NOT A SABINO'S SAGA

Gruppo: MS_C

Componenti: Nido Marianna *[matricola 681041],* Regini Sebastiano *[matricola 677636]*

Descrizione Generale del Gioco

Il gioco è ambientato nell'età contemporanea ed ha come protagonista un uomo ingiustamente incarcerato per un crimine che non ha commesso.

Esso è stato condannato alla pena di morte tramite ghigliottina (per qualche strano e sadico motivo).

Durante la permanenza all'interno del penitenziario, il protagonista stringe un'amicizia sempre più forte con il suo compagno di cella: Sabino Ciampa. Costui è un soggetto abbastanza singolare, molto creativo e con una leggerissima dipendenza da droghe leggere. La loro amicizia diventa quasi un legame fraterno, e Sabino sente di potersi fidare di lui al punto da mostrargli il suo ultimo esperimento.

Il gioco inizia la notte prima del giorno stabilito per l'esecuzione. Ciampa mostra il funzionamento della sua invenzione al protagonista, di cui vestiremo i panni, e i rischi che ne derivano. Utilizzando questo strumento, riusciamo ad uscire dalla cella quasi "attraversando" delle strane allucinazioni. Da questo punto inizia la partita vera e propria.

Dovremo muoverci all'interno del piccolo carcere denominato QGFC, tra mill...cent...diec...qualche insidia e pericolo (oltre a qualche bug e glitch che non guasta mai), fino a raggiungere l'uscita e scappare via. Per riuscire a raggiungere il finale sarà necessario sbloccare aree con delle chiavi, interagire con altri carcerati, barattare oggetti con informazioni e affrontare gli strani viaggi astrali che ci provocherà l'invenzione di Sabino e che saranno fondamentali per progredire nell'avventura.

Un paio di righe per descrivere questa fantomatica invenzione: si tratta di una normale pistola, al cui interno è stato inserito l'ago di una siringa, reso retrattile tramite un meccanismo interno, che ha il compito, premendo il grilletto, di penetrare il cranio dell'utilizzatore ed iniettare una dose della strana droga creata dal tuo coinquilino. La dose non durerà molto, ma ti permetterà di interagire con gli elementi che vedrai negli strani luoghi in cui ti ritroverai una volta usata.

Sviluppo e scelte progettuali

Il progetto è diviso in 5 package, contenenti ognuno delle classi che, messe insieme, formano il gioco in sé.

Il **mainPackage**, quello principale, contiene una Main Class che ci permette di inizializzare ed avviare il gioco (la classe *Starter*) e una Abstract Class (la classe *GameDescription*), che contiene gli attributi e i metodi generali che un gioco 'tipo' dovrebbe contenere. Nella classe *Starter*, verrà preso da tastiera il comando digitato dall'utente, e poi passato il contenuto al *Parser* che, una volta riconosciute le componenti nel comando, passerà la classificazione alla funzione *nextMove()* della classe *GameDescription*.

Il **mainPackage.game** contiene il gioco vero e proprio, la classe *NASS*. Essa eredita da *GameDescription* e la estende aggiungendo attributi e metodi specifici del gioco stesso.

Il **mainPackage.parser** contiene due classi: il *Parser* e il *ParserFilter*.

La classe *Parser*, grazie al metodo parse() e ai metodi privati che riconoscono gli elementi inseriti dall'utente in riga di comando, analizza i comandi inseriti dall'utente stesso, e classifica i vari componenti ritrovati attraverso la classe *ParserFilter*, che conterrà gli attributi che permettono di "conservare" i token riconosciuti dal *Parser*.

I comandi accettati dal sistema dovranno essere coniugati alla seconda persona singolare, e il *Parser* riconosce le seguenti combinazioni di comandi:

<comando>
<comando> <NPC>
<comando> <NPC> <oggettoInventario>
<comando> <oggetto>
<comando> <oggettoInventario>
<comando> <oggettoInventario> <NPC>
<comando> <oggettoInventario> <oggetto>

Per generare dei messaggi di errore abbastanza specifici, il *ParserFilter* è corredato di un attributo che conserva eventuali token extra, che non corrispondono ad oggetti del gioco e sono, a tutti gli effetti, "stringhe inutili" digitate dall'utente.

