RobberyAdventure

Students of Computer Science

Alessandro Piccolo

Paolo Marchisella

**Esame di**

Metodi Avanzati di Programmazione

*(track M-Z)*

A.A. 2024/2025

# **Descrizione dell’avventura**

Il gioco è un’avventura testuale con alcune componenti grafiche, ambientato in epoca contemporanea in un contesto realistico.

Il giocatore deve affrontare l’avventura nei panni di un ladro di appartamenti che deve svaligiare una lussuosa villa di un ricco magnate.

L’obiettivo del giocatore è entrare nella villa per cercare tutti gli oggetti di valore.

Per fare questo, il protagonista deve superare diversi ostacoli, come porte che devono essere aperte per esplorare la mappa, una password a difesa della cassaforte, oggetti e stanze nascoste. Alcuni di questi oggetti sono fondamentali per il proseguimento dell’avventura.

Il giocatore deve inoltre combinare alcuni oggetti che trova nella mappa per superare alcuni ostacoli.

Durante la partita il giocatore può raccogliere degli oggetti che vengono inseriti nell’inventario o nello zaino della refurtiva, se gli oggetti sono preziosi.

Lo strumento fondamentale a disposizione del giocatore è la possibilità di osservare l’ambiente in cui si trova in un certo momento e ognugno degli oggetti presenti nel gioco, per ottenere degli indizi o suggerimenti per raggiungere i propri obiettivi.

# **Progettazione**

Il principio cardine che ha guidato la progettazione di alto livello del software è stato l'adozione del modello ***ECB* (Entity-Control-Boundary)**, facendo sì che a ciascuna classe venisse attribuita una responsabilità esclusivamente afferente al ramo di appartenenza della tassonomia.

Le classi del **Package Entity** possiedono la conoscenza della struttura interna degli elementi modellati nel sistema, incapsulandone le operazioni e le caratterstiche. Sono il risultato dell' attività di astrazione dati esercitata sugli elementi del dominio di interesse dell'applicazione.

