Relazione Progetto PMO

Applicativo di un videogioco di simulazione di vita.

Nasrine Aboufaris
n.aboufaris@campus.uniurb.it
321885

Nicole Fabbri
n.fabbri5@campus.uniurb.it
321119

Alessia Giuseppetti
a.giuseppetti@campus.uniurb.it
322984

Alice Neri
a.neri14@campus.uniurb.it
32165

A.A. 2024 - 2025

Indice

1. Analisi

L'obiettivo del progetto è la realizzazione di un'applicazione che simuli il funzionamento di un life simulator, ispirato alla console My Life.
Il sistema è composto da:

- Un personaggio principale, controllato dall'utente.
- Tre NPC, ciascuno responsabile dell'assegnazione di una missione.
- Sei stanze, ognuna dotata di oggetti diversi e interattivi.
- Una mappa, per lo spostamento tra le stanze.
- Un sistema di bisogni del personaggio, come energia, fame e igiene.

Il flusso del sistema prevede che il giocatore crei il proprio personaggio e successivamente acceda ad una stanza tramite la mappa. L'ingresso in una stanza permette l'interazione con l'eventuale NPC associato e l'utilizzo degli oggetti presenti. L'esecuzione delle azioni ha un effetto sui bisogni del personaggio e, livelli troppo bassi di determinati bisogni, bloccano la possibilità di svolgere azioni. Le missioni, assegnate dagli NPC, si considerano completate quando tutti i relativi requisiti risultano soddisfatti.

La partita può concludersi in due esiti:

- Vittoria, quando vengono completate tutte le missioni degli NPC.
- Sconfitta, se uno dei bisogni del personaggio scende a zero prima del completamento.

1.1. Requisiti

1.1.1. Requisiti minimi

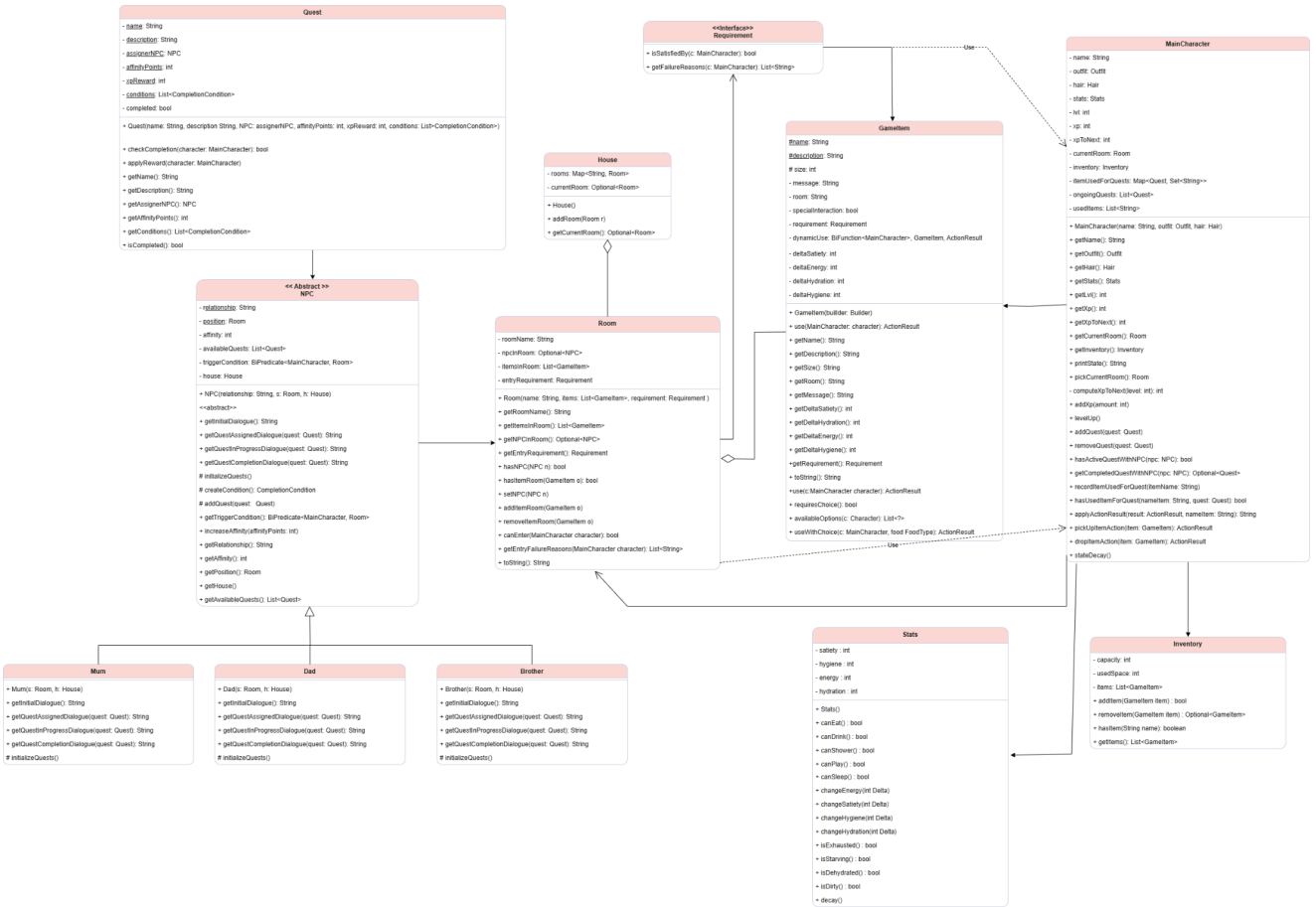
- ★ Permettere all'utente di inserire il nome del personaggio e personalizzarlo all'avvio.
- ★ Mappa per l'accesso alle varie stanze.
- ★ Implementazione di tre NPC e una missione per ciascuno di essi.
- ★ Scelta della stanza in cui far spostare il personaggio.
- ★ Interazione con gli NPC per svolgere le missioni.
- ★ Display dei bisogni del personaggio.
- ★ Interazione esclusiva con gli oggetti della stanza in cui è situato il personaggio.