Il **mainPackage.type** contiene tutte le classi che rappresentano i tipi utilizzati all'interno del gioco:

- la classe *Command* e l'enumerativo *TypeCommand*: gestiscono la costruzione dei singoli comandi, che saranno caratterizzati da un tipo e da un set di sinonimi;
- le classi *GameObject* e *ContainerObject* (eredita dalla prima): descrivono gli oggetti fisici che compongono il gioco, e sono costituite da tanti attributi che contengono le informazioni interessanti per ogni oggetto, e metodi utili ad accedere a queste informazioni;
- la classe *Inventory*: rappresenta l'inventario del personaggio, ed ha come attributi una lista di oggetti di gioco e un numero definito di slot;
- la classe *NPC* descrive i personaggi secondari con cui il protagonista potrà interagire. Ogni NPC è caratterizzato da una sua mappa dialoghi, la quale conterrà il risultato delle varie interazioni che il protagonista avrà con essi;
- la classe *Room*: descrive le stanze giocabili, e tramite attributi interni che indicano i 'confini', è possibile costruire una sorta di mappa di gioco, nella quale il personaggio potrà muoversi. Ogni stanza è vista come un "contenitore" di oggetti e di NPC;
- la classe *DoseGun*: specifica di *NASS*, è la classe che rappresenta l'arma utilizzata nel gioco. Essa è formata da due interi, che indicano il numero massimo di munizioni e il numero di colpi ancora disponibili, ed è caratterizzata da metodi specifici che ne permettono l'utilizzo.

Il **mainPackage.utilities** contiene delle classi di utilità, e cioè *NassDB*, che contiene tutti i metodi utili all'istanziazione degli elementi di gioco, e *UWManager*, con la quale il Parser gestisce la rimozione delle cosiddette 'stopwords' dai comandi inseriti dall'utente.

Costruzione del gioco

L'istanziazione di tutti gli oggetti di gioco avviene tramite caricamento dal **database H2**, le cui credenziali di accesso sono:

username: Sabino password: Ciampa

La classe che implementa tutti i metodi utili al reperimento dei dati è *NassDB*, che contiene tutte le stringhe di selezione dalle table, la stringa di connessione, e un attributo di tipo Connection, che caratterizza la connessione stessa al DB.

Per connettersi al DB abbiamo utilizzato dei *PreparedStatement* su stringhe specifiche per ogni table, catturando i risultati in dei *ResultSet* e popolando man mano le strutture atte a contenere gli elementi di gioco (soprattutto Liste).

Salvataggio e Caricamento

Il gioco è provvisto di meccanismi per il salvataggio dello stato corrente della partita ed il suo recupero per proseguire nell'avventura in un secondo momento. Il tutto viene gestito attraverso l'utilizzo dei file: la classe principale del gioco implementa l'interfaccia *Serializable*, che permette al compilatore di comprendere che la classe può essere "serializzata", ovvero convertita in una stringa di bit salvabile nella memoria dell'elaboratore. Ovviamente, per fare in modo che tutto venga salvato, anche le classi di cui fa uso la classe serializzabile devono implementare l'interfaccia.

Appena si esegue un'azione all'interno del gioco, una variabile booleana viene impostata a false. Questa variabile è un flag che permette al compilatore di comprendere se sono presenti delle modifiche che devono essere salvate.

Il comando "salva" inserito dall'utente permette di richiamare l'operazione **SAVE**, che si occupa di scrivere su file lo stato corrente della partita. Esso, dopo aver terminato il salvataggio, imposterà nuovamente il flag a true, così da comunicare che non vi sono modifiche non salvate.

Ma perché ci serve questo flag?

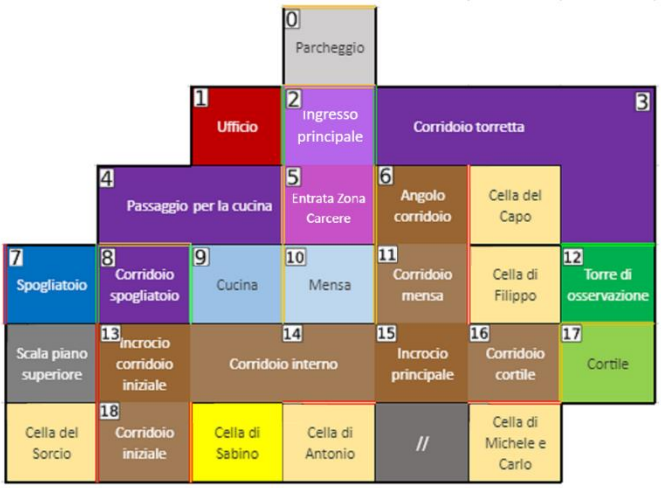
Per l'altro comando, o meglio gli altri due: "carica" ed "esci".

