Documentazione progetto Metodi Avanzati di Programmazione 22-23

# **M0U53**

Corso in:

Metodi Avanzati di programmazione (Corso B)

Stefano Todaro-matricola:720335

Sommario

[**M0U53** 1](#_Toc144370909)

[Descrizione avventura testuale 2](#_Toc144370910)

[Idee progettuali 2](#_Toc144370911)

[Lato Client 2](#_Toc144370912)

[Lato Server 3](#_Toc144370913)

[Package Elementi 3](#_Toc144370914)

[Definizione degli elementi di gioco 3](#_Toc144370915)

[Meccaniche di gioco 5](#_Toc144370916)

[Parserizzazione dei comandi 6](#_Toc144370917)

[Meccaniche di gioco (seconda parte) 7](#_Toc144370918)

[Diagramma delle classi (GameSettings) 9](#_Toc144370919)

[9](#_Toc144370920)

[Specifica algebrica (Map) 10](#_Toc144370921)

[Specifica sintattica 10](#_Toc144370922)

[Specifica semantica 10](#_Toc144370923)

[Specifica di restrizione 11](#_Toc144370924)

# Descrizione avventura testuale

Il gameplay e le meccaniche del gioco prendono spunto principalmente da giochi classici del genere, come ad esempio “Zork”, mentre il tono dell’avventura e la caratterizzazione dei personaggi segue la linea del ben più famoso “Monkey Island”. La trama dell’avventura testuale, invece, è largamente ispirata al programma TV del 1995 prodotta da Warner Bros “Mignolo col Prof”. L’avventura, infatti, vede come protagonista un topo da laboratorio, Mickola, che in seguito alla sperimentazione di un farmaco (l’MCP95) acquisisce un intelletto perfino superiore rispetto a quello degli stessi scienziati che lo studiano. Mickola, conscio di come il suo futuro verrà interrotto da una vivisezione, decide di fuggire la notte prima dell’operazione. Una volta raccolto il materiale ed aver studiato i meccanismi del laboratorio, giunge la notte della fuga. Per prima cosa Mickola deve uscire dalla teca in cui è nato e cresciuto insieme alla sua famiglia, ovvero altri topi da laboratorio su cui è stato testato il farmaco che però non ha avuto lo stesso successo che ha avuto con lui. Una volta ottenuto lo zaino dalla mamma, così da poter raccogliere gli oggetti che troverà utili, Mickola sfrutterà i materiali disseminati per la teca e le caratteristiche dei suoi fratelli (Gouda, Mortimer e Speedy) per costruire una via di uscita dalla teca. Una volta fuori Mickola dovrà riuscire ad affrontare il malvagio inserviente, un uomo con una malsana passione per la caccia e la disinfestazione. Per sconfiggerlo Mickola dovrà risolvere dei test intellettivi eseguiti dalle cavie per la sperimentazione del farmaco e dovrà affrontare creature mostruose nate da strani esperimenti, così da poter ottenere le chiavi dell’unica uscita del laboratorio e poter portare fuori con sé la sua famiglia e tutte le cavie del laboratorio.

# Idee progettuali

Il programma presenta tre strutture package principali, ovvero: **Client**, **Server** e il package contente il corpo vero e proprio del gioco, ovvero il package **Elementi**. Il gioco presenta un’architettura client-server, in modo che queste due componenti possano eseguire funzioni complementari, con il client concentrato nella visualizzazione dell’avventura e della GUI, mentre il server si occupa della gestione delle interazioni dell’utente.

## Lato Client

Il lato client, una volta avvenuta la connessione con il lato server con successo, richiama metodi appartenenti al framework **Java Swing** per l’inizializzazione dell’interfaccia utente. Il frame presenta tre sezioni principali: Il JLabel, che durante il gioco riporta il nome della stanza attuale, mentre nella schermata iniziale presenta il titolo dell’avventura; la JtextArea, il corpo principale del frame, dove verranno visualizzate dal giocatore tutti gli avvenimenti dell’avventura e otterrà il responso ad ogni azione eseguita. Inoltre, al giocatore verrà permesso di visualizzare tutti gli avvenimenti avvenuti durante la partita, in quanto la schermata presenta un pannello a scorrimento verticale (JScroolPane) che presenta anche i messaggi inseriti dal giocatore; il JtextField, ovvero la sezione in cui il giocatore inserisce i propri input.

Alla fine viene determinato il comportamento del frame alla chiusura. Questo richiederà conferma al giocatore sulla scelta di terminare la sessione di gioco. In caso di risposta positiva verrà mandato un messaggio al server per eseguire una corretta chiusura.Inizializzata la GUI viene inizializzato il thread utile per ottenere l’input dall’utente.