1.1.2. Requisiti opzionali

- ★ Livello di affinità con l'NPC che aumenta una volta completata la sua missione.
- ★ Inventario del personaggio che contiene un determinato numero di oggetti.

1.2. Modello del dominio

L'entità principale della simulazione è **MainCharacter** che rappresenta il personaggio principale del gioco. Attraverso essa l'utente può interagire con le altre entità maggiori: NPC, House e Item.
Gli **NPC** sono tre istanze che svolgono il medesimo ruolo: assegnare la propria **Quest** al MainCharacter e, quando l'Affinity con il personaggio supera una determinata soglia, possono collaborare durante l'esecuzione di specifici obiettivi.
La **House** è articolata in più **Room**, ciascuna contiene **Item** diversi con i quali il MainCharacter può interagire per svolgere le Quest. Entrare in una Room abilita le azioni sugli Item presenti e, quando previsto, l'interazione con l'NPC associato.
MainCharacter mantiene una scheda di **Stats**, i cui valori vengono modificati dalle azioni sugli Item. Valori troppo bassi di determinati bisogni possono bloccare determinate azioni del personaggio, come esplicitato nei **Requirements**.



2. Design

2.1. Architettura

L'architettura del sistema è basata sul pattern Model-View-Controller, che permette di separare la logica del gioco, contenuta all'interno del Model, dalla rappresentazione su schermo(View) e dal controllo degli input dell'utente(Controller). In questo modo si favorisce la modularità, manutenibilità e testabilità del codice.
Il Model rappresenta lo stato e la logica di dominio del gioco. Al suo interno ci sono le entità principali come MainCharacter, NPC , Room , House, GameItem, Stats, QuestSystem e le regole di interazione tra di esse. Il Model è stato progettato in modo che possa essere completamente indipendente dalla View e dal Controller.
La View presenta lo stato del Model e raccoglie gli input dell'utente senza conoscere i dati interni del dominio.
Infine, il Controller traduce gli eventi della View in azioni su Model e propaga alla View gli aggiornamenti rilevanti.

2.1.1. Model

Il Model mantiene lo stato di gioco e le regole di dominio. Comprende MainCharacter(dove vengono gestite le statistiche, progressione XP/livello, stanza corrente, inventario e oggetti usati), House/Room(all'interno di cui sono presenti GameItem e NPC con requisiti di accesso), Quest/QuestSystem(stato e avanzamento missioni), Requirement(regole d'uso di GameItem e ingresso stanze) e ActionResult(esito delle azioni).