Infatti, quando l'utente digiterà uno di questi due comandi, il compilatore verificherà lo stato del flag: nel caso sia true, verrà immediatamente effettuato il comando, rispettivamente, **LOAD** o **EXIT**, altrimenti sarà stampato un messaggio in cui si avvisa l'utente che sono presenti modifiche non salvate e si domanderà se si vuole comunque procedere. Il giocatore, in questo modo, non rischierà di perdere tutti i progressi se vorrà uscire dal gioco o, nel caso di caricamento, verrà a conoscenza del fatto che ha eseguito azioni che lo hanno portato in uno stato differente rispetto all'ultima volta che ha caricato (oppure rispetto all'inizio della partita, se non ha caricato nulla).

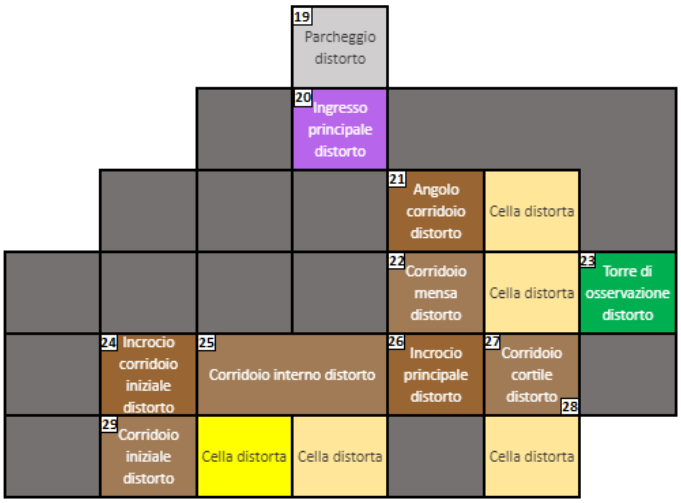
Il caricamento attiverà l'operazione **LOAD**, che convertirà la stringa di bit memorizzati nel file e la deserializzerà, riportando lo stato del gioco a quello memorizzato con il comando "salva"

Mappa di Gioco

La mappa principale è composta da 19 aree accessibili e giocabili, ognuna contenente degli oggetti utili a proseguire il gioco. I vari colori differenziano le varie zone e, nello specifico, le zone in **viola** rappresentano zone accessibili principalmente dalle guardie, e quindi saranno accessibili in tutta sicurezza una volta svolte alcune azioni per neutralizzarle; le zone in **marrone** rappresentano le zone tranquillamente accessibili dai detenuti.



Per rendere il tutto abbastanza veritiero, le **celle** dei detenuti non sono accessibili, ma il giocatore potrà interagire con loro stando nella stanza adiacente alla cella.