Il *thread* utilizza il proprio metodo *run* per rimanere in attesa sfruttando il metodo *sleep*, ad intervalli regolari.In tal modo il processore non viene sovraccaricato dal richiamo costante del metodo . Quando l’utente preme il tasto invio l’azione del thread viene interrotta, cambiando il valore di un attributo booleano di classe, e il flusso dei processi ritorna nell’esecuzione del client che , per prima cosa stampa il comando nella JTextArea per poi inviarlo al server dove verrà eseguita la parserizzazione del messaggio, rendendolo formattato in modo utile per il sistema. Infine il messaggio viene cancellato dal JTextField,permettendo l’inserimento di un nuovo input. Prima della stampa a schermo dell’input viene richiamato il metodo *appendText*, il quale principalmente si occupa della stampa, ma è possibile che un dato input dell’utente possa dar inizio allo scambio di messaggi specifici tra client e server utili per il compimento di alcune meccaniche. Ad esempio, ogni volta che il giocatore si sposterà da una stanza all’altra al client giungerà il messaggio *“room”*, il che farà in modo che il client richiami il metodo *updateGUI*, che si occuperà della modifica della JLabel, che assumerà il nome della nuova stanza in cui si trova l’eroe dell’avventura. Altri messaggi speciali sono: *stop, fine sessione, salva, game over, start e menu.* La connessione con il server verrà interrotta una volta che l’utente inserisca uno specifico comando oppure una volta chiuso il frame.

## Lato Server

Il lato server si occupa principalmente della ricezione degli input dell’utente e nella traduzione di questi in comandi di gioco. La traduzione dei messaggi verrà eseguita tramite parserizzazione, ma tale procedimento verrà trattato più dettagliatamente in seguito.

Il server una volta inizializzato rimane in attesa di una connessione client. Una volta avvenuta tale connessione vengono inizializzati gli stati principali che il gioco può raggiungere e il parser. Gli stati sono tre e sono rappresentati da attributi booleano, che in modo alternato assumono valore vero. Questi son: *game, go (game over)* e *title.* Quindi avvenuta la connessione con il client il server rimane in attesa dell’input. Una volta giunto il metodo *run* viene richiamato. Da qui il controllo del sistema viene passato alle classi appartenenti al package **Elementi**. Prima di giungere al *run* il server controlla, come il client, che il messaggio giunto non sia un comando speciale per l’avvio di meccaniche particolari.

Eseguito il controllo sui messaggi speciali si giunge al richiamo effettivo del metodo *run*, il quale verifica lo stato attuale del gioco, controllando quali delle tre variabili di stato sia *true*. All’inizio di ogni partita il gioco avrà la variabile *title* settata a true, il che porterà il giocatore alla visualizzazione della schermata principale di gioco (anche questo aspetto verrà approfondito in seguito). Una volta che l’utente avvia la partita il sistema ricerca il file *“M0U53.sav”,* ovvero il file in cui vengono salvati i dati di gioco. Se tale file non viene ritrovato allora viene richiamato il metodo *init (*anche questo metodo verrà approfondito nella sezione dedicata al package *Elementi*) che si occupa dell’inizializzazione di tutti gli elementi che andranno a comporre il gioco effettivo. Invece, se viene trovato il file viene richiamato il metodo *loadGame*. Il metodo crea un *FileInputStream* e un *ObjectInputStream* e tramite il metodo di lettura di file *readObject* avvalora la variabile game con i dati conservati all’interno del file, in modo da far riprendere il gioco dall’ultimo salvataggio effettuato. La classe *GameSettings* di cui *game* è oggetto, come tutte le classi utili, sono serializzate, in tale modo ne viene garantita la persistenza e reso possibile il salvataggio dei dati su file. Sotto determinate condizioni la variabile *go* assume valore vero, facendo sì che il metodo *run* porti il giocatore alla visualizzazione della schermate di game over( ma anche questo aspetto verrà approfondito in una sezione dedicata).

## Package Elementi

Per prima cosa verranno introdotti e spiegati quelli che sono gli elementi principali dell’avventura.