2.1.2. View

2.1.3. Controller

2.2. Design dettagliato

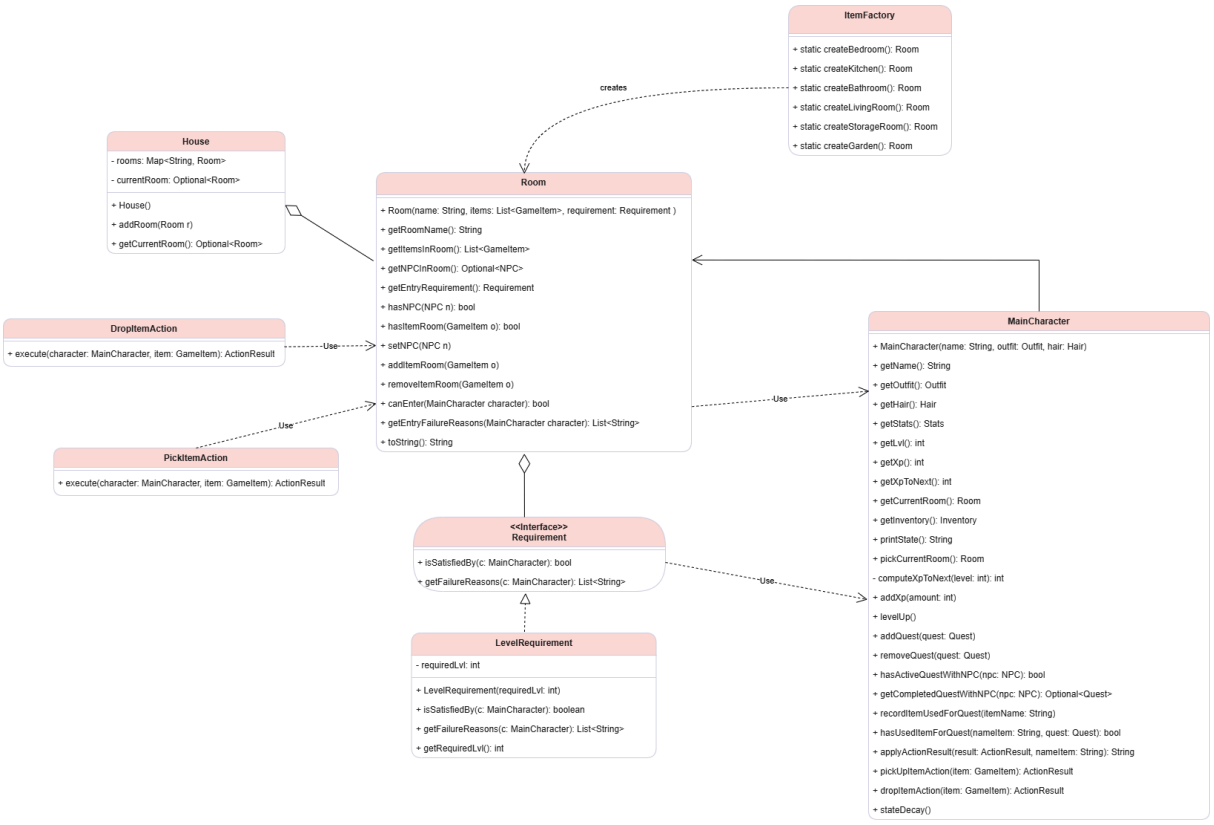
2.2.1. Aboufaris

★ Room

L'entità **Room** modella una singola stanza della casa e incapsula il nome, una collezione di GameItem interattivi ed un riferimento opzionale all'NPC presente. All'interno di Room non è garantita la presenza di un NPC, per cui si è fatto uso di Optional, in modo da gestire i casi di null in maniera più sicura e pulita. La classe offre diversi metodi che permettono di rimuovere o aggiungere GameItem, o anche di verificarne la presenza. L'accesso a Room viene gestito mediante l'interfaccia Requirement, così che i criteri possano essere estendibili senza modificare Room.