* Il Package **“effects”** contiente le classi rappresentanti gli effetti di gioco applicabili dopo aver superato le particolari condizioni necessarie dello specifico oggetto:
* **ContainerEffect:** La classe rappresenta un effetto (eseguito solo dopo aver superato la completeCondition) su un qualsiasi container del gioco.
* **CurrentPositionEffect:** La classe rappresenta un effetto (eseguito solo dopo aver superato la completeCondition) sulla posizione corrente del giocatore nella partita.
* **GameEffect:**  La classe rappresenta l'insieme di tutti i possibili effetti che potrebbero essere eseguiti sulla situazione corrente della sessione di gioco.
* **InventoryEffect**: La classe rappresenta un effetto (eseguito solo dopo aver superato la completeCondition) sulla situazione corrente nell’inventario della sessione.
* **LootBagEffect:**La classe rappresenta un effetto (eseguito solo dopo aver superato la completeCondition) sulla situazione corrente nellla LootBag della sessione di gioco.
* **ObjectEffect:** La classe rappresenta un effetto (eseguito solo dopo aver superato la completeCondition) sulle proprietà di un oggetto del gioco
* **RoomEffect:** La classe rappresenta un effetto (eseguito solo dopo aver superato la completeCondition) sulla situazione corrente di una stanza del gioco, L’effetto viene conservato come composizione di possibile oggetto da aggiungere alla stanza e possibile oggetto da aggiungere alla stanza.
* **SpecialAction:**  Questa interfaccia funzionale è stata pensata per permettere ad ogni gameEffect di conservarne un attributo, avvalorato con una espressione lambda al momento della sua inizializzazione. In generale questa interfaccia rappresenta un metodo speciale che deve essere eseguito come effetto (dopo aver superato la completeCondition associata alla specifica proprietà di uno specifico oggetto).
* Il Package “**conditions”** contiene le classi che rappresentano le condizioni necessarie della specifica dell’azione di gioco, per permetterne la sua applicazione:
* **CompleteCondition:** La classe rappresenta la condizione di soddisfacimento completa (relativa all'oggetto) necessaria per eseguire effetti sulla sessione di gioco (Modifica di proprietà di altri oggetti, etc...) ed individuare il messaggio relativo al superamento delle condizioni necessarie.
* **InventoryCondition:** La classe rappresenta una condizione sull'inventario (lista di oggetti necessariamente presenti in esso) che deve essere rispettata per permettere di modificare una qualsiasi delle proprietà di un oggetto.
* **ObjectCondition:** La classe rappresenta la condizione completa su di un oggetto che deve essere necessariamente rispettata, affinché possa essere eseguita un'azione effettiva sulla sessione di gioco.
* Il Package “**Objects”** conserva la gerarchia di ereditarietà delle classi di tipo AdvObject:
* **AdvObject:** La classe rappresenta l’oggetto comune dell’avventura, conserva le caratteristiche di un oggetto di base di una qualsiasi avventura testuale.
* **Door**: La classe rappresenta un oggetto di tipo porta, estende InteractiveObject in quanto oggetto interattivo, e conserva altri attributi specifici per modellare le caratteristiche della porta del gioco.
* **InteractiveObject:** La classe rappresenta un qualsiasi oggetto interattivo dell’avventura, Estende AdvObject, infatti conserva tutti gli attributi di un normale oggetto, e in più conserva delle specifiche di azioni di gioco. Esse rappresentano l’insieme delle condizioni necessarie e degli effetti da applicare alla sessione di gioco al momento del superamento di tutti i vincoli presenti nella specifica stessa.
* **ValuableObject:** La classe rappresenta un qualsiasi oggetto di valore dell’avventura. Poiché questi oggetti sono sempre raccoglibili e necessari per il proseguimento dell’avventura, sono stati progettati per estendere InteractiveObject (poiché, appunto, necessariamente interattivi).
* Il Package **“properties”** conserva le classi rappresentanti le diverse proprietà degli oggetti e la gerarchia di ereditarietà di queste ultime.
  + **Activatable:**  La classe estende PropertyWithValue (poiché una proprietà con un valore booleano associato). La classe modella la proprietà di attivabilità di un oggetto, conserva un valore rappresentante il suo stato di attivazione.
  + **Breakable:**  La classe estende PropertyWithValue (poiché una proprietà con un valore booleano associato). La classe modella la proprietà di rompibilità di un oggetto, conserva un valore rappresentante il suo stato di rottura.
  + **Consumable:** La classe estende PropertyWithValue (poiché una proprietà con un valore booleano associato). La classe modella la proprietà di consumabilità di un oggetto, conserva un valore rappresentante il suo stato di consumazione.
  + **Container:** La classe estende Property, e si differenzia da propertyWithValue. La classe modella una proprietà di contenitore di oggetti, non conservando uno stato, ma una collection di oggetti che un altro oggetto potrebbe contenere.
  + **Fillable:** La classe estende PropertyWithValue (poiché una proprietà con un valore booleano associato). La classe modella la proprietà di riempibilità di un oggetto, conserva un valore rappresentante il suo stato di riempimento.
  + **Movable:** La classe estende PropertyWithValue (poiché una proprietà con un valore booleano associato). La classe modella la proprietà di mobilità di un oggetto, conserva un valore rappresentante il suo stato di spostamento.
  + **Openable:** La classe estende PropertyWithValue (poiché una proprietà con un valore booleano associato). La classe modella la proprietà di apribilità di un oggetto, conserva un valore rappresentante il suo stato di apertura/chiusura.
  + **Pickupable:** La classe estende PropertyWithValue (poiché una proprietà con un valore booleano associato). La classe modella la proprietà di consumabilità di un oggetto, conserva un valore rappresentante il suo stato di consumazione.
  + **PropertyWithValue:** La classe estende Property, in quanto proprietà di un oggetto, tuttavia modella una proprietà con un valore di stato associato.
  + **Property:** La classe modella una qualsiasi proprietà di un oggetto dell’avventura.
  + **Pushable:** La classe estende PropertyWithValue (poiché una proprietà con un valore booleano associato). La classe modella la proprietà di premibilità di un oggetto, conserva un valore rappresentante il suo stato.
  + **Usable:** La classe estende PropertyWithValue (poiché una proprietà con un valore booleano associato). La classe modella la proprietà di usabilità di un oggetto, conserva un valore rappresentante il suo stato d’utilizzo.
* Il Package **Types** conserva le classi che modellano le altre entità del gioco:
  + **CardinalPoint:** Enumerativo che conserva gli identificativi dei quattro punti cardinali.
  + **Command:** La classe Modella un qualsiasi comando valido dell’avventura.
  + **CommandAnalysisResult:** La classe modella il risultato del metodo analyze degli Analyzers dell’avventura. Conserva il messaggio da mostrare all’utente, l’oggetto target su cui si vuole eseguire l’azione, l’oggetto ausiliario necessario per eseguirla ( se inserito nel comando integrale) e il tipo di proprietà individuato (corrispondente allo specifico comando invocato).
  + **ConditionEvaluationResult:** La classe modella il risultato di valutazione delle condizioni di applicabilità presenti in una delle specifiche di azioni di gioco dell’oggetto target.
  + **FailingConditionMessages:** La classe conserva i messaggi di fallimento per un qualsiasi possibile fallimento della valutazione della completeCondition.
  + **GameActionResult:** La classe incapsula il risultato di un’azione sulla sessione di gioco (messaggio da mostrare all’utente e azione speciale da eseguire).
  + **GameActionSpecification:** La classe modella la specifica di un’azione sul gioco, quindi ne conserva le condizioni di applicabilità, i risultati associati al superamento delle condizioni e i messaggi da mostrare in caso di fallimento.
  + **GameDescription:** La classe modella la descrizione di una qualsiasi avventura/gioco. Può essere estesa per permetterne la sua personalizzazione.
  + **Inventory:** inventario del giocatore in una sessione di gioco. Nell'inventario vengono inseriti gli oggetti non di valore, cioé che non sono di classe ValuableObject, che vengono raccolti dal giocatore durante la partita.
  + **LootBag:** Zaino della refurtiva del giocatore, in una sessione di gioco
  + **ObjectPropertyReference:** Riferimento ad una specifica proprietà di uno specifico oggetto
  + **ParserOutput:** risultato dell'elaborazione da parte di un oggetto Parser della stringa inserita dall'utente.
  + **PassingConditionResult:** risultato della valutazione della condizione completa rappresentata da un oggetto CompleteCondition di una specifica di un'azione sul gioco, nel caso in cui tale condizione sia soddisfatta rispetto alla sessione di gioco in corso.
  + **PropertyValue:** associazione di un valore dicotomico ad una specifica tipologia di proprietà, rappresentanta dal suo identificativo PropertyType
  + **RobberyAdventure:** descrizione completa di tutte le informazioni statiche e dinamiche relative ad una sessione di gioco.
  + **Room:** stanza che compone la mappa del gioco. La classe conserva l'identificativo della stanza, il suo nome, la descrizione, i collegamenti verso le altre stanze e le eventuali corrispondenti porte, dove la corrispodenza è rispettivamente tra il punto cardinale e la stanza , e il punto cardinale e la porta.

Le classi del **Package Control** detengono esclusivamente la logica di business dell'applicazione e la offrono all'ambiente esterno in forma di servizi. Espongono un'interfaccia definita secondo il principio di *information hiding*.