### Definizione degli elementi di gioco

La classe *CommandType* è una classe enumerativa. Questa classe contiene (potremmo dire) dei valori label. Ognuno di questi valori rappresenta una tipologia di comando utilizzabile all’interno del gioco, in questo modo i comandi, che verranno in seguito definiti e inizializzati in altre classi, potranno cadere unicamente in una di queste specifiche label, e tramite essa verrà riconosciuta l’azione del comando. Queste aggregazioni di comandi in varie tipologie è stata selezionata in quanto ogni comando è identificato non solo da un nome principale, ma anche da una sequenza di alias riconducibili sempre al medesimo tipo di comando. In tal modo il giocatore si sentirà più libero di potersi esprimere a proprio modo, rendendo l’avventura più piacevole e personale, andando inoltre a rispettare gli stilemi del genere di gioco.

Nella classe *Command* vengono delineate le caratteristiche dei comandi che verranno utilizzati dai giocatori. Gli oggetti di classe *Command* sono identificati da una tipologia (di classe *CommandType*), da un nome principale e da degli alias. Gli alias sono definiti all’interno di un Set di tipo stringa, in tale modo ogni alias che viene fornito al comando deve essere unico e viene identificato da un codice hash.

Un’altra classe fondamentale è la classe *AdvObjct*. Questa classe va a definire quelli che sono gli oggetti effettivi con cui il giocatore andrà ad interagire. Gli oggetti definiti all’interno della classe *AdvObject*  sono caratterizzati da un id che vada a caratterizzare l’oggetto, tale id deve essere unico per un corretto funzionamento delle interazioni ed è definito manualmente dallo sviluppatore, ergo il giusto assegnamento è affidato a lui; un nome e un insieme di alias, questi attributi sono omologhi a quelli dei comandi; in più,però, rispetto ai comandi, ogni oggetto è caratterizzato da una descrizione che potrà essere visualizzata dal giocatore durante la partita. Inoltre ogni oggetto di gioco è caratterizzato da diverse variabili booleane che ne specifica le caratteristiche: ad esempio un oggetto può reso non visibile al giocatore, o ancora, non tutti gli oggetti possono essere raccolti dal giocatore. Dalla classe *AdvObject* derivano due sottoclassi per tipologie di oggetti differenti, ovvero la classe *Character* e la classe *AdvObjectContainer*.

La sottoclasse *AdvObjectContainer* rappresenta gli oggetti contenitori, ovvero degli oggetti che sono in grado di contenerne degli altri. Gli oggetti di tale classe godono delle stesse caratteristiche della superclasse, ma hanno in più l’ArrayList *list*, contenente oggetti di tipo *AdvObject.*

La sottoclasse *Character* oltre agli attributi ereditati da *AdvObject*, possiedetre attributi aggiuntivi. Il primo attributo *dialogue*. Come già è intuibile dal nome questa collection contiene i dialoghi del personaggio. In questo caso l’interfaccia è realizzata tramite un’HashMap, in tal modo ogni dialogo è identificato da un codice hash univoco. Il secondo attributo è direttamente collegato a *dialogue* e si tratta della variabile *dialogueId.* Questo attributo viene utilizzato per identificare quale linea di dialogo il personaggio fornirà al giocatore una volta interagito con lui.L’ultimo attributo caratteristico della classe è la lista di tipo *AdvObject* *object*, appunto. Questa collection ha un funzionamento analogo alla *list* in *AdvObjectContainer*. Questa viene usata in quanto i personaggi possono possedere oggetti che non devono essere immediatamente accessibili al giocatore.

L’ultima componente degli elementi che vanno a costituire “fisicamente” il gioco è la classe *Room*, la classe fondamentale per far sì che il gioco abbia una struttura vera e proprio e che costituisce lo spazio in cui inserire tutti gli elementi precedentemente descritti e in cui il giocatore si muoverà. Gli elementi di classe *Room* sono identificati da un Integer, *id*. L’identificativo deve essere unico per ogni stanza e il corretto inserimento è lasciato al programmatore. Inoltre, ogni stanza, come accade per gli oggetti, presenta un nome (*name)* e una descrizione (*description)*, che verranno entrambi visualizzati dal giocatore ogni volta che si sposterà da una stanza all’altra. Il nome della stanza sarà visibile in ogni momento alla testa della schermata. Non ogni stanza è immediatamente accessibile o visibile al giocatore, alcune diventano attraversabili solo dopo aver compiuto determinate azioni. Per fare in modo da costruire una vera e propria mappa di gioco è necessario che ogni stanza sia collegata alle altre. Il giocatore si può muovere nella mappa andando a nord, sud, ovest, est, su e giù. Perciò, ciascuna stanza deve indicare la stanza che il giocatore raggiungerà scegliendo di muoversi in una delle direzioni indicate. Ovviamente è contemplato che per date direzioni una stanza non abbia nessun’altra stanza collegata. Infine le stanze contengono due array list distinte, una contente gli oggetti (lista di *AdvObject*) e l’altra contente i personaggi (lista di *Character*), anche se per motivazioni funzionali sarà necessario inserire all’interno delle stanze, oltre al personaggio effettivo, anche un oggetto corrispondente che ne permetta l’interazione, ma tale aspetto verrà approfondito in seguito.