★ House

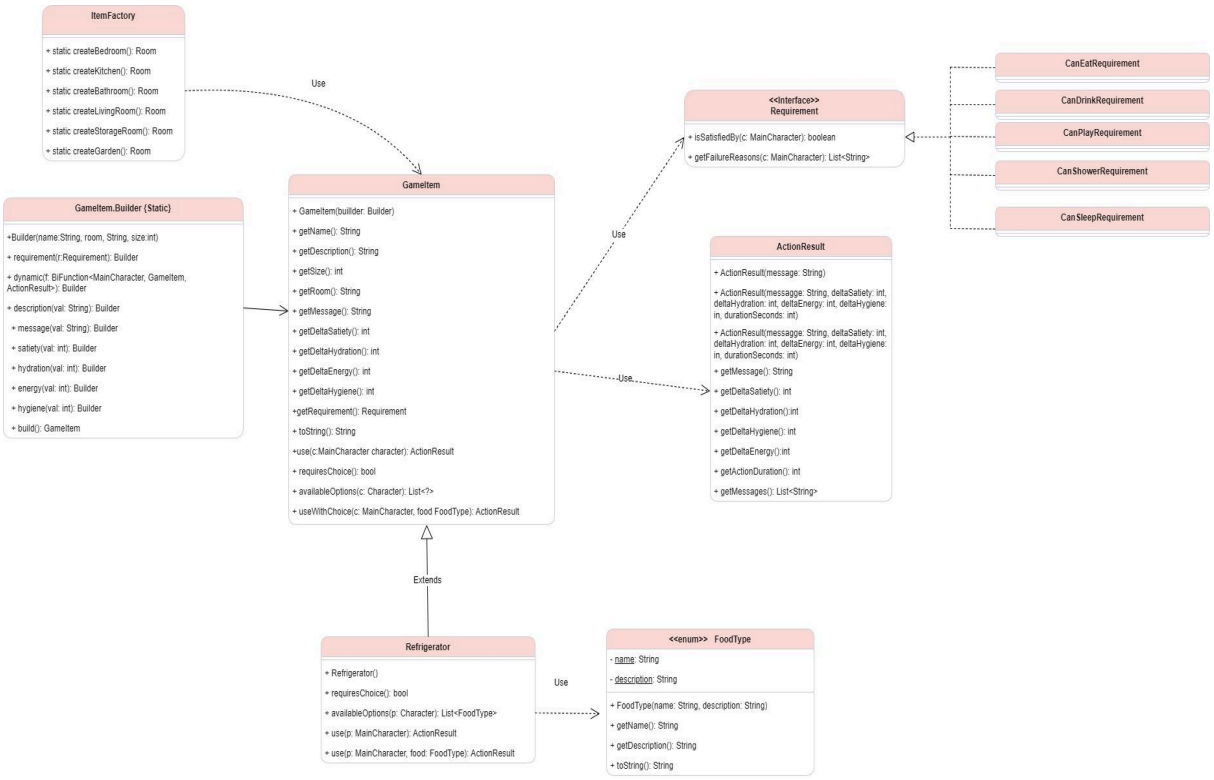
L'entità **House** aggrega le stanze tramite una mappa `Map<String, Room>` basandosi sull'accesso per nome e mantiene un `Optional<Room> currentRoom` per rappresentare la Room corrente in cui si trova il MainCharacter. Si è deciso di fare uso dell'Optional in quanto all'avvio del gioco, l'utente non si trova all'interno di una Room specifica. Metodi come `enterRoom()` ed `exitRoom()` gestiscono l'entrata/uscita dalla Room.



★ GameItem e ItemFactory

L'entità **GameItem** rappresenta un oggetto interattivo ed è costruita con un Builder interno per gestire in modo leggibile e sicuro le numerose proprietà opzionali. Si tratta di una classe che detiene molti parametri opzionali, per cui la scelta del Builder Pattern era la migliore per leggibilità e manutenzione del codice. Ogni item ha nome, descrizione, Requirement d'uso e un'interazione con il personaggio che modifica le Stats. Viene offerta anche una funzione `dynamic(MainCharacter, GameItem) -> ActionResult` la cui presenza calcola effetti e durata di un'azione in modo contestuale. La funzione `use(MainCharacter c)`, che è la funzione principale con cui il MainCharacter interagisce con il GameItem, verifica i vincoli e, se presente applica la funzione dinamica, in ogni caso restituisce un ActionResult.