Il package “**analyzers**” conserva la gerarchìa di ereditarietà delle classi specializzate nell’analisi semantica delle specifiche tipologie di comando

* **CommandAnalyzer:** la classe è specializzata nell'analisi della semantica di generiche tipologie di comandi unari e binari
* **ActivateCommandAnalyzer:** la classe è specializzata nell'analisi semantica di un comando relativo al comando ACTIVATE
* **BreakCommandAnalyzer:** la classe è specializzata nell'analisi semantica di un comando relativo al comando BREAK
* **CloseCommandAnalyzer:** la classe è specializzata nell'analisi semantica di un comando relativo al comando CLOSE
* **ConsumeCommandAnalyzer:** la classe è specializzata nell'analisi della semantica di generiche tipologie di comandi unari e binari
* **DeactivateCommandAnalyzer:** la classe è specializzata nell'analisi semantica di un comando relativo al comando DEACTIVATE
* **DropCommandAnalyzer:** la classe è specializzata nell'analisi semantica di un comando relativo al comando DROP
* **FillCommandAnalyzer:** la classe è specializzata nell'analisi semantica di un comando relativo al comando FILL
* **LookAtCommandAnalyzer:** la classe è specializzata nell'analisi semantica di un comando relativo al comando LOOK\_AT
* **MoveCommandAnalyzer:** la classe è specializzata nell'analisi semantica di un comando relativo al comando MOVE
* **OpenCommandAnalyzer:** la classe è specializzata nell'analisi semantica di un comando relativo al comando OPEN
* **PickUpCommandAnalyzer:** la classe è specializzata nell'analisi semantica di un comando relativo al comando PICK\_UP
* **PushCommandAnalyzer:** la classe è specializzata nell'analisi semantica di un comando relativo al comando PUSH
* **UseCommandAnalyzer:** la classe è specializzata nell'analisi semantica di un comando relativo al comando USE

Il Package “**observers”** conserva le classi degli observers dei comandi tecnici e dei gameObserver (Observer sul comando di visualizzazione dell’inventario e della lootBag).

* **GameObserver:** Interfaccia rappresentante qualsiasi observer del gioco, ha un solo metodo update, Il metodo esegue un update sul gioco, relativamente all'output del parser nella corrente interazione CLI.
* **InventoryCommandObserver:** La classe si occupa di gestire la richiesta dell'utente di visualizzazione dell'inventario.
* **LootBagCommandObserver:** La classe gestisce la richiesta dell'utente di visualizzazione della refurtiva (lootBag nel gergo inglese).
* **TechnicalObserver:** Interfaccia implementata dai gestori dei comandi tecnici di gioco. Conserva il solo metodo update. Esso rappresenta l'update eseguito da un comando tecnico di gioco sulla sessione.
* **MenuCommandObserver:** La classe implementa l'interfaccia dei gestori tecnici di gioco. Gestisce il comando del gioco che permette di visualizzare il menu di gioco.
* **SaveCommandObserver:** La classe implementa l'interfaccia dei gestori tecnici di gioco. Gestisce il comando del gioco che permette di salvare i progressi di gioco.
* **EndCommandObserver:** La classe modella il gestore del comando di fine gioco.
* **Engine:** la classe costituisce l'ambiente di esecuzione astratto dell'applicazione.
* **GameActionSpecificationProcesser:** la classe è specializzata nell'elaborazione di un oggetto GameActionSpecification, di cui conosce la struttura Possiede dei metodi specializzati nell'elaborazione di una GameActionSpecification a diversi livelli di granularità, ognuno detentore di tale logica di elaborazione
* **GameControl:** la classe è specializzata nel funzionamento di alto livello della partita. La classe si occupa del coordinamento generale delle attività inerenti al funzionamento della partita, delegando l'analisi della semantica dei comandi inseriti dall'utente e l'esecuzione delle operazioni sulla partita a classi specializzate
* **Parser:** la classe è specializzata esclusivamente nell'analisi lessicale e sintattica di stringhe rispetto ad una grammatica fissata per i comandi, per un generico gioco.
* **PositionChangeHandler:** la classe è specializzata nell'analisi semantica del risultato dell'elaborazione della stringa inserita dall'utente da parte del Parser rispetto a un comando di movimento tra le stanze del gioco, e nella modifica coerente della poszione del giocatore nella mappa del gioco

Le classi del **package “Boundary**” si occupano esclusivamente dell'interazione tra il sistema e l'utente.

* Il **Package “GUI”** conserva le classi necessarie per l’interazione per mezzo di interfaccia grafica (GUI) con l’utente.
* **EndCommandGUI:** La classe swing rappresenta la GUI per interagire con il sistema per l'abbandono del gioco (uscita dal gioco).
* **FrameCreator:** La classe si occupa di creare un frame di base per garantire la visibilità di dialog durante l'esecuzione del gioco.
* **MenuCommandGUI:** La classe swing rappresenta l'interfaccia grafica del menu del gioco.
* **RescueProgressGUI:**
* **SafeDialogGUI:** La classe swing rappresenta la GUI per permettere all'utente di interagire con il tastierino della cassaforte dell'Adventure.
* **SaveCommandGUI:** La classe swing rappresenta l'interazione con l'utente per il salvataggio dei progressi di gioco.

- Il Package “**Services”** conserva le classi che rappresentano le diverse configurazioni REST del server da avviare.

* **GameService:** La classe rappresenta la configurazione REST del server per permettere richieste http per quanto riguarda i servizi di salvataggio di progressi di gioco e recuperi di questi ultimi.
* **ObjectService :** La classe rappresenta la configurazione REST del server per permettere richieste http per quanto riguarda i servizi di inserimento degli oggetti in uno storage e recupero degli stessi dallo storage.
* **ClientManager:** La classe si occupa di gestire le richieste al server REST, per quanto riguarda il servizio di aggiunta e richiesta di oggetti e di salvataggio e recupero di progressi di gioco.
* **DatabaseManager:** La classe si occupa di gestire il database, di conservarne la connessione, di effettuare creazioni di tabelle, inserimento di dati di oggetti e progressi di gioco, e infine di eseguire queries su di esso.
* **ServerManager:** La classe si occupa di gestire un qualsiasi server (http) e di permetterne l'avvio.