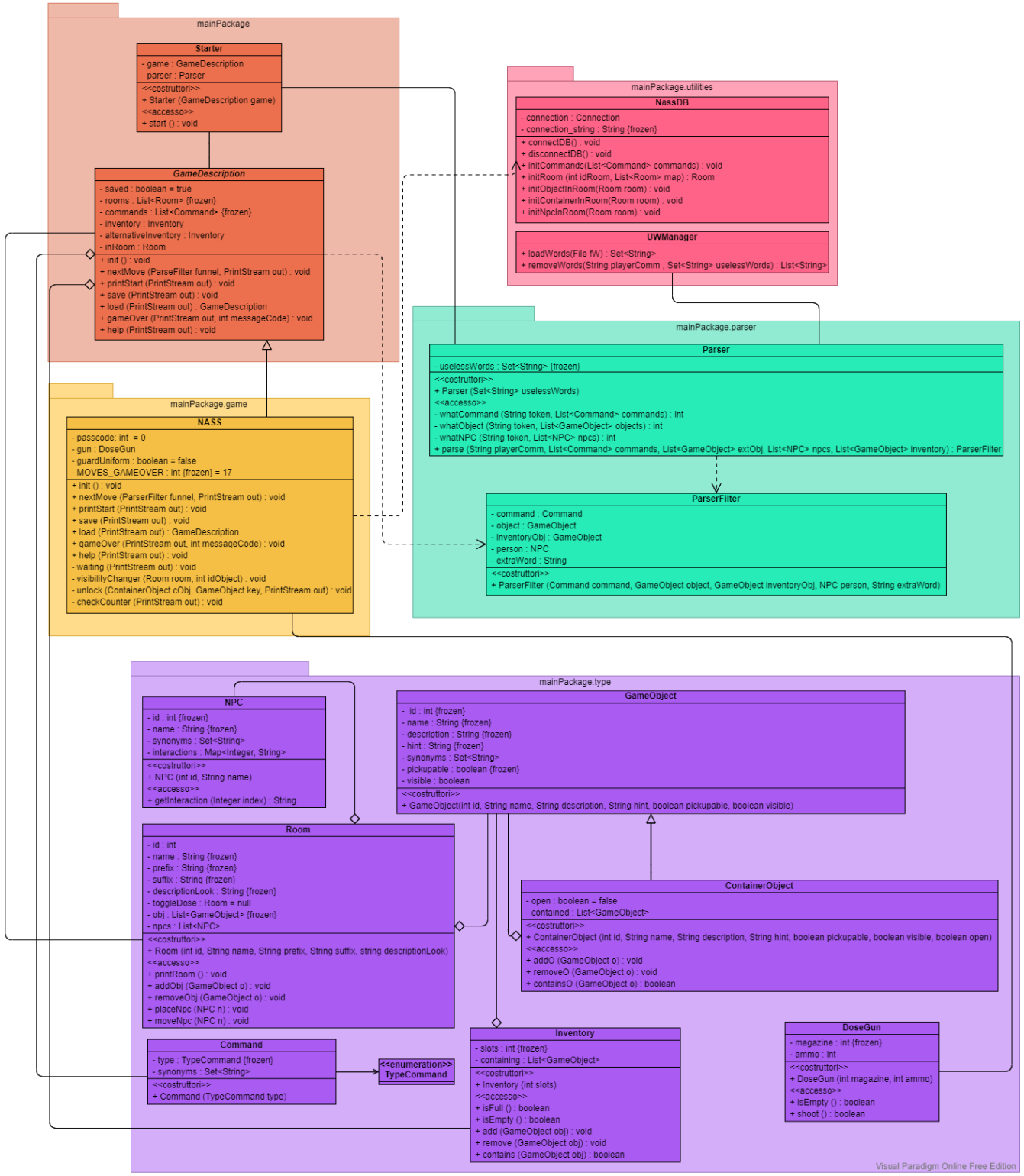


Inoltre, ci sono delle zone alternative accessibili solo tramite l'utilizzo delle **dosi** presenti nel gioco, che permetteranno di sbloccare altri oggetti e completare degli enigmi. Queste zone sono viste come una sovrapposizione delle zone esistenti, come se fosse un secondo o terzo piano di un edificio, e le dosi sono l'ascensore.

La permanenza delle zone alternative dipende dalla singola interazione possibile al loro interno, che scatenerà degli eventi positivi o negativi per il protagonista, facendolo catapultare in stanze nelle quali la logica non fa da padrona.

I *numeri* in figura rappresentano gli ID delle stanze. Le *doppie linee rosse* indicano che le porte che conducono ad una determinata zona sono inaccessibili, e nulla potrà sbloccarle (ad esempio, le celle). Le *doppie linee gialle* indicano che le porte che conducono ad una determinata zona sono temporaneamente chiuse, a meno che non si trovi qualcosa che sblocchi l'accesso (ad esempio, una chiave). Le *doppie linee verdi* indicano che le porte che conducono ad una determinata zona sono aperte, e quindi sempre accessibili. Le *linee nere* indicano la presenza di un muro. Se non ci sono delimitatori, semplicemente le zone sono adiacenti tra loro, senza porte che le separano.

Diagramma delle Classi



Il diagramma delle classi qui sopra comprende gli aspetti più “interessanti” del progetto. Sono state inserite tutte le classi, ma il loro contenuto è stato "sfoltito", mantenendo metodi e attributi di maggior rilievo.

