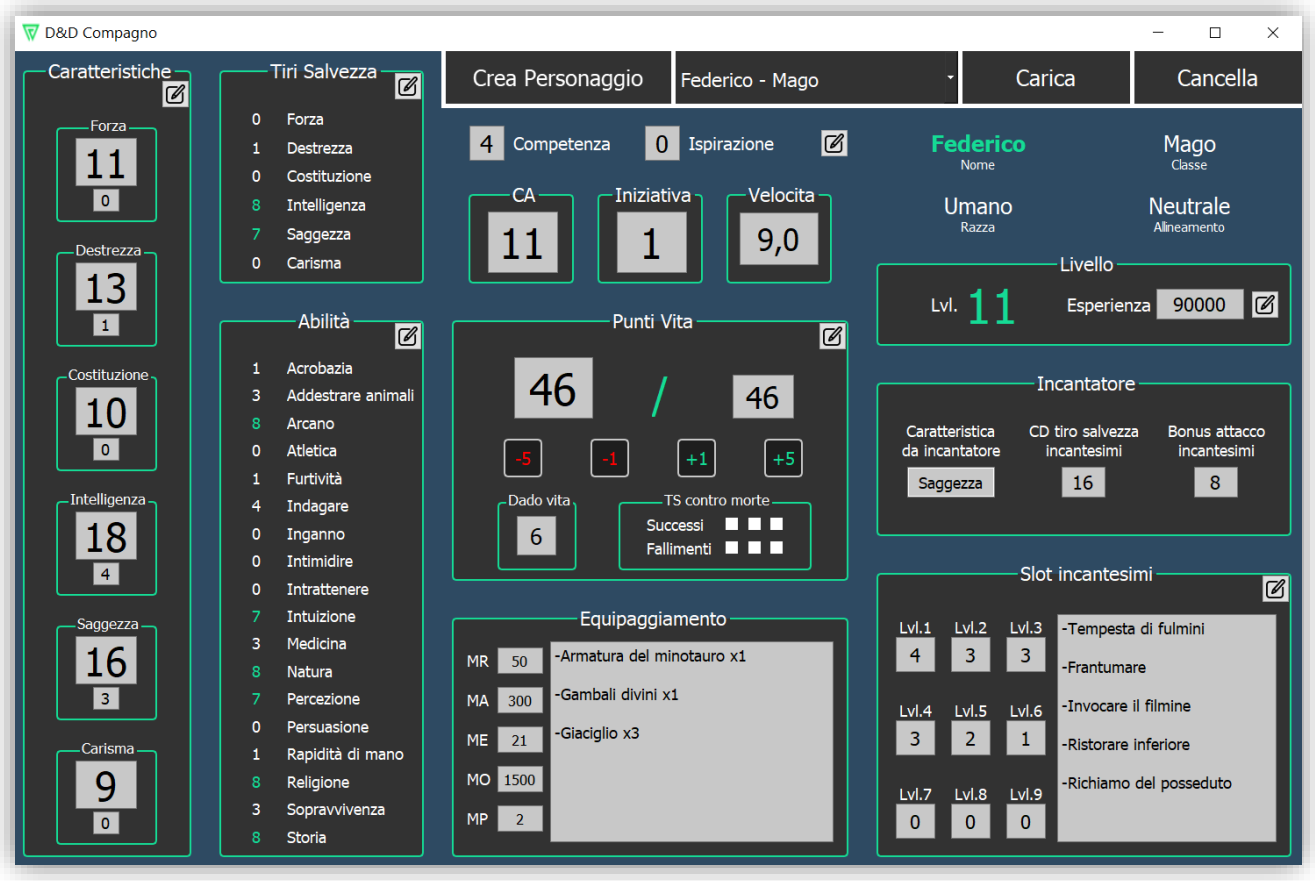
D&D Compagno

Progetto di Programmazione ad Oggetti

Anno accademico 2020/21

Federico Ballarin 1123718



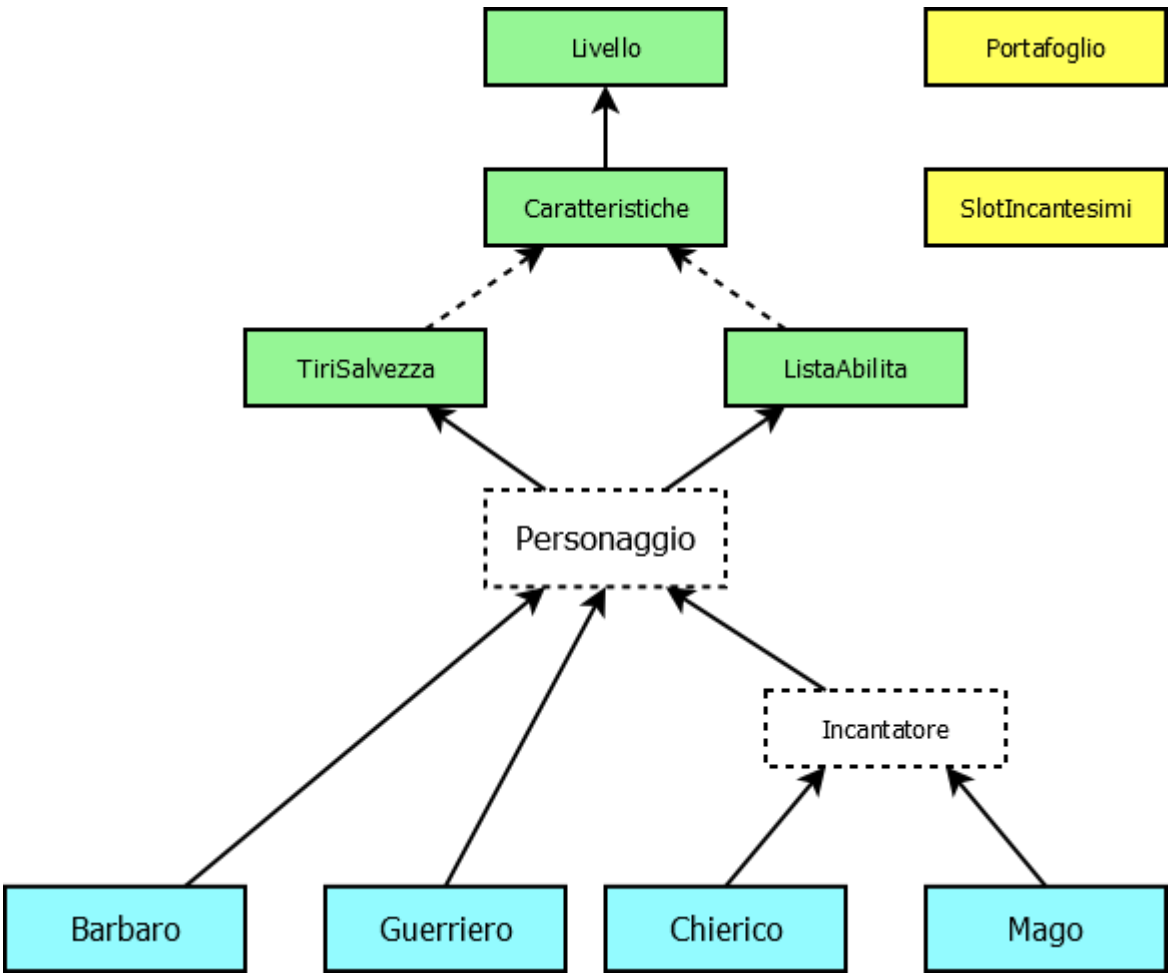
Introduzione

Dungeons & Dragons (abbreviato come DnD/D&D) è un gioco di ruolo fantasy che viene giocato seduti intorno ad un tavolo. Ogni giocatore interpreta un singolo personaggio (detto PG) che rappresenta il protagonista di un'avventura, insieme ai personaggi degli altri giocatori, in un'ambientazione fittizia di genere fantasy, sotto la guida di un giocatore detto dungeon master (DM) che descrive le situazioni i cui si trovano i personaggi. I soli oggetti necessari per giocare sono i manuali delle regole, diversi dadi e una scheda personaggio per ogni giocatore che contiene tutte le informazioni riguardanti il proprio personaggio.