La creazione di Room e GameItem è centralizzata in **ItemFactory**, che compone Room con la lista di GameItem e il relativo LevelRequirement, e ospita le costanti di bilanciamento (durate, costi/benefici) mantenendo la logica vicino alla definizione degli oggetti. Per elementi come Letto, Divano, o Doccia, il Builder accetta una lambda di uso dinamico, permettendo di variare soglie o effetti.



★ Requirement

L'interfaccia **Requirement** applica il pattern Strategy ai vincoli d'uso dei GameItem e d'accesso alle Room. Definisce due metodi: `isSatisfiedBy(MainCharacter c)` e `getFailureReasons(MainCharacter c)`. Il primo interroga le statistiche del personaggio per controllare se un'azione può essere svolta o meno e, infine, restituisce un boolean. Il secondo serve a spiegare le ragioni di fallimento di un'azione e lo fa restituendo una lista di String, di modo da far capire chiaramente all'utente i motivi del fallimento.

★ Stats

L'entità **Stats** gestisce lo stato fisiologico del personaggio, tramite dei valori mantenuti nel range 0-100 e offrendo sia controlli sintetici che flag puntuali. Le soglie di dominio sono centralizzate come costanti private e usate da tutti i metodi. I metodi presenti si dividono in due categorie: quelli che verificano la possibilità di eseguire determinate azioni in base alle soglie correnti dei bisogni e quelli di modifica dello stato del personaggio, i quali garantiscono che i limiti non vengano superati, tramite un metodo ausiliario clamp().

2.2.2. Fabbri

★ MainCharacter, Outfit e Hair

L'entità **MainCharacter** è stata progettata come componente del Model (MVC) e applica il principio della composizione, delegando aspetti specifici a Stats, Outfit e Hair per mantenere alta coesione e isolare le regole di dominio. L'accesso ai valori delle statistiche è centralizzato nella classe Stats mentre la progressione (lvl, xp, xpToNext) e il contesto (stanza corrente, oggetti già utilizzati) risiedono in MainCharacter in quanto parti dello stato persistente del personaggio, mentre l'aggiornamento periodico delega a stats.decay() come fonte unica di verità. Le interazioni con il World e Quest sono restituite come ActionResult di modo da ridurre l'accoppiamento, prevenire incoerenze e semplificare test e manutenzione.

L'entità **Outfit** è stata realizzata come enum tipizzato, in quanto l'insieme degli stili di vestiti è finito, semplificando così la validazione. All'interno di MainCharacter è utilizzata per composizione, così da separare la personalizzazione statica del personaggio da quella dinamica delle statistiche ma garantendo comunque la possibilità di aggiungere nuove opzioni facilmente. L'API minimale facilita la visualizzazione nella View, mantenendo chiaro il contratto del Model.

L'entità **Hair** è stata implementata come enum e, anche qui, si applica il pattern Value Object immutable, che previene stati non validi e rende l'entità sicura da condividere. In MainCharacter viene usata per composizione, mantenendo il personaggio coerente. Questa scelta mantiene il dominio consistente e permette l'aggiunta di nuovi stili senza impatti sul codice client.

★ FoodType

FoodType è modellato come enum arricchito, in linea con Hair e Outfit: funge da catalogo tipizzato e immutabile che centralizza i metadati dei cibi. In MainCharacter e negli oggetti di consumo, le regole leggono questi parametri dall'enum, mantenendo bassa la dipendenza dal caso specifico. L'evoluzione nuovamente tratta semplicemente di aggiungere una costante con i suoi metadati.

★ QuestSystem

QuestSystem gestisce stato e avanzamento delle missioni in modo event-driven, in linea con l'impostazione usata per MainCharacter e gli enum tipizzati. Sfrutta il pattern Observer su eventi di gioco (es. ingresso in stanza, uso oggetto, dialogo con NPC) per aggiornare gli Quest.

Le missioni sono modellate con un QuestState che funge da mini-State pattern e può assumere una tra tre alternative:

- attiva
- completata
- fallita

L'aggiunta di nuove quest o trigger è nuovamente additiva: si registrano nuovi handler di evento o nuovi tipi di Quest, senza toccare l'API pubblica né la UI. Il collegamento con MainCharacter è indiretto e a basso accoppiamento: i Controller propagano gli eventi, QuestSystem aggiorna il progresso e rende disponibile lo stato per la View.