Tutte le classi rappresentate sono state inserite nei loro appositi package (quelli che sono raffigurati con un colore più tenue rispetto alle classi contenute in essi).

Iniziando dall'alto troviamo il **mainPackage**, al cui interno troviamo una prima classe denominata *Starter*. Questa è la classe contenente il metodo *main()* da cui parte l'intero programma, ed ha due attributi che associano questa classe ad altre due, ovvero:

1. *GameDescription*, contenuta all'interno dello stesso package di *Starter*. È una classe astratta, nel cui interno si possono trovare gli strumenti generici che verranno ereditati dai vari giochi creati. Vi sono anche una serie di metodi astratti che rappresentano le funzioni base del gioco (alcuni esempi sono *nextMove()*, *save()*, *load()* ecc..);
2. *Parser*, la classe che si occupa di riconoscere i comandi inseriti in input dall'utente. Al suo interno, oltre ai vari metodi per riconoscere gli elementi inseriti nella frase, vi è anche il metodo *parse()*, che restituisce un elemento di tipo *ParserFilter* nel quale sono stati suddivisi e catalogati i token rilevanti per l'esecuzione di un determinato comando.

Spostandoci dal **mainPackage** al **mainPackage.parser**, troviamo la sopra citata classe *Parser* e la classe *ParserFilter*, legata alla prima tramite un'associazione di dipendenza (di queste parleremo più avanti).

Completando i collegamenti tra i due package nominati sopra, esiste un'associazione di dipendenza tra *GameDescription* e *ParserFilter*.

Ci spostiamo ora nel package **mainPackage.game**, dove troviamo un'unica classe, che costituisce il nucleo centrale dell'avventura testuale: *NASS*. Questa classe eredita da *GameDescription* ed è collegata tramite un'associazione di dipendenza ad una classe del prossimo package.

mainPackage.utilities contiene due classi: *NassDB*, ovvero la classe utilizzata per caricare i dati dal database (e che, inoltre, è in relazione con la classe NASS attraverso l'associazione di dipendenza descritta precedentemente) e *UWManager*, ovvero la classe che gestisce le useless words, cioè le parole inutili ai fini del riconoscimento dei comandi inseriti dall'utente (gli articoli, le preposizioni ecc.).

Prima di spostarci nell'ultimo package, parliamo delle dipendenze: esse sono delle associazioni che descrivono un legame tra due classi sulla base del quale una classe ha bisogno dell'altra per l'istanziazione e l'implementazione. Possiamo infatti vedere che il metodo *nextMove()* di *GameDescription* non potrebbe funzionare senza l'utilizzo di un'istanza di *ParserFilter*. Stessa cosa si può dire per quanto riguarda *NASS*, che utilizza i metodi statici della classe *NassDB*.

Concludiamo con **mainPackage.type**, in cui sono presenti le classi che istanziano gli elementi principali contenuti all'interno del gioco. Partiamo da *NPC*, contenente i dati dei vari personaggi non giocanti. Oltre a id, nome e sinonimi, ciò che assume una certa rilevanza è l'attributo *interactions*: una mappa contenente i dialoghi ed i tipi di interazioni di ogni personaggio. Questa classe è associata con un legame di tipo aggregazione alla classe successiva: *Rooms*.

La classe *Rooms* è quella che contiene i dati delle stanze che compongono la mappa del gioco, nelle quali saranno presenti oggetti, personaggi e contenitori di altri oggetti. Saranno, inoltre, i luoghi in cui il nostro protagonista potrà spostarsi. Questa classe è associata direttamente alla classe *GameDescription*.

Passiamo ora a *Inventory*: la classe che rappresenta l'inventario del protagonista. Come la *Room*, anche questa è un contenitore, ma il suo contenuto è formato da soli oggetti (non può contenere NPC).

ContainerObject è la terza ed ultima classe che contiene altri elementi, anche qui limitati al solo tipo di oggetto.

Gli oggetti che possono essere contenuti nelle tre classi citate sopra sono raccolti sotto il nome di *GameObject*. Oltre ad essere la superclasse della classe *ContainerObject*, essa possiede tre associazioni di tipo aggregazione, rispettivamente, con *Room*, *Inventory* e la stessa *ContainerObject*.

