test (casi principali):

- Inizializzazione di `Player`, `Game`, `Room` di destinazione.
- Creazione di `StoryChoiceImpl` con verifica delle condizioni: `always`, `has_item`, `not_has_item`.
- Aggiunta di `Item` e verifica disponibilità tramite `isAvailable()`.

Implementazioni tecniche:

- Classi utilizzate: `StoryChoiceImpl`, `PlayerImpl`, `ItemImpl`, `GameImpl`, `RoomImpl`.
- Metodi principali invocati: `isAvailable()`, `addItem(Item)`, `hasItem()`.

Risultati attesi:

- Con la condizione `always` la scelta deve essere sempre disponibile al giocatore.
- Con la condizione `has_item` la scelta deve essere disponibile se il giocatore possiede nell'inventario l'oggetto richiesto.
- Con la condizione `not_has_item` la scelta deve essere disponibile se il giocatore non possiede nell'inventario l'oggetto richiesto.

3.1.5 StoryChoiceEffectsTest

L'**obiettivo** di questo test è verificare gli effetti sul gioco in generale, applicati da una determinata scelta: `if_has_item`, `lose_life_random`, `attack_enemy`, `add_item`, `lose_life`, `infect`, `winner`, `game_over`, `show_message`, e il cambio stanza previsto dal `targetRoom` della scelta.

NOTA: gli effetti `add_item`, `lose_life`, `infect`, `winner`, `game_over` e `show_message` sono testati ma non utilizzati come effetto diretto di una scelta, in questa versione della narrazione.

Scenario di test (casi principali):

- Inizializzazione di `Game`, `Player`, stanza corrente e stanza di destinazione.

- Creazione della lista di effetti (lambda `SerializableConsumer<Game>`) applicata a `StoryChoiceImpl`
- Verifica dei vari effetti, dopo l'utilizzo del metodo `apply(game)`.

Implementazioni tecniche:

- Classi utilizzate: `StoryChoiceImpl`, `GameImpl`, `PlayerImpl`, `EnemyImpl`, `ItemImpl`, `RoomImpl`.
- Metodi principali invocati: `apply(Game)`, `hasEnemy()`, `getEnemy()`, `attack(Player)`, `hasItem(item)`, `addItem(Item)`, `loseLife()`, `infect()`, `setPlayerState(PlayerState)`, `setGameState(GameState)`, `addPendingMessage(String)`, `usePendingMessages()`, `getCurrentRo`

Risultati attesi:

- Con l'effetto `if_has_item`, in caso di successo cambio stanza a `targetRoom`, altrimenti decremento di 1 vita.
- Con l'effetto `lose_life_random`, decremento di 1 vita, oppure presenza di un messaggio aggiunto.
- Con l'effetto `attack_enemy`, in caso di successo stato del `Player` invariato (`IS_PLAYING`), altrimenti decremento di 1 vita o stato del `Player` impostato a `GAME_OVER`.
- Con l'effetto `add_item`, presenza nell'inventario dell'Item aggiunto, e cambio stanza a `targetRoom`.
- Con l'effetto `lose_life`, decremento di 1 vita, e cambio stanza a `targetRoom`.
- Con l'effetto `infect`, `Player` infetto con un tempo residuo iniziale maggiore di 0, e la stanza corrente cambia a `targetRoom`.
- Con l'effetto `winner`, stato del `Player` impostato a `WINNER` e stato del gioco a `OFF`. Presenza del messaggio "Vittoria" (se previsto).
- Con l'effetto `game_over`, stato del `Player` impostato a `GAME_OVER` e stato del gioco a `OFF`. Presenza del messaggio "Game over" (se previsto).
- Con l'effetto `show_message`, presenza del messaggio nei pending messages, e cambio stanza a `targetRoom`.

3.2 Metodologia di lavoro

Lo sviluppo del progetto è stato svolto in autonomia, mantenendo comunque un metodo di lavoro strutturato e costante.

3.2.1 Workflow

La fase iniziale è stata dedicata all'analisi dei requisiti e alla progettazione dell'architettura, attività che hanno richiesto diverse ore di riflessione e lavoro tramite diagrammi UML, schemi narrativi del gioco e definizione dei principali pattern da adottare.