2.2.3. Giuseppetti

★ NPC

L'entità **NPC** modella un personaggio non giocabile all'interno della casa e contiene informazioni riguardo il tipo di relazione con il MainCharacter, la posizione fissa (Room), il livello di affinità e una lista di quest disponibili. La classe mantiene anche un riferimento alla House per permettere agli NPC di accedere agli oggetti presenti nelle varie stanze durante l'inizializzazione delle quest. Si è scelto di utilizzare un pattern Template Method, e di rendere quindi NPC una classe astratta per definire un comportamento comune per tutti i personaggi, delegando alle sottoclassi concrete l'implementazione dei dialoghi specifici e l'inizializzazione delle quest personalizzate.
La classe offre metodi concreti per la gestione dell'affinità tramite increaseAffinity(), che mantiene il valore nell'intervallo 0-100, il quale dovrebbe aiutare il giocatore a completare determinate quest (funzionalità ancora da implementare), e per l'aggiunta di quest mediante addQuest(). È presente inoltre un BiPredicate<MainCharacter, Room> per definire condizioni di trigger per determinare quando un NPC deve interagire con un giocatore e consegnare una determinata quest, per rendere il sistema più dinamico.

★ Mum, Dad and Brother

Le classi **Brother**, **Dad** e **Mum** estendono NPC fornendo implementazioni specifiche per ciascun membro della famiglia. Ogni classe definisce dialoghi personalizzati che riflettono la personalità del personaggio e le quest specifiche date al giocatore. Il metodo initializeQuests() è implementato seguendo un pattern comune: si recupera la Room specifica dalla House, si cerca il GameItem necessario tramite stream filtering, e se l'oggetto esiste si crea e si aggiunge una Quest.

★ Quest

L'entità **Quest** rappresenta un obiettivo assegnato da un NPC al MainCharacter e contiene informazioni quali nome, descrizione testuale, riferimento all'NPC che assegna la quest, ricompense (punti affinità e esperienza) e una lista di condizioni di completamento. La classe mantiene uno stato booleano completed per tracciare se la quest è stata portata a termine. Il metodo checkCompletion(MainCharacter) implementa la logica di verifica: itera attraverso tutte le CompletionCondition e restituisce true solo se ogni condizione è soddisfatta, marcando contemporaneamente la quest come completata. Il metodo applyReward(MainCharacter). La lista di condizioni viene copiata nel costruttore per prevenire modifiche esterne indesiderate.

★ CompletionCondition

L'entità **CompletionCondition** rappresenta un singolo criterio di completamento per una quest e si basa sull'utilizzo di uno specifico GameItem. La classe mantiene un riferimento all'item richiesto e delega al MainCharacter la verifica tramite il metodo

checkCompletion(MainCharacter), che interroga se il personaggio ha già utilizzato l'oggetto per scopi di quest mediante hasUsedItemForQuest(GameItem).

★ LevelRequirement

L'entità **LevelRequirement** implementa l'interfaccia Requirement per rappresentare un vincolo basato sul livello del MainCharacter. La classe contiene un valore requiredLvl che specifica il livello minimo necessario, validato nel costruttore per garantire che sia almeno 1, lanciando un'IllegalArgumentException in caso contrario. Il metodo isSatisfiedBy(MainCharacter) confronta il livello corrente del personaggio con quello richiesto, restituendo true se il requisito è soddisfatto. Il metodo getFailureReasons(MainCharacter) costruisce una lista di stringhe che descrive il motivo del fallimento di una determinata azione, includendo sia il livello richiesto che quello attuale del personaggio.

2.2.4. Neri

★ **ActionResult**

★ **DropItemAction e PickItemAction**

3. Sviluppo

3.1. Testing automatizzato

3.1.1. HouseTest

L'obiettivo del test è quello di validare le classi **House e Room**: si verificherà la corretta creazione e gestione delle stanze, così come la navigazione, la presenza di NPC in determinate stanze, i requisiti di accesso e la gestione dei casi limite.