Command è il nome della classe contenente i vari comandi che possono essere utilizzati dal giocatore all'interno del gioco, tutti raccolti in categorie elencate all'interno dell'enumerativo *TypeCommand*.

Sia *Command* che *Inventory* hanno un'associazione di tipo aggregazione con *GameDescription*.

Ultima classe: *DoseGun*. Questa classe rappresenta l'arma utilizzata all'interno del gioco, con metodi che ne descrivono il funzionamento e attributi che immagazzinano il numero rimasto di utilizzi e il numero massimo di questi ultimi.

Specifica Algebrica Struttura Dati Utilizzata

Nel nostro progetto, abbiamo utilizzato molto la struttura dati **Lista**, una struttura dati omogenea che rappresenta una sequenza finita di elementi dello stesso tipo.

La Lista è, inoltre, una struttura dati dinamica, può crescere e decrescere nel tempo, e gli elementi al suo interno possono comparire più volte in posizioni diverse.

Gli elementi della lista sono detti *nodi* o *atomi*, e ad ogni elemento *i* è associata una posizione *pos(i)* (che non è detto che corrisponda all'indice della lista) e un valore *a(i)*. Il numero degli elementi che compongono la lista denota la sua lunghezza, e se la lista non ha elementi allora è detta vuota.

Indichiamo la Lista con la notazione $L = <a_1, a_2, \dots, a_n>$, con $n \geq 0$.

Della Lista si riporta, in seguito, la specifica algebrica, della quale distinguiamo la specifica sintattica, la specifica semantica e la specifica di restrizione.

Specifica Sintattica

Sorts: list, item, position, boolean

Operations: newList() -> list //Crea una lista vuota

isnewList(list) -> boolean //Verifica se una lista è vuota

addList(list, item, position) -> list //Aggiunge un elemento alla lista, nella posizione data

removeList(list, position) -> list //Rimuove un elemento dalla lista, nella posizione data

writeList(list, item, position) -> list //Scrive un elemento nella lista, nella posizione data

readList(list, position) -> item //Legge un elemento dalla lista, nella posizione data

Tabella Costruttori/Osservazioni

Osservazioni	Costruttori di L'	
	newList()	addList(L,i,p)
isnewList(L')	True	False
removeList(L',p')	Error	If p=p' then L else addList(removeList(L,p'), i, p)
writeList(L',i',p')	Error	addList(removeList(L,p'), i', p')
readList(L',p')	Error	If p=p' then i else readList(L,p')

Specifica Semantica e di Restrizione

Declare: L : list, i,i' : item, p,p' : position

- isnewList(newList()) = true
- isnewList(addList(L, i, p) = false
- removeList(newList(), p) = error

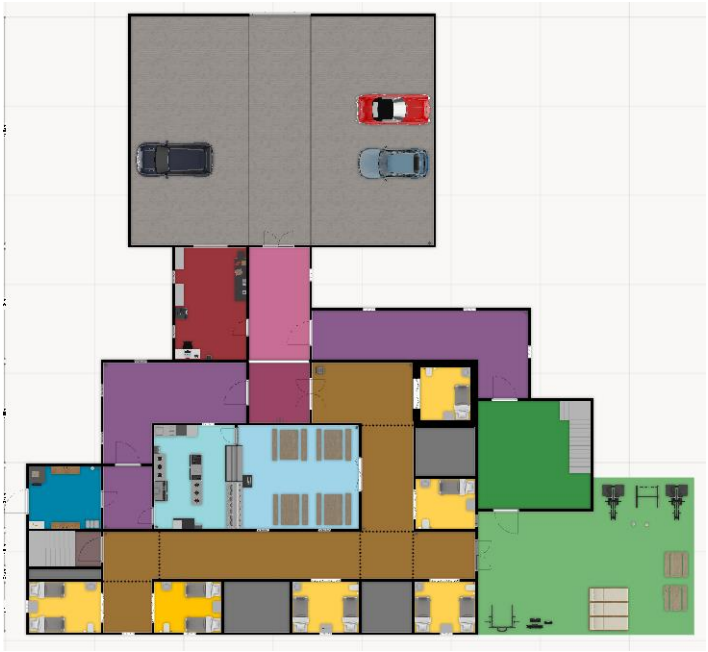
- removeList(addList(L, i, p), p') = if p=p' then L else addList(removeList(L, p'), i, p)
- writeList(newList(),i,p) = error
- writeList(addList(L,i,p), i', p') = addList(removeList(L,p'), i', p')
- readList(newList(),p) = error
- readList(addList(L,i,p), p') = if p=p' then i else readList(L,p')

Organizzazione

Per questioni di comodità a causa della distanza tra le sedi dei componenti del gruppo, si è rivelato fondamentale l'utilizzo di varie piattaforme che permettessero il lavoro condiviso.