Questa prima fase è stata molto importante per organizzare il progetto nelle tre componenti indipendenti dell'architettura **MVC** (**Model, View, Controller**). Grazie a questo lavoro l'implementazione delle interfacce e classi concrete è risultata più lineare e senza particolari ostacoli.

Anche durante lo sviluppo, il lavoro è stato portato avanti in maniera indipendente ma seguendo un flusso di lavoro chiaramente definito: prima la realizzazione del Model (con le entità di dominio e la logica di gioco), successivamente il Controller (deleghe verso il model), e infine la View (GUI `Swing`). Durante questa fase è stata progressivamente sviluppata la parte di testing automatizzato tramite JUnit 5, al fine di verificare e prevenire eventuali regressioni.

Infine l'ultima parte del lavoro è stata focalizzata su controlli e test generali, verificando il corretto funzionamento complessivo del gioco

3.2.2 Realizzazione

- **Package main.model**
  - Interfaccia `Model` e implementazione `ModelImpl`. Interfaccia `ModelListener` per l'implementazione del pattern **Observer**.
  - Package `main.model.game`: interfaccia `Game` e implementazione `GameImpl` per la gestione della logica di gioco, infezione con `Timer` e costruzione dinamica delle stanze da JSON. Classe `StoryOneDTO` (Data Transfer Object) per gestire il parsing del file JSON. Interfacce funzionali `SerializableSupplier<Boolean>` (per gestire la disponibilità delle scelte) e `SerializableConsumer<Game>` (per la creazione della lista di effetti). Enum `GameState`, `Difficulty`, rispettivamente per i possibili stati del gioco e i livelli di difficoltà.
  - `Package main.model.player` per le condizioni riguardanti il giocatore come vite, inventario, e infezione: interfacce `Player`, `Item`, `Inventory`, `Infection` e relative implementazioni delle classi. Enum `PalyerState`, per i possibili stati del giocatore.
  - Package `main.model.story`: interfacce `Room`, `StoryChoice`, `Enemy` e relative implementazioni delle classi.
- Package `main.controller`
  - Interfaccia `Controller` e implementazione `ControllerImpl`, responsabile della mediazione tra `View` e `Model`.
- Package `main.view`

  - Interfaccia `View` per rappresentare la GUI con l'unico metodo per avviarla. Implementazione della classe `ViewImpl`, realizzata con Swing (`JFrame`, `CardLayout`), che implementa anche l'interfaccia `ModelListener` per l'applicazione del pattern **Observer**. Classe di utilità `ViewTexts` per l'utilizzo di alcuni testi fissi.

- Package `main.app`
  - Classe `Main`, responsabile dell'avvio dell'assemblaggio delle componenti MVC, e dell'avvio dell'applicativo.

- Package `test`
  - Classi `GameTest`, `PlayerTest`, `SaveLoadGameTest`, `StoryChoiceConditionsTest`, `StoryChoiceEffectsTest` per le verifiche con test JUnit.

- Altri contributi
  - Realizzazione dei diagrammi UML.
  - Scrittura del file JSON `story_01.json` contenente l'intera narrazione, scelte con condizioni ed effetti.
  - Implementazione del salvataggio e caricamento della partita tramite `ObjectOutputStream` e `ObjectInputStream`.
  - Stesura della relazione del progetto.

3.3 Note di sviluppo

- Utilizzo di **lambda expressions** per implementare condizioni ed effetti delle scelte narrative, inizialmente con `Supplier` e `Consumer`, poi sostituite con le interfacce serializzabili custom `SerializableSupplier` e `SerializableConsumer`.
- Utilizzo di **Stream** per filtrare e raccogliere le scelte disponibili, e per trasformare gli oggetti dell'inventario.
- Gestione di un timer di infezione tramite la classe `java.util.Timer` (`GameImpl`) e `TimerTask`, con logica periodica di controllo che decrementa progressivamente le vite del giocatore. Questa scelta è stata fatta perché il timer appartiene alla logica di dominio, e non alla View (che invece utilizza `javax.swing.Timer` solo per aggiornare la GUI).
- Definizione di **interfacce serializzabili** per uniformare il salvataggio e caricamento (le interfacce di dominio estendono direttamente `Serializable`).

3.3.1 Sorgenti

Link al progetto su GitHub: https://github.com/Luca9119/Progetto_PMO_2024_2025_PiovaticciLuca