In DnD le regole ed i vincoli sono molti, tutte le varie caratteristiche di ogni personaggio sono fortemente collegate, una piccola modifica ad una sola caratteristica può portare al variare di molte altre.

Lo scopo del progetto è di creare un programma che permetta al giocatore di visualizzare e gestire tutte le informazioni riguardanti i propri personaggi andando così a sostituire la scheda personaggio ed in parte anche il manuale di regole implementandole all'interno del programma. L'obbiettivo è facilitare il lavoro dei giocatori ed evitare possibili errori, spesso commessi sulla scheda cartacea dai meno esperti.

Gerarchia



Nel cuore della gerarchia è presente classe base astratta *Personaggio* che implementa il concetto di personaggio generico, andando ad implementare tutte le funzionalità di base comuni tra tutti i personaggi giocanti. La classe è realizzata attraverso una serie di specializzazioni dalle classi *Livello, Caratteristiche, TiriSalvezza, ListaAbilita.* Dato il grande numero di informazioni e funzionalità offerte dalla classe *Personaggio* si è preferito evitare di implementarle tutte all'interno della stessa classe, si è quindi cercato di modulare il più possibile il codice isolando le varie informazioni incapsulandole in più classi. La classe *Personaggio* viene quindi creata attraverso una serie di specializzazioni dalle sottoclassi andando tra l'altro a creare un caso di ereditarietà a diamante.

Le 4 sottoclassi concrete della gerarchia sono le classi *Barbaro, Guerriero, Chierico, Mago*; Esse rappresentano le classi a cui appartiene ogni personaggio. Le classi *Barbaro* e *Guerriero* derivano direttamente dalla classe base *Personaggio*, mentre le classi *Chierico* e *Mago* derivano indirettamente dalla classe *Personaggio,* passando dalla classe *Incantatore,* che rappresenta tutti i personaggi che fanno uso di magia.

Le ultime due classi della gerarchia sono la classe *Portafoglio,* che rappresenta il numero di monete in possesso ad un personaggio, presente come campo dati pubblico della classe *Personaggio;* E la classe *SlotIncantesimi* che rappresenta il numero di incantesimi che ogni incantatore è in grado di lanciare in un determinato momento, presente come campo dati privato nella classe *Incantatore.*

Molti dei campi dati nelle varie classi della gerarchia sono stati dichiarati come *int* piuttosto che *short* anche se spesso sono dei valori compresi tra 0 e 20, questa scelta è stata effettuata per due motivi:

- *Short* ovviamente permette di risparmiare spazio in memoria ma non sempre questo significa una maggiore velocità di esecuzione del programma, spesso l'hardware gestisce in maniera più efficiente il tipo *int* perché è considerato il tipo di default.
- Il tipo *int* permette di interagire meglio con molte delle librerie, ad esempio la maggior parte dei widget Qt gestiscono valori di tipo *int*, si è così evitato di fare molte conversioni inutili.

La gerarchia è stata realizzata in modo tale da non utilizzare nessuna classe proprietaria del framework Qt, potrebbe quindi essere utilizzata anche per altri progetti.