Per alcune classi infine non risulta possibile individuare una collocazione coerente rispetto alla tassonomia Entity-Control-Boundary

* Il Package **exceptions** conserva tutti i nuovi tipi di eccezioni creati per supportare specifiche condizioni di gioco.
  + **AlreadyLinkedException:** questa eccezione viene lanciata quando si tenta di associare un oggetto ad un altro, il quale però partecipa già ad un' associazione con un altro oggetto
  + **AmbiguousCommandException:** questa eccezione viene lanciata quando un comando risulta ambiguo
  + **DuplicateException:** questa eccezione viene lanciata quando si tenta di aggiungere un elemento già presente in una collezione o in una sequenza di elementi
  + **IdAlreadyTakenException:** questa eccezione viene lanciata quando si tenta di assegnare ad un oggetto un identificativo già assegnato ad un altro oggetto
  + **EndGameException:** questa eccezione viene lanciata quando il gioco deve terminare, sia per la terminazione ordinaria dell'applicazione, che per casi speciali legati alla partita, come la fine dell'avventura
  + **InconsistentInitializationException:** questa eccezione viene lanciata quando si tenta di costruire un oggetto portandolo in uno stato inconsistente rispetto alla semantica della classe
  + **NotValidSentenceException:** questa eccezione viene lanciata quando la stringa inserita dall'utente non risulta corretta sul piano lessicale o sul piano sintattico o sul piano semantico di una specifica tipologia di comando
  + **NotValidTokenException:** questa eccezione viene lanciata quando un token proveniente da una stringa di input non rientra nel lessico supportato dal gioco
  + **PasswordGuessedException:** questa eccezione viene lanciata quando il giocatore inserisce la password corretta per la cassaforte interna (ObjectId : INNER\_SAFE)
* Il package **identifiers** conserva gli enumerativi rappresentanti gli identificativi di numerose entità dell’avventura.
  + **CommandType:** questo enumerativo contiene gli identificativi delle tipologie di comando disponibili nel gioco
  + **ObjectId**: questo enumerativo contiene gli identificativi di tutti gli oggetti presenti nel gioco, interattivi e non interattivi
  + **PrepositionType:** questo enumerativo contiene gli identificativi di tutte le preposizione supportate dal gioco che l'utente può utilizzare nella scrittura dei comandi. Ad ogni identificativo è associato un insieme di parole che rappresentano le preposizioni per quella tipologia, in relazione alle tipologie di comando
  + **PropertyType:** questo enumerativo contiene gli identificativi delle tipologie di proprietà aventi un valore dicotomico, conferibili agli oggetti presenti nel gioco. Ogni costante possiede un insieme di parole che rappresentano i comandi inseribili all'utente che fanno riferimento alla specifica tipologia di proprietà.
  + **RoomId:** questo enumerativo contiene gli identificativi di tutte le stanze che compongono la mappa in cui il giocatore può spostarsi nel gioco
* Il package **utilities** conserva classi che danno supporto alla realizzazione di alcune funzionalità del software:
  + **DatabaseGameTable:** tupla di una tabella che associa un oggetto RobberyAdventure ad un identificativo
  + **DatabasePopulationHandler:** gestore del popolamento della base di dati con le tuple che rappresentano tutti gli oggetti del gioco, interattivi e non interattivi
  + **Preposition:** preposizione riconoscibile da un oggetto Parser
  + **PropertyCommandsCorrespondence:** mappa che associa ad un PropertyType l'insieme di CommandType corrispondente.
  + **Utils:** contenitore di servizi generici di varia natura: operazioni su file, metodi per la gestione di stringhe e Collection generiche

**Individuazione delle classi - Esempi notevoli**

* La classe Control **GameControl** è specializzata nel coordinamento di alto livello del funzionamento di una sessione di gioco. I suoi servizi vengono utilizzati dalla classe **Engine** che in questo modo non detiene anche la logica del funzionamento della partita del gioco, ma si limita a contenere il meccanismo di funzionamento dell’intera applicazione, occupandosi degli aspetti tecnici e delegando invece il controllo della sessione di gioco alla classe GameControl.
* Le classi contenute nel Package **“analyzers”** sono state individuate per eseguire un’analisi semantica del contenuto del **parserOutput**. Perciò è stata adottata la strategia per cui il gameControl, contestualmente al tipo di comando individuato dal parser e inserito nel **parserOutput**, notifica l’analyzer corrispondente al comando (seguendo un semplice dizionario), e quest’ultimo restituisce, quindi, l’analisi semantica del comando integrale (restituendo il risultato dell’analisi come insieme di:

oggetto target individuato, oggetto ausiliario se presente, messaggio da mostrare all’utente, e risultato dell’analisi semantica (superata o meno)). Questo approccio permette di separare le responsabilità dell’analyzer e del processer della **gameActionSpecification**, in modo che il primo esegua solo un’analisi semantica del comando integrale, e il secondo, sulla base del risultato restituito dal primo, completi il processo di analisi delle condizioni, applicazione degli effetti (contenute nelle specifiche delle azioni di gioco dello specifico oggetto target individuato).

* La porzione del progetto relativa alla partita si fonda sulla modellazione degli oggetti presenti nel gioco come elementi a cui è possibile assegnare delle proprietà. Tali proprietà sono state individuate sulla base delle azioni di interesse nella partita e tale attività progettuale ha portato all’individazione di una gerarchìa di ereditarietà che modella le proprietà assegnabili ad un oggetto, mostrate nel diagramma delle classi, nel paragrafo seguente.