Con quest’ultima classe termina la descrizione delle classi contenti gli elementi di gioco, di seguito verranno presentate le classi che permettono l’effettivo utilizzo delle meccaniche di gioco.

### Meccaniche di gioco

La classe astratta *GameDecription* viene utilizzata per modellare la struttura del gioco per le varie sue sezioni. *GameDescription* contiene tre array list*.* rooms è una array list di stanze. Questa non contiene tutte le stanze, ma bensì unicamente quelle che hanno ruoli particolari nell’avventura e che, per esempio, subiscono modifiche dei propri attributi. commands contiene la lista dei comandi utilizzabili dal giocatore. inventory è una lista che contiene gli oggetti raccolti dal giocatore, questi oggetti, che vengono trasportati dal giocatore e possono essere utilizzati per scatenare degli eventi, almeno se utilizzati in giusta combinazione.Infine *GameDescription presenta l’attributo currentRoom*, di fondamentale importanza, che rappresenta la stanza in cui si trova attualmente il giocatore. Oltre a questi attributi, la classe contiene anche la definizione di metodi che verranno poi definite nelle classes derivanti: il metodo *init* e il metodo *nextMove*

La classe *GameSettings*, derivante da GameDescription, ne implementa ogni attributo e i metodi, dando una implementazione specifica. *GameSettings*  presenta un attributo particolare di tipo *Conditions* con.

La classe *Conditions* è una classe contenente unicamente delle variabili di tipo booleano. Queste variabili servono a tenere traccia del progresso del giocatore lungo l’avventura. In base al cambiamento delle variabili al giocatore sarà permesso eseguire delle nuove azioni.

Torniamo a discutere della classe *GameSettings*. La classe deve obbligatoriamente implementare i metodi della super classe astratta. Il primo è il metodo *init*. Questo metodo è utilizzato per l’inizializzazione e definizione degli elementi di gioco. La prima porzione del metodo è dedicata alla definizione dei comandi: vengono, per prima cosa, definiti i nomi dei comandi e viene associato il tipo di comando (*CommandType*); in seguito vengono definiti tutti gli alias, ovvero tutti i modi in cui il giocatore si può riferire al comando. Infine, il comando viene aggiunto alla lista dei comandi. In seguito vi è la inizializzazione delle stanze: alle stanze viene associato immediatamente l’identificativo numerico, il nome e la descrizione. Alla definizione delle stanze vengono indicate anche le stanze raggiungibili nelle varie direzioni, ma questo non è sempre possibile, in quanto stanze raggiungibili da altre verranno definite solo in seguito, per questo la definizione delle stanze viene ultimata dopo aver definito le stanze raggiungibili nelle varie direzioni dovuti. Al termine della definizione delle stanze vengono aggiunte le stanze utili alla lista *rooms*. Definite le stanze, ora vengono definiti gli oggetti: come per le stanze vengono definiti principalmente l’identificativo numerico, il nome e la descrizione. Vari oggetti hanno bisogno di specifiche in più, ad esempio è necessario specificare quali oggetti non è possibile raccogliere. Per gli oggetti contenitore (*AdvObjectContainer*) bisogna specificare se sono aperti o meno, e soprattutto va specificato se sono apribili , mentre bisogna specificare che gli oggetti contenuti al loro interno non sono visibili immediatamente, almeno se il contenitore è chiuso. Si giunge alla definizione dei personaggi. Questi sono definiti due volte: una volta definiti di tipo *AdvObject* e un’altra come oggetti di tipo *Character.* Questa scelta è stata effettuata in quanto al giocatore è unicamente possibile interagire con gli oggetti all’interno della stanza, ma non con altre tipologie di elementi, come i personaggi. Stessa scelta è effettuata anche per gli oggetti contenuti da oggetti contenitori, in quanto se questi fossero contenuti unicamente all’interno del contenitore non esisterebbero all’interno della stanza, rendendone l’interazione impossibile. Quindi sarà presente un identificativo sia per il personaggio di classe *AdvObject* e per il personaggio di classe *Character*, come anche il nome. La descrizione invece sarà presente solo al personaggio *AdvObject*, in quanto, come già spiegato, il giocatore potrà interagire solo con l’oggetto. Di seguito viene riempita la lista dei dialoghi del personaggio: quindi viene fornito l’id alla linea di dialogo e, ovviamente, viene inizializzata la stringa. Ci sono casi in cui vi siano personaggi definiti più volte, in quanto rappresentano lo stesso personaggio in differenti stadi dell’avventura. In questo modo viene reso più semplice lo spostamento di un personaggio da una stanza all’altra, senza la necessità di doverlo effettivamente spostare, ma semplicemente rendendo non più visibile il personaggio nello stato precedente e facendo l’opposto al personaggio nello stato successivo all’interno di un’altra stanza. Nella sezione del metodo tutti gli oggetti e i personaggi vengono inseriti all’interno delle stanze apposite, ma vi è da precisare come gli oggetti contenuti all’interno dei contenitori vengono aggiunti sia all’interno della stanza che all’interno del contenitore. Il secondo metodo ereditato all’interno della classe è il metodo *nextMove*, che prende in input un parametro di tipo ParserOutput e con il secondo parametro indica l’output stream. Per comprendere al meglio il funzionamento del metodo verranno spiegate di seguito le classi *Parser* e *ParserOutput.*