Metodi polimorfi

La classe *Personaggio* ha i seguenti metodi virtuali puri pubblici :

- virtual Personaggio* clone() const =0 : Classico metodo di clonazione polimorfa, la funzione viene implementata in ogni classe derivata concreta cioè dalle classi *Barbaro, Guerriero, Chierico e Mago* così come gli altri metodi virtuali puri.
- virtual int getDadoVita() const = 0 : Ritorna uno short con il valore del dado vita del personaggio, ogni classe ha un proprio dado vita che viene utilizzato per determinare il numero di punti vita di un personaggio e come crescono all'aumentare del livello del PG.
- virtual std::string getClasse() const = 0 : Ritorna una stringa contenente il nome della classe del personaggio.
- virtual Classi getEnumClasse() const = 0 : Uguale alla precedente ma il tipo di ritorno è Enum.
- virtual void setCompetenzeBaseTiriSalvezza() = 0 : Assegna al PG competenza nei tiri salvezza base, ogni PG ha di default competenza in 2 tiri salvezza in base alla classe. Ad esempio se il PG è un guerriero gli verrà assegnata competenza nei tiri salvezza di forza e costituzione che sono le sue competenze base.

Molti dei metodi della classe *Personaggio* sono dichiarati virtuali per motivi di estendibilità, quindi anche nel caso l'implementazione iniziale che la classe fornisce è sufficiente per tutte le classi concrete che sono state realizzate. Nelle ultime versioni del gioco originale da tavolo le classi sono molte di più di quelle

implementate in questo progetto, marcare i metodi virtuali permette alle nuove classi che verranno create di modificare l'implementazione dei metodi a proprio piacimento.

La classe *Incantatore* ha invece i seguenti metodi virtuali puri pubblici, che riguardano le capacità magiche del PG:

- virtual SlotIncantesimi suggestedSlotIncantesimi() const = 0 : Ritorna uno *SlotIncantesimi* che consiste in un vector di int contenente il numero di incantesimi, per ogni livello, che il PG è in grado di lanciare di default, ogni incantesimo ha un livello che rappresenta la sua potenza.
- virtual std::string getCaratteristicaIncantatore() const = 0 : Ritorna, attraverso una stringa, la caratteristica da incantatore del PG usata per il calcolo dei danni degli incantesimi.
- virtual int getCDTiroSalvezzaIncantesimo() const = 0 : Effettua il calcolo per ottenere la CD(classe difesa) del tiro salvezza dell'incantesimo e lo ritorna. Questo valore rappresenta il valore da superare da chi riceve l'incantesimo per ridurne gli effetti subiti.
- virtual int getModificatoreAttaccoIncantesimo() const = 0 : Effettua e ritorna il risultato del calcolo del modificatore di attacco dell'incantesimo, che corrisponde al danno da aggiungere in caso di successo nel lancio di un incantesimo.

Contenitore

Il contenitore di oggetti polimorfi realizzato è una lista singolarmente linkata templatizzata dove sono state implementate le principali funzionalità ed operatori che ci si aspettano da un contenitore di questo tipo necessari a garantire un utilizzo ideale. Il contenitore fa uso di iteratori, costanti e non. La scelta è ricaduta su questa tipologia di contenitore per effettuare le operazioni di inserimento e rimozione in maniera efficiente; Nonostante l'accesso casuale non sia permesso si è cercato di evitare il problema inserendo nella classe Modello (che fa da interfaccia per la Lista) un puntatore al Personaggio che è al momento selezionato, potendo cosi accedervi direttamente senza scorrere la lista. Questo perché si presume che in un'intera sessione di gioco ogni giocatore utilizzi sempre lo stesso personaggio, rendendo cosi la lista un ottimo contenitore per il contesto applicativo.

Come richiesto è stata anche realizzata la classe *DeepPointer* in aiuto alla gerarchia per la gestione della memoria, implementata tramite template in maniera indipendente.

Il modello è quindi formato da un *Contenitore* di *DeepPointer* a *Personaggio* cioè la classe base astratta della nostra gerarchia e da un puntatore al personaggio attualmente in uso. I metodi del modello non sono molti in quanto il focus del programma è di gestire in maniera esaustiva le modifiche avvenute al singolo personaggio, e questo avviene con i metodi propri delle varie classi, la selezione del personaggio avviene in genere solamente all'inizio della partita.