In questo modo sono state individuate le classi **Property**, **Container**, **PropertyWithValue**, **Consumable**, **Activatable**, **Breakable**, **Openable**, **Fillable**, **Pushable**, **Movable**, **Pickupable** e **Usable**.

* La porzione del progetto relativo alla logica del funzionamento della partita si basa essenzialmente su un modello sofisticato frutto di una profonda attività di astrazione dati.

La progettazione di tale modello aveva come obiettivo l’automatizzazione del processo di elaborazione delle azioni da applicare alla partita , fornendo un’alternativa al modello che si era adottato precedentemente, molto più elementare e intuitivo. Quest’ultimo modello prevedeva l’implementazione di un metodo di un’interfaccia funzionale che sarebbe dovuta essere fornita ad hoc per ogni oggetto presente del gioco, moltiplicando la logica dei controlli e di applicazione delle azioni per il numero di oggetti, conferendo al progetto enormi e numerosissime ridondanze.

Il nuovo modello prevede invece l’assegnazione ad ogni oggetto interattivo di una mappa che associa ad ogni proprietà posseduta dall’oggetto un insieme di comandi ad essa relativi, e ad ogni comando una specifica dell’azione di gioco da eseguire quando il relativo comando viene invocato sull’oggetto proprietario della mappa.

Il cuore di questo modello è costituito dalla classe **GameActionSpecification** che rappresenta una specifica relativa ad un’azione di gioco, composta come segue:

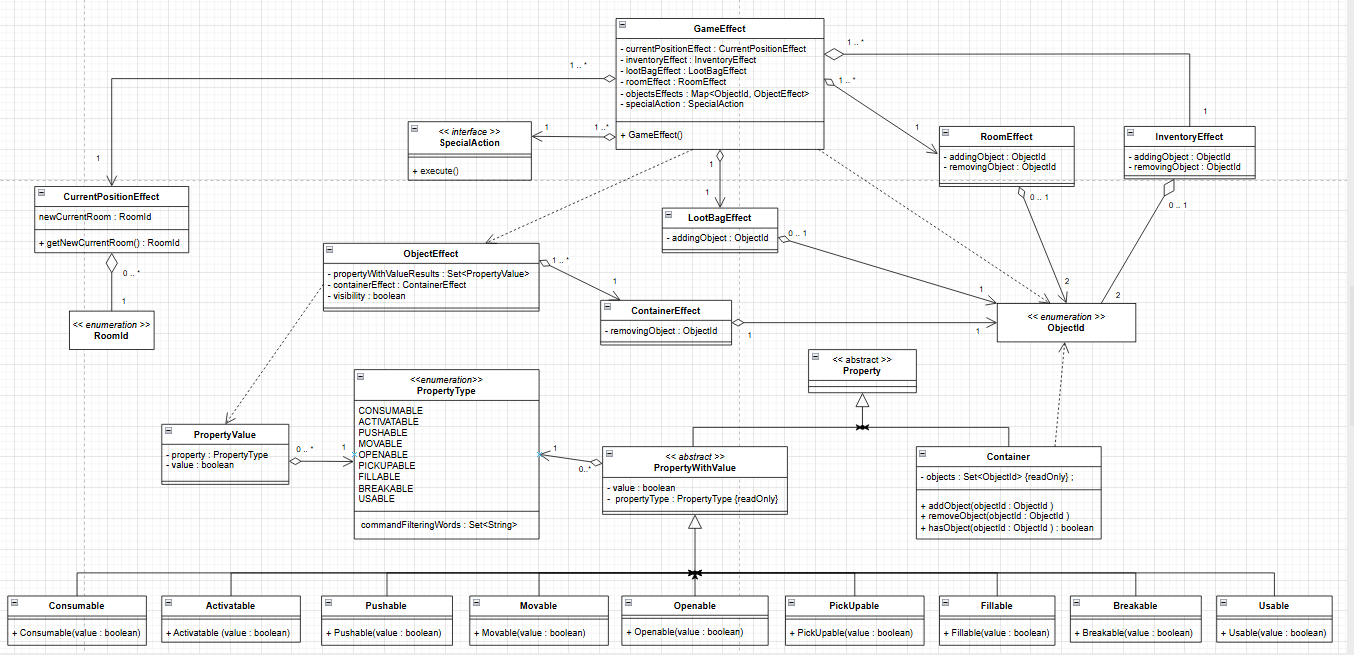
* Una condizione completa di applicabilità dell’azione di gioco, da valutare sulla partita, rappresentata da un oggetto di classe **CompleteCondition**
* Un risultato del superamento della valutazione della condizione precedente, a sua volta composto da un oggetto di classe **GameEffect** e da un messaggio che notifica l’applicazione dell’azione sul gioco. L’oggetto GameEffect rappresenta l’insieme degli effetti dell’azione sulla partita.
* Un insieme di messaggi relativi al fallimento della valutazione della condizione completa sulla partita in corso, rappresentato da un oggetto di classe **FailingConditionMessages**

Ognuna delle classi sopracitate CompleteCondition, FailingConditionMessages e PassingConditionResult e tutte le loro sottoclassi costituiscono una notazione che nell’insieme compone dettagliatamente un’intera specifica di un’azione di gioco.

La classe **GameActionSpecificationProcesser** detiene la logica per la valutazione della condizione espressa dalla classe **CompleteCondition**, per l’applicazione degli effetti sulla partita espressi dalla classe **GameEffect** in caso di soddisfacimento della condizione, e per il reperimento dei corrispondenti messaggi di fallimento contenuti nella classe **FailingConditionMessages**, nel caso in cui una delle sottocondizioni della condizione completa non sia soddisfatta.