1. **Creazione e gestione base della casa.** In questo test si crea la casa per poi aggiungere al suo interno 6 stanze predefinite. Si verifica che la casa contenga il numero corretto di stanze e si controlla che tutte le stanze siano presenti con i nomi attesi. Inoltre si testa la navigazione tra stanze e si verifica il comportamento della classe in caso di stanze inesistenti.

2. **Gestione dell'NPC all'interno della stanza.** In questo test si assegnano NPC specifici a stanze diverse e si verifica che ogni stanza riporti correttamente la presenza del suo NPC. Si verifica il comportamento del sistema ad un tentativo di assegnare un NPC duplicato e, infine, si controlla che nelle stanze a cui non sono stati assegnati NPC, non ce ne siano.

3. **Verifica requisiti di accesso alle stanze.** Il test controlla che i requisiti per l'accesso alle stanze funzionino correttamente e che il personaggio possa accedere solo alle stanze concesse dal suo livello. Si aumenta il livello del personaggio e si verifica l'accesso alle stanze con requisito di livello inferiore e dei livelli superiori. Si controllano i messaggi di fallimento per gli accessi negati.

4. **Gestione item nelle stanze.** Si verifica che le stanze gestiscano correttamente gli Item. Si verifica che ogni stanza contenga il numero corretto di item all'inizializzazione, in seguito si aggiunge un item di test a una stanza per verificare l'inserimento corretto e si esegue un'operazione analoga per la rimozione. Infine si testa la verifica di presenza per item non esistenti.

5. **Gestione degli errori.** Si tenta di entrare in una stanza inesistente e si verifica che non venga impostata alcuna stanza corrente. Si tenta di uscire senza stanza corrente impostata e si verifica che venga lanciata l'eccezione appropriata.

3.1.2. GameItemTest

L'obiettivo di questo test è quello di verificare:

1. **Costruzione corretta via Builder di GameItem.** Si verifica che i parametri siano stati impostati correttamente e si controlla che i valori non specificati assumano il valore di default.

2. **Azione bloccata se le statistiche del personaggio non rispettano i Requirement.** In questo test si crea un GameItem con un determinato requisito (CanSleepRequirement) e si tenta di utilizzare l'item con un personaggio che non soddisfa il requisito. Si verifica inoltre che vengano restituiti i messaggi di fallimento appropriati.

3. **Verificare il funzionamento di GameItem con effetti statici.** Anche in questo caso si crea un GameItem con un requisito, si modificano le Stats del personaggio per soddisfare il requisito e si utilizza l'Item, verificando il risultato.

4. **Verificare il funzionamento di GameItem con effetti dinamici.** Si crea un GameItem con funzione dinamica che modifica l'energia guadagnata in base all'energia corrente e poi si testa l'item con diversi livelli di energia del personaggio. Infine si verifica che gli effetti applicati corrispondano alla logica condizionale.

3.1.3. ItemFactoryKitchenTest

Lo scopo di questo test è quello di validare `ItemFactory.createKitchen()` e il comportamento dei tre item della cucina.

1. **Creazione cucina.** In questo test viene creata la cucina tramite il metodo `ItemFactory.createKitchen()`. Si verifica che il nome della stanza sia "Cucina", si controlla che il requisito di accesso sia un LevelRequirement di livello 1 e si conferma che la stanza contenga esattamente 3 item.

2. **Verifica attributi degli item.** Per ogni item all'interno della cucina si verifica che attributi come nome, stanza di appartenenza, dimensione attesa e comportamento rispetto alle scelte sia giusto.

3. **Comportamento di "Fornelli".** Essendo un item con un comportamento dinamico, si verifica il funzionamento con diversi livelli di energia. Inoltre "Fornelli" ha anche un requirement che se non viene soddisfatto non ne permette l'uso.

4. **Comportamento di "Frigorifero".** In questo caso si verifica che il frigorifero richieda la scelta dell'utente, si testa l'interazione base che richiede la selezione di cibo e poi si testa l'uso con una scelta specifica. Infine si verifica che le statistiche vengano modificate correttamente.

5. **Comportamento di "Lavandino".** Nuovamente per verificare la correttezza dell'azione dinamica.

6. **Requisito di accesso alla cucina.** Si crea un personaggio di livello base e si verifica che soddisfi il requisito di livello 1.

3.1.4. ItemFactoryBedroomTest

In modo analogo al test ItemFactoryKitchenTest, qui si cerca di valutare il comportamento di `ItemFactory.createBedroom()` e il comportamento dei **GameItem** presenti.