### Parserizzazione dei comandi

Il metodo principale della classe è il metodo parse. Per prima cosa si inizializza a zero l’indice di scorrimento della stringa, in modo che ogni stringa venga controllata dall’inizio. Il passaggio successivo consiste nel trasformare la strina in lower case, rendendo l’utilizzo dei comandi non-case sensitive e facilitare l’interazione col sistema. Fatto questo la stringa viene divisa in token utilizzando lo *split* in base agli spazi, andando poi a riempire l’array *tokens*. Controllato che *tokens* abbia affettivamente un numero di elementi maggiore di zero, si richiama il metodo *checkForCommand,* che prende in input il primo token e la lista dei comandi. Il metodo cerca una corrispondenza tra il token e uno dei comandi (o uno degli alias dei comandi), e in caso di esito positivo il metodo ritorna il valore corrispondente al tipo di comando (in quanto si ricorda che la classe CommandTyper è una classe enumerativa), altrimenti viene ritornato il valore -1. Di seguito si controlla che il numero di token sia maggiore di uno. Se questo è il caso allora viene richiamato il metodo *saltaConnettivi*. Questo metodo, in combinazione al metodo *scorriStringa* aumenta l’indice di scorrimento della stringa quando trova una corrispondenza tra il token e uno degli elementi di *connettivi*. *connettivi,* come suggerisce il nome, è una lista contenente tutti i connettivi logici che il giocatore potrebbe inserire all’interno del comando, che però non hanno alcuna utilità per la parserizzazione. Il metodo *checkForObject* è molto simile al *checkForCommand*, ma questo effettua il controllo solo sugli oggetti all’interno della stanza corrente, confrontando il token con il nome o gli alias. I valori ritornati sono gli id degli oggetti o il -1 per il responso negativo. Nel caso in cui sia stata trovata la corrispondenza fra il token e un’oggetto, si effettua un controllo sugli oggetti all’interno del l’inventario tramite il *checkForObject,* che in questo caso prenderà in input non la lista degli oggetti della stanza, ma bensì l’inventario. All’interno di *parse* vengono effettuati dei controlli specifici per le tipologie di comandi, in quanto, sapendo qual è il comando che il giocatore vuole effettuare è più semplice intuire la formattazione della frase, rendendo più semplice la ricerca degli elementi utili alla parserizzazione. Un esempio è il comando di tipo *DROP* (“lascia”) che permette di lasciare oggetti dall’inventario. La formulazione del comando tipo è “lascia \*oggetto dell’inventario\*”. Proseguendo, si effettuano altri controlli sul comando. Se il comando è di tipo GIVE(“dai”) viene ricercato prima l’elemento dell’inventario e poi il personaggio che riceverà l’oggetto (come detto precedentemente, non sarà cercato il personaggio di tipo *Character,* ma la sua controparte *AdvObject*). Successivamente si effettua il controllo per il comando di tipo USE(“usa”). Questo comando può avere varie forme( “utilizza \*oggetto dell’invetario\* su \*oggetto nella stanza\*”, oppure “utilizza \*oggetto nella stanza\*” o ancora “usa su \*oggetto nella stanza\* \*oggetto dell’inventario\*”) dunque per controllare quale di queste formulazioni è stata utilizzata dal giocatore si effettua un primo controllo per un’oggetto dell’inventario, in caso di responso negativo si ricerca l’oggetto all’interno della stanza*.* Stessa cosa avverrà nella seconda porzione del comando inserito, ma verrà effettuato prima il check nella stanza e solo dopo, eventualmente, all’interno dell’invetario. Gli ultimi controlli sul comando vengono effettuati per i comandi ATTACK (“attacca”) e OPEN(“apri”) che hanno la stessa formattazione. Per prima cosa si avvalora *io,* cercando l’elemento all’interno della stanza (“attacca \*oggetto nella stanza\*”). Se i token sono più di due, allora, viene cercato di avvalorare l’attributo *ioinv*(apri \*oggetto nella stanza\* con \*oggetto dell’inventario”). Anche se non è stato esplicitamente detto, fra ogni elemento di rilevanza all’interno del comando e l’altro viene richiamato il metodo *saltaConnettivi.* Alla fine il metodo ritornerà degli elementi di tipo *ParseOutput.* Tali elementi ritornati varieranno in base agli elementi ritrovati e alla lunghezza del comando(ovvero il numero di token).