## 

## **Diagramma delle classi**

Di seguito è mostrato il diagramma delle classi UML di una porzione significativa del progetto :

Il diagramma delle classi rappresenta la porzione del progetto relativa alla specifica dell’effetto di un’azione sulla partita e la gerarchia di ereditaritetà delle classi che modellano le proprietà degli oggetti.

La classe **GameEffect** rappresenta l’insieme di tutti gli effetti che devono essere applicati alla situazione corrente della sessione di gioco, dopo la valutazione della condizione completa associata alla specifica dell’azione di gioco associata alla particolare proprietà dell’oggetto. **GameEffect** si presenta come aggregazione di diversi attributi rappresentanti specifici effetti sulla sessione di gioco:

* Un attributo di Interfaccia **SpecialAction**. Quest’ultima è un’interfaccia funzionale, pensata per permettere il suo avvaloramento all’inizializzazione del **GameEffect**, per mezzo di una espressione lambda specifica associata all’azione speciale da applicare alla sessione di gioco.
* Un attributo di classe **ObjectEffect**. Questa classe rappresenta un effetto eseguito sulle valori caratteristici associati alle proprietà di uno specifico oggetto, dopo averne valutato le condizioni di applicabilità necessarie. La classe è a sua volta un’aggregazione di altri attributi:

- Un attributo che conserva gli effetti sull’oggetto sulle proprietà con valore di stato. Queste ultime sono rappresentate dalla classe **PropertyValue**. Questa conserva l’identificativo **PropertyType** associato alla proprietà di cui si vuole cambiare il valore di stato ed il valore di stato risultato dell’applicazione dell’effetto.

- un attributo di classe **ContainerEffect**. la classe rappresenta l’effetto da applicare alla proprietà (senza valore di stato) **Container** se l’oggetto su cui si stanno applicando gli effetti ha una proprietà di questo tipo.

- un valore booleano rappresentante l’effetto sulla visibilità dello specifico oggetto.

* Un attributo di classe **CurrentPositionEffect**. Esso rappresenta un effetto (eseguito solo dopo aver superato la completeCondition) sulla posizione corrente del giocatore nella partita. L'effetto viene conservato come **RoomId** (rappresentante l’identificativo della stanza) della nuova stanza corrente. Esso sarà usato per impostare la stanza associata a questo id come nuova currentRoom all'interno della gameDescription.
* Un attributo di classe **LootBagEffect**. Esso rappresenta un effetto sulla situazione corrente dellla **LootBag** nella sessione di gioco. L’effetto viene conservato come composizione di un **ObjectId** (addingObject), il cui oggetto associato deve essere aggiunto alla **LootBag**.
* Un attributo di classe **InventoryEffect** che rappresenta l’eventuale effetto sull’inventario dell’esecuzione dell’azione applicata sul gioco, in seguito alla valutazione con successo della condizione completa della specifica dell’azione stessa. La classe **InventoryEffect** ha un attributo **ObjectId** che conserva l’identificativo dell’oggetto da rimuovere eventualmente dall’inventario e un attributo **ObjectId** che conserva l’identificativo dell’oggetto da aggiungere eventualmente all’inventario
* Un attributo di classe **RoomEffect** che rappresenta l’eventuale effetto sulla stanza in cui il giocatore si trova nell’istante corrente, dell’esecuzione dell’azione applicata sul gioco, in seguito alla valutazione con successo della condizione completa della specifica dell’azione stessa. La classe **RoomEffect** ha un attributo **ObjectId** che conserva l’identificativo dell’oggetto da rimuovere eventualmente dalla stanza corrente e un attributo **ObjectId** che conserva l’identificativo dell’oggetto da aggiungere eventualmente alla stanza corrente
* Gli oggetti della classe enumerativo ObjectId sono aggregati negli oggetti delle classi **Container**, **ContainerEffect**, **LootBagEffect**, **RoomEffect**, **InventoryEffect** e rappresentano gli identificativi di tutti gli oggetti presenti nel gioco, interattivi e non interattivi.

Gli enumerativi **PropertyType** rappresentano gli identificativi delle tipologie di proprietà aventi valore dicotomico, conferibili agli oggetti presenti nel gioco.

La classe astratta **Property** costituisce la radice della gerarchia di ereditarietà delle classi che rappresentano le proprietà. Non possiede membri e permette, per il principio di sostituibilità, di considerare ogni oggetto delle sue sottoclassi come un’istanza di **Property**.

La classe **Property** ha due sottoclassi dirette: la classe astratta **PropertyWithValue** che rappresenta proprietà aventi uno stato con valore dicotomico e la classe concreta **Container**, che rappresenta l’oggetto alla quale un’istanza della classe viene assegnata come un contenitore di altri oggetti.

La classe **PropertyWithValue** ha quindi un attributo boolean che rappresenta lo stato dicotomico del valore della proprietà generica e un attributo **PropertyType** che rappresenta l’identificativo della proprietà.

Dalla classe **PropertyWithValue** ereditano nove classi concrete. Si tratta di una relazione di ereditarietà per implementazione, in quanto ognuna delle sottoclassi **Consumable**, **Activatable**, **Breakable**, **Openable**, **Fillable**, **Pushable**, **Movable**, **Pickupable** e **Usable** rappresenta un’effettiva tipologia di proprietà, sfruttando la struttura della classe padre **PropertyWithValue**.