Nello specifico, si è fatto utilizzo di: *Microsoft Teams* per mettersi in contatto rapido e discutere del progetto; *Notion* come software di appoggio per "dare forma" alle idee e schematizzarle al meglio; *Visual Paradigm* come software di stesura del diagramma delle classi; *GitHub* come ambiente di lavoro condiviso che, collegato a *NetBeans*, ha permesso un accesso rapido alle varie modifiche apportate al codice senza scambio continuo di file tra un componente e l'altro.

Altro tool online utilizzato è *FloorPlanner*, che ci ha permesso di modellare un'idea più concreta di ciò che stavamo andando a realizzare, aiutandoci a generare descrizioni dei luoghi quanto più accurate possibili. In seguito, un esempio della struttura dall'alto (purtroppo incompleto).



Ulteriori Dettagli

Come avviare il gioco

Per avviare *Not A Sabino's Saga*, basterà scaricare il progetto Maven dal link al repository GitHub, aprirlo con un IDE adatto e far partire la classe main *Starter.java*. In alternativa, si può avviare il file *NotASabinosSaga-0.1-jar-with-dependencies.jar* da linea di comando.

Soluzione del gioco

La soluzione è stata scritta utilizzando un colore molto chiaro, in modo da non essere letta per errore durante la visualizzazione di questo documento. Per vederla, si può selezionare il testo con il cursore ed evidenziarlo.

INTERAGISCI UGO – NORD – DOSATI – GUARDA – INTERAGISCI MURO – EST – GUARDA – PRENDI STAGNOLA – OVEST – OVEST – USA STAGNOLA SU TELECAMERA – OVEST – GUARDA – APRI TASCA – PRENDI OROLOGIO – EST - SUD – SUD – DAI OROLOGIO A UGO – NORD – NORD – OVEST – INTERAGISCI ARMADIO – PRENDI CHIAVE2 – EST – EST – USA CHIAVE2 – EST – USA CHIAVE2 – EST – NORD – GUARDA – INTERAGISCI CAPO – SUD – GUARDA – DOSATI – GUARDA - INTERAGISCI FILIPPO – SUD – EST – GUARDA – DOSATI – USA PORTICINA SU UKS – EST – GUARDA – PRENDI PESETTO – USA PESETTO SU CASTORPIO – NORD – GUARDA – PRENDI FASCETTE – USA FASCETTE SU CASTORPIO – DOSATI – GUARDA – PRENDI BAMBOLA – INTERAGISCI LANTERNA – NORD - OVEST – GUARDA – INTERAGISCI BARBUINO – USA FASCETTE SU BARBONI – OVEST – GUARDA – PRENDI CHIAVI – INTERAGISCI CALENDARIO – PRENDI CHIAVE1 – APRI CASSETTO – PRENDI CELLULARE – EST – SUD – USA CHIAVI – OVEST – USA CHIAVI – SUD – SUD – SUD – DAI CELLULARE A UGO – NORD – NORD – EST – EST – GUARDA – INTERAGISCI TAVOLI – PRENDI CHIAVE3 – EST – NORD – OVEST – NORD – OVEST – APRI CASSAFORTE – PRENDI ANELLO – EST – SUD – OVEST – SUD – SUD – SUD DAI ANELLO A UGO – INTERAGISCI SABINO – DOSATI – INTERAGISCI SABINO – NORD – EST – DOSATI – GUARDA – INTERAGISCI OCCHI – EST – EST – DOSATI – GUARDA – DAI BAMBOLA A BAMBINE – OVEST – NORD – INTERAGISCI FILIPPO – NORD – DOSATI – GUARDA – INTERAGISCI DVCE – OVEST – NORD – USA CHIAVI – NORD – ITERAGISCI PAD.