La classe *ParseOutput* è una classe definita appositamente per fornire una formattazione ai valori restituiti dalla parserizzazione. La classe ha tre attributi: uno di tipo *Command,* e due tipo AdvObject.

Ora che sono state spiegate tutte le classi utili è possibile descrivere il funzionamento del metodo nextMove. Questo prende in input il *ParseOutput* risultato del metodo *parse* e lo stream di output.

### Meccaniche di gioco (seconda parte)

Il metodo *nextMove* effettua principalmente dei controlli sul tipo di comando selezionato dal giocatore, e su questa base effettua azioni o invoca metodi specifici. Se il comando è di tipo COMMAND, allora al giocatore apparirà una legenda contenente tutti i comandi utilizzabili all’interno del gioco, insieme alla formulazione tipo delle frasi per i comandi e qualche consiglio. Se il comando corrisponde un a comando di movimento (NORTH, SOUTH, EAST, WEST, UP, DOWN) allora la stanza corrente verrà impostata in base alla stanza associata alla direzione scelta. Inoltre la variabile *move* sarà impostata a true, il che farà si che vengano fornite le informazioni necessarie al client per modificare il nome della stanza in cima al frame e mostrarne la descrizione. I comandi di tipo INVENTORY mostreranno al giocatore tutti gli oggetti nel suo inventario affiancati dalla loro descrizione*.* Nel caso in cui il giocatore non abbia ancora recuperato lo zaino o nel caso non ci sia nessun oggetto nell’inventario verrà mostrato a schermo un messaggio apposito. Se il comando è di tipo LOOK\_AT(“guarda”) in base alla struttura del comando fornisce varie risposte al giocatore: se il comando specifica un oggetto all’interno della stanza allora ne viene fornita la descrizione, invece se così non è allora al giocatore vengono elencati tutti gli oggetti nella stanza e le stanze raggiungibili nelle varie direzioni. Se l’oggetto specificato è un oggetto contenitore aperto viene richiamato il metodo *containerLook,* che fornisce la lista degli oggetti contenuti al suo interno. Nel caso in cui non ci siano oggetti all’interno della stanza o del contenitore vengono stampati a schermo dei messaggi appositi. Per i comandi di tipo PICK\_UP(“raccogli”) si controlla che l’oggetto sia visibile, che sia raccoglibile e che non corrisponda ad un personaggio, in tali casi verrebbero mostrati vari messaggi. Invece se l’oggetto rispetta tali condizioni viene richiamato il metodo *pickupEvent*, che va a eliminare l’oggetto dalla stanza aggiungendolo all’inventario del giocatore, e occasionalmente per specifici scatena degli eventi specifici. Raccolto l’oggetto si controlla se questo è contenuto all’interno di un contenitore o meno, in caso affermativo l’oggetti viene eliminato dal suo interno e il giocatore “raccoglierà” la copia dell’oggetto all’interno della stanza inizialmente non visibile. I comandi di tipo DROP(“lascia”) fanno lasciare l’oggetto specificato dal giocatore all’interno della stanza attuale, a patto che questo sia contenuto nell’inventario del giocatore. In tal caso l’oggetto verrà eliminato dall’inventario del giocatore e verrà aggiunto alla lista degli oggetti nella stanza. Il comando OPEN(“apri”) effettua vari controlli per determinare l’azione da eseguire: se l’oggetto è apribile allora viene aperto e tutte le copie degli oggetti contenuti al suo interno nella stanza diventano visibili, se l’oggetto non è apribile questo indica che questo deve essere aperto utilizzando un altro oggetto (una chiave ad esempio) (è possibile incontrare oggetti non apribili), controlla che l’oggetto non sia già aperto oppure che l’oggetto indicato non sia il corrispondente di un personaggio. Negli ultimi casi verranno forniti dei messaggi specifici. Per questo comando, come molte sezioni della classe utilizza la lamba expression generalizzata MatchingObject per la ricerca di elementi, prendendo in input uno stream, il predicato di confronto e l’azione da eseguire. I comandi CLOSE (“chiudi”) controllato che l’oggetto contenitore non sia già chiuso ricerca corrispondenza tra gli oggetti contenuti al suo interno e gli oggetti corrispondenti nella stanza e li rende entrambi non più visibili. Per i comandi di tipo TALK\_TO(“parla”) una volta trovata corrispondenza tra il nome dell’oggetto indicato e un personaggio all’interno della stanza si richiama il comando talkEvent. In generale questo comando gestisce la progressione