## 

## 

## 

## 

## Specifica algebrica

Specifica algebrica - String array

**Specifica sintattica:**

* sorts: array, integer, String
* operations: create(integer, integer) -> array
* assign(array, integer, String) -> array
* first(array) -> integer // restituisce lower bound
* last(array) -> integer // restituisce upper bound
* eval(array, integer) -> String // restituisce l’elemento

**Specifica semantica:**

* declare: x, y, i, j: integer; e: String; v: array
* first(create(x,y)) = x
* first(assign(v, i, e)) = first(v)
* last(create(x, y)) = y
* last(assign(v, i, e)) = last(a)
* eval(create(x, y), i) = error
* eval(assign(v, i, e), j)=if jlast(v) then error else if i=j then e else eval(v, j)

## **Dettagli implementativi**

Per ciascun argomento del corso spiegare se e come è stato utilizzato all’interno del progetto.

### **Programmazione generica**

Nella classe **Utils** (package avdenture.utilities) è stato definito il metodo generico union() che dati due oggetti Collection di tipo generico T, restituisce l’insieme unione (in senso matematico) di tutti gli elementi presenti nelle due Collection, anch’esso di tipo generico T.

### **File**

I file sono stati utilizzati per memorizzare in maniera persistente i dati relativi alle descrizioni delle stanze del gioco all’interno del path ***“./resources/descriptions/”.***Al momento dell’inizializzazione delle stanze della RobberyAdventure, viene invocato il metodo (loadFileDescriptionInString()) della classe **Utils** del Package utilities. Quest’ultimo restituisce una stringa con l’intero testo conservato nel file, inizializzando così la stringa di descrizione della stanza.

Il metodo di **Utils** permette, dato in input un File di testo, di inserire in una Stringa l’intero testo del file, leggendo linea per linea, attraverso un **BufferedReader** il contenuto dello stream del **FileReader.**

Allo stesso modo viene utilizzato un file di testo per memorizzare le stopwords, che saranno poi prelevate da un oggetto di classe Engine per inizializzare un set di Stringhe (contenitore delle stesse stopwords).

### **Database (JDBC)**

E’ stata utilizzata l’interfaccia JDBC, per permettere di conservare all’interno di un database locale all’applicazione, dati generici degli oggetti dell’avventura quali id dell’oggetto, nome, descrizione, e alias. La popolazione del database avviene per mezzo di una classe specifica la quale contiene uno script (main) che, avviato, permette di popolare l’intero database con tutti i dati generici degli oggetti dell’avventura. Questi dati vengono raccolti dal db ad ogni avvio del programma, per mezzo di queries che selezionano l’oggetto specifico attraverso il suo id. Inoltre, è la classe **DatabaseManager** che si occupa di conservare la connessione al database locale e di effettuare update e queries su di esso.

L’interfaccia JDBC è stata, inoltre, utilizzata per permettere il salvataggio dei progressi di gioco e il recupero degli stessi dal database. Sono state, tuttavia, incontrate difficoltà riguardanti la serializzazione dell’oggetto gameDescription, perciò questa funzionalità risulta implementata (tutte le classi del DatabaseManager sono presenti, e i metodi di recupero e salvataggio del gioco implementati), ma il suo funzionamento dà luogo a eccezioni di Stack Overflow per motivi che siamo riusciti ad individuare, ma non a risolvere per via di un’iniziale scelta progettuale riguardante i links tra le stanze del gioco.

### **Lamba Expression (compresi stream e pipeline)**

* Il metodo update() della classe **InventoryCommandObserver** fa uso di una pipeline applicata ad uno stream generato a partire dalla lista di oggetti contenuti all’istante corrente nell’inventario del giocatore, che mappa ciascuno oggetto al proprio attributo name e raccoglie gli oggetti così mappati in una lista di string che vengono poi stampati nella visualizzazione del contenuto dell’inventario.
* Il metodo costruttore della classe **Engine** costruisce il proprio attributo di classe **Parser** passando al suo costruttore quattro oggetti di tipo interfaccia funzionale BiPredicate, fornendo per ciascuno di essi un’implementazione specifica per il riconoscimento di un token come comando, oggetto, stanza o preposizione.
* Il metodo processInventoryCondition() della classe **GameActionSpecificationProcesser** fa uso di una pipeline applicata ad uno stream generato a partire dalla lista di oggetti contenuti all’istante corrente nell’inventario del giocatore, che mappa ciascuno oggetto al proprio attributo id e raccoglie gli oggetti così mappati in una lista diObjectId, utile all’attività interna del metodo.
* Nel metodo parse() della classe **Parser** viene ordinato l’insieme di tutti i nomi degli oggetti della stanza e delle stanze della mappa specificandone il criterio (la lunghezza delle stringhe) attraverso un oggett Comparator la cui implementazione viene fornita con una espressione lambda.
* Il metodo getLook() della classe **Room** fa uso di tre pipeline su stream. La prima preleva dagli oggetti contenuti all’istante corrente nella stanza solo quelli visibili. La seconda filtra gli oggetti, scartando quelli che sono istanze della classe **Door** e mappandoli al proprio attributo name, conservandoli infine in una nuova lista. La terza pipeline è la più complessa. Essa genera una pipeline a partire dalla lista di oggetti visibili ottenuti dalla prima pipeline, filtra tali oggetti selezionando solo quelli che sono istanza di **Door** e li mappa ad una stringa che concatena il loro attributo name ad un carattere di spazio e all’attributo name della stanza corrispondente.
* Il metodo toLowerCaseStringList della clase **Utils** utilizza una pipeline generata a partire dal parametro words, mappando ciascun oggetto ad una stringa corrispodente avente tutti i caratteri minuscoli e memorizzandoli in una nuova lista di stringhe, che viene poi restituita dal metodo.