1. **Creazione stanza.** Si verifica che la stanza venga creata correttamente, controllando nome, requisito di accesso e numero di item.

2. **Attributi GameItem.** In questo test si verifica che ogni item nella camera da letto abbia gli attributi corretti.

3. **Funzionamento del letto.** Il letto dispone sia di un requirement che non ne permette l'uso quando il personaggio ha energia elevata, sia di uso dinamico. Questo significa che il guadagno di energia varia in base al livello di energia del personaggio.

4. **Funzionamento del computer.** Il computer viene usato in maniera dinamica e la perdita di energia dipende dal livello di energia del personaggio.

5. **Funzionamento dell'armadio.** Si verifica che gli effetti statici di "Armadio" vengano applicati correttamente.

3.1.5. RefrigeratorTest

Essendo **Refrigerator** un GameItem che estende i comportamenti di un GameITem base, abbiamo deciso di verificarne la validità.

1. **Creazione corretta del frigorifero.** Si verifica che il nome, la stanza e la dimensione siano corretti. Inoltre si verifica che richieda la scelta utente e si controlla che il requisito sia di tipo CanEatRequirement.

2. **Offerta delle opzioni alimentari.** Nel secondo test si verifica che il frigorifero offra la lista completa delle opzioni alimentari definite nell'enum FoodType.

3. **Messaggio di selezione.** Si verifica che l'uso base del frigorifero restituisca il messaggio di prompt che richiede all'utente di selezionare il cibo. In ogni caso l'uso del frigorifero è possibile solo se si soddisfa il CanEatRequirement.

4. **Uso del frigorifero con una scelta alimentare specifica.** Per ogni tipo di cibo nell'enum, si modifica la sazietà del personaggio per soddisfare il requisito, si usa il frigorifero con la scelta del cibo corrente e infine si verifica che messaggio ed effetti corrispondano al cibo selezionato.

5. **Gestione degli errori.** Si verifica il funzionamento del frigorifero in caso di scelta non valida, in questo caso viene restituito un messaggio di errore.

6. **Gestione dei requisiti insoddisfatti.** Se CanEatRequirement non è soddisfatto, il frigorifero dovrebbe restituire messaggi d'errore appropriati sia per l'uso base che per l'uso con la scelta.

7. **Gestione dei requisiti soddisfatti.** Si modifica la sazietà del personaggio per soddisfare il requisito e poi si utilizza il frigorifero, prima senza scelta e poi con scelta specifica, verificando messaggio ed effetti.

3.1.6. StatsTest

Lo scopo del test è quello di validare la classe **Stats**. Si verifica che inizializzazione, modifiche, decadimento naturale, stati critici e condizioni di azione funzionino nella maniera corretta.

1. **Inizializzazione e modifiche.** Si verifica che le statistiche siano state inizializzate correttamente a 100, che i metodi di modifica funzionino appropriatamente e che i valori rimangano entro i limiti consentiti. Inoltre si testa il decadimento naturale e gli stati critici.

2. **Condizioni per mangiare.** Si verifica che il metodo `canEat` restituisca true solo quando sono soddisfatte tutte le condizioni: energia > 40, igiene > 50 e sazietà < 80

3. **Condizioni per dormire.** Si verifica che il metodo `canSleep` restituisca true solo quando energia <70 e igiene > 30;

4. **Condizioni per bere.** Verifica che il metodo `canDrink` restituisca true solo quando idratazione < 70;

5. **Condizioni per lavarsi.** Verifica che il metodo `canShower` restituisca true solo quando igiene < 70 e energia > 20;

6. **Condizioni per giocare.** Verifica che il metodo `canPlay` restituisca true solo quando energia > 20, indipendentemente dalle altre statistiche.

3.2. Metodologia di lavoro

Strumenti utilizzati:

- GitHub: utilizzato come repository principale;
- Git: utilizzato per il versionamento e lo sviluppo cooperativo;
- Eclipse: utilizza come IDE;
- draw.io: utilizzato per la creazione e la modellazione dell'UML;
- Google Documenti: utilizzato per la redazione della relazione.

3.3. Sorgenti

Link GitHub: https://github.com/nissan345/progetto_pmo_202425.git