dei dialoghi ogni qualvolta che il giocatore parla con uno dei personaggi, facendo proseguire al dialogo successivao. Può anche accadere che un dialogo possa scatenare degli eventi (come, per esempio, la consegna dello zaino/inventario). È anche possibile che i dialoghi vadano a variare in base alle condizioni attuali del gioco, ovvero in base alle azioni eseguite dal giocatore sino a quel punto o anche in base agli oggetti nel suo inventario. Per i metodi di tipo GIVE(“dai”) viene ricercata una corrispondenza fra il nome dell’oggetto del parser e uno dei personaggi della stanza, che sarà il personaggio ricevente l’oggetto specificato dell’inventario, per poi richiamare il metodo *giveEvent.* Tale metodo fa sì, che se viene fornito al personaggio il giusto oggetto questo venga eliminato dall’inventario (dando l’impressione di dare l’oggetto) scatenando degli eventi utili per la progressione del gioco. Nel caso in cui l’oggetto fornito non sia di gradimento del personaggio verrà visualizzata una linea di dialogo specifica del personaggio. I metodi dai comandi di tipo USE(“usa”) sono i metodi principali per il proseguimento del gioco. Questi vengono chiamati a seconda che venga specificato un secondo oggetto o meno. Nel caso si utilizzi un oggetto dell’inventario su un oggetto nella stanza (quindi vengono indicati due oggetti) verrà richiamato il metodo *objectEvent*. Questo metodo controlla che la combinazione tra l’oggetto dell’inventario e l’oggetto nella stanza (i controlli vengono effettuati sugli id degli oggetti)sia corretta, e in tal caso questo attiverebbe uno specifico evento. E’ possibile che alcuni eventi siano azionabili unicamente se il giocatore ha effettuato determinate azioni precedentemente. Gli avvenimenti possono avere vari effetti, dal modificare le stanze raggiungibili al modificare la posizione del giocatore cambiando la stanza attuale. Mentre se non è specificato il secondo oggetto all’interno del comando, allora verrà richiamato il metodo *interactEvent*. Questo metodo ha un comportamento abbastanza simile a *objectEvent*, ma questo permette di effettuare delle interazioni con i personaggi o comunque permette di interagire con gli oggetti nella stanza senza la necessità di utilizzare degli strumenti. I comandi di tipo ATTACK (“attacca”) richiamano il metodo *attackEvent*, questo solo dopo aver trovato corrispondenza tra l’oggetto e uno dei personaggi nella stanza, in quanto sarà possibile attaccare unicamente i personaggi e non gli oggetti. Un attacco per essere valido deve essere eseguito utilizzando uno strumento dell’inventario, mentre se l’oggetto non è specificato verrà dato al giocatore un messaggio per bloccarne l’azione. Gli ultimi tipi di comandi SAVE e END (“salva” e “fine”) sono comandi che inviano al client dei messaggi specifici, rispettivamente, per il salvataggio della partita e per la sua chiusura, che avviano tali procedure tramite scambi di messaggi tra client e server. Il funzionamento di queste funzioni è stato meglio analizzato nella parte di documentazione dedicata al client e al server. Con questo termina la descrizione della classe di *GameSettings.*

All’inizio e, potenzialmente, alla fine del gioco si presentano altri due stati. Il primo si incontra all’avvio del gioco. In questa situazione al giocatore verrà presentata la schermata principale di gioco, contenuta nella classe *TitleScreen*. *TitleScreen* è una classe derivante da *GameSettings*,,tant’è che ne implementa il metodo *init* per l’inizializzazione dei comandi utilizzabili all’interno della schermata. Sempre all’interno di *init*, inoltre, viene inizializzata un’unica stanza che corrisponderà proprio alla schermata principale. Il nome della stanza è il titolo di gioco e la descrizione sono i comandi disponibili all’interno della schermata. Ovviamente questa sarà impostata come stanza corrente. Il metodo *menuOption* avrà un comportamento analogo al *nextMove* di *GameSettings*. Questo specifica il comportamento per tre tipologie di comando. I tipi di comando COMMAND sono presenti anche qui e si comportano nella stessa maniera che nel *GameSettings.* I tipi di comando STORY(“storia”) invece quando invocati mostrano al giocatore la trama del gioco, in modo tale da aumentare l’immersione. Infine i comandi di tipo START(“inizia”) porranno il valore della variabile faranno entrare il giocatore all’interno dell’avventura.