### **SWING**

Le SWING sono state utilizzate per l’interazione con l’utente per mezzo di interfacce grafiche. Le classi SWING create riguardano:

* Una finestra **JDialog** (**SafeDialogGUI**) per l’interazione con il tastierino della cassaforte. Quest si compone di una **textArea** con il nome della cassaforte, una **JPasswordField** per l’inserimento del pin della cassaforte, e dei **Jbutton** per l’inserimento della password e la chiusura della **JDialog**.
* Una finestra **JDialog (MenuCommandGUI)** per l’interazione con il menu grafico dell’avventura. Il menu a sua volta, a seconda del Jbutton con cui si interagisce, permette di interagire con nuove JDialog, come la Dialog per l’abbandono del gioco (**EndCommandGUI**), una Dialog (**SaveCommandGUI**) per il salvataggio dei progressi di gioco (Funzionalità non operativa) , e una Dialog (**RescueProgressGUI)** per il recupero dei salvataggi di gioco (Funzionalità non operativa).
* La JDialog per l’abbandono del gioco (**EndCommandGUI**) si compone di un semplice insieme di **JButtons**, che permettono di confermare l’uscita dal gioco, oppure di continuare a giocare.
* La Dialog per il salvataggio del gioco (**SaveCommandGUI)** si compone di una **textArea** per l’inserimento del nome del nuovo salvataggio, e un **JButton** per confermarne il salvataggio. Se viene portata a termine l’interazione con Il **Jbutton**, la classe sopracitata effettuerà una chiamata REST per interagire con il database, e quindi salvarne i progressi di gioco all’interno.
* La Dialog per il recupero dei progressi di gioco (**RescueProgressGUI)** si compone di una Jlist utilizzata per mostrare tutti i nomi dei salvataggi di gioco aggiunti nel database, e un **JButton** per permettere la selezione di uno di questi dalla Jlist. E’ stato quindi implementato un **ListSelectionListener** per permettere di individuare la scelta del salvataggio da caricare. L’action performed del **JButton** richiede, attraverso chiamata REST al server, il salvataggio associato al nome selezionato dall’utente, che viene quindi salvato in un riferimento d’istanza della **RescueProgressGUI**.

### **Thread e programmazione concorrente**

E’ stata definita un’implementazione dell’interfaccia **Runnable** per mezzo della classe **ServerManager.** Questa classe conserva al suo interno un riferimento al server Http inizializzato, e definisce il metodo run() di Runnable, per permettere l’avvio e quindi l’esecuzione parallela del servizio Server durante l’intero flusso di gioco.

### 

### **Socket e/o REST**

Sono stati implementati diversi servizi REST:

* Servizio REST per salvataggio dei progressi di gioco e recupero dei salvataggi: Questo servizio è incapsulato nella classe **GameService**. Questa contiene un riferimento al database per permettere di effettuare richieste (update e queries sui salvataggi di gioco). Inoltre conserva due metodi GET e un metodo PUT. Il metodo GET per la richiesta del salvataggio di gioco verrà invocato quando il **ClientManager** eseguirà una richiesta http al server utilizzando il verbo sopracitato.
* Il server riceverà la richiesta solo se avviato e configurato con la classe di servizio REST gameService.
* Il server inoltre viene gestito per mezzo della classe **ServerManager**, che nel suo costruttore riceve i parametri di configurazione del server (classe service REST) e l’URI di localizzazione del server.
* Il **ClientManager** conserva questi metodi per accedere al servizio REST sui salvataggi di gioco:
  + **addGameSavingRequest**() : Il metodo si occupa di aggiungere il salvataggio dei progressi di gioco (gameDescription) in uno storage, per mezzo di una richiesta http ad un server REST.
  + **getAllNamesGamesRequest() :** Il metodo fa richiesta http al server REST, restituendo tutti i nomi dei salvataggi di gioco aggiunti nello storage.
  + **getGameSavingRequest()** : Il metodo si occupa di fare richiesta del salvataggio di gioco corrispondente al gameId passato per parametro al metodo, tramite chiamata al servizio REST del server.

* Il server REST per ognuna delle precedenti richieste http avrà i metodi:
  + **addGameSaving()** : Il metodo si occupa di convertire la tupla serializzata (json) in un oggetto di classe opportuna, di recuperarne i valori effettivi e di effetuare una richiesta al dbManager per salvare i progressi di gioco raggiunti.
  + **getAllGameSaving() :** Il metodo restituisce, attraverso una query del dbManager, tutti i nomi dei salvataggi di gioco presenti nel db.
  + **getGameSaving()** : Il metodo restituisce, effettuando una chiamata al db, il salvataggio di gioco associato all'id passato nel path della richiesta http al server.
* Allo stesso modo è stato implementato il servizio REST per la popolazione degli oggetti dell’avventura nel database, e il recupero di questi ultimi. E’ stata creata quindi una classe di servizio REST specifica per permettere al server http di ricevere richieste dal **ClientManager** per l’inserimento di oggetti e recupero di essi.
* Il **ClientManager** conserva questi metodi per accedere al servizio REST sull’inserimento di oggetti dell’avventura e recupero di questi ultimi :
  + **populate()** : Il metodo si occupa di effettuare una richiesta http al server (di aggiunta oggetto allo storage per ogni oggetto che sarà successivamente istanziato nell'avventura.
  + **addObjectRequest() :** Il metodo si occupa di richiedere al server REST, tramite richiesta http l'aggiunta dell'oggetto (parametro) in uno storage.
  + **getObjectRequest()** : Il metodo si occupa di richiedere al server REST, tramite richiesta http l'oggetto contenuto nello storage che abbia l'id passato come parametro.

* Il servizio REST **ObjectService** per ognuna delle precedenti richieste http avrà i metodi:
  + **addObject()** :Il metodo REST si occupa di convertire il jsonObject in AdvObject e di chiamare il metodo di inserimento (del dbManager) di un oggetto all'interno del db.
  + **getObject()** :Il metodo REST permette, attraverso l'inserimento dell'id dell'oggetto nel path della richiesta http, di restituire l'oggetto associato all'id sopracitato.