Nel caso in cui il giocatore non abbia rispettato la condizione di fornire tre pezzi di formaggio al porcellino d’India Albert, giungerà lo stato di game over, che verrà attivato nel lato server. Quando ciò accade il server richiama il metodo *gameOverChoice* all’interno della classe *GameOver*. La classe ha due attributi, entrambi liste di stringhe. L’attributo *conferma* contiene possibili risposte affermative, mentre l’attributo *rifiuta* contiene termini di valore opposto. Il metodo *gameOverChoice* quando richiamato mostrerà a schermo le opzioni disponibili al giocatore, ovvero riprendere la partita dal momento precedente il game over oppure terminare il gioco. In caso di risposta non valida al giocatore sarà mostrata nuovamente la stessa schermata. Il compito del metodo sarà quello di riconoscere il comando inserito dal giocatore.

## Diagramma delle classi (GameSettings)

## 

## Specifica algebrica (Map)

### Specifica sintattica

Sorts: Map, int, key, value, boolean

* newmap()-> Map

crea una nuova map

* get(Map, key) -> value

restituisce il valore associato alla chiave

* put(Map,key, value) -> Map

aggiunge la coppia chiave-valore nella Map

* equals(Map,Map) -> boolean

confronta tutte le coppie chiave-valore di entrambe le map

* remove(Map,key) -> Map

rimuove la coppia chiave-valore corrispondente alla chiave data in input

* replace(Map,key, value) -> Map

Posto che il valore associato alla chiave non sia nullo, questo viene sostituito dal valore dato in input

* clear (Map) -> Map

rimuove tutte le coppie chiave-valore nella map

* containsKey(Map, key) -> boolean

controlla che la map contenga la chiave

* containsValue(Map, value) -> boolean

controlla che la map contenga il valore

* putAll(Map, Map) -> Map

inserisce tutte le coppie chiave-valore della seconda map all’interno della prima

* isEmpty(Map) -> boolean

controlla che la map non contenga nessuna coppia chiave-valore

* size(Map) -> int

restituisce il numero di coppie all’interno della map

Assunzioni:

* not(boolean, boolean) -> boolean

### Specifica semantica

Declare m, m1, m2: Map; s: Set; k, k1,v2: key; v,v1,v2: value

* equals(newmap,newmap)=true
* equals(newmap,put(m1,v1,k1))=false
* equals(put(m,k,v),newmap)=false
* equals(put(m,k,v),put(m1,k1,v1)= if size(m) = size(m1) then

if contains(m,k1) and get(m,k1)=v1 and contains(m1,k) and  
 get(m1,k)=v then equals(m,m1)   
 else false

else false

* containsKey(newmap,k)=false
* containsKey(put(m,k, v),k1)= if k=k1 then true

else if containsKey(m,k1)=true then true

else false

* contaisValue(newmap,v)=false
* containsValue(put(m,k, v),v1)= if v=v1 then true

else if containsValue(m,v1)=true then true

else false

* putAll(newmap,newmap)=newmap
* putAll(newmap,put(m,k,v))=put(m,k,v)
* putAll(put(m,k,v),newmap)=put(m,k,v)
* putAll(put(m,k,v),put(m1,k1,v1))= if equals(m, m1)=true then m else then put(m,k1,v1)
* isEmpty(newmap)=true
* size(newmap)=0
* size(put(m,v,k))=size(m)+1
* get(put(m,k, v),k1)= if k=k1 then v then true

else if containsKey(m,k1)=true then v1

else error

* remove(put(m,k,v),k1)= if k1=k then m

else put(remove(m,k1),k,v)

* replace(put(m,k, v),k1,v1)= if k=k1 and v not null then put(remove(m,k),k1,v1)

else error

* clear(put(m,k,v))= newmap
* isEmpty(put(m,k,v)=false

### Specifica di restrizione

* get(newmap,k)=error
* remove(newmap,k)=error
* replace(newmap,k,v)=error
* clear(newmap)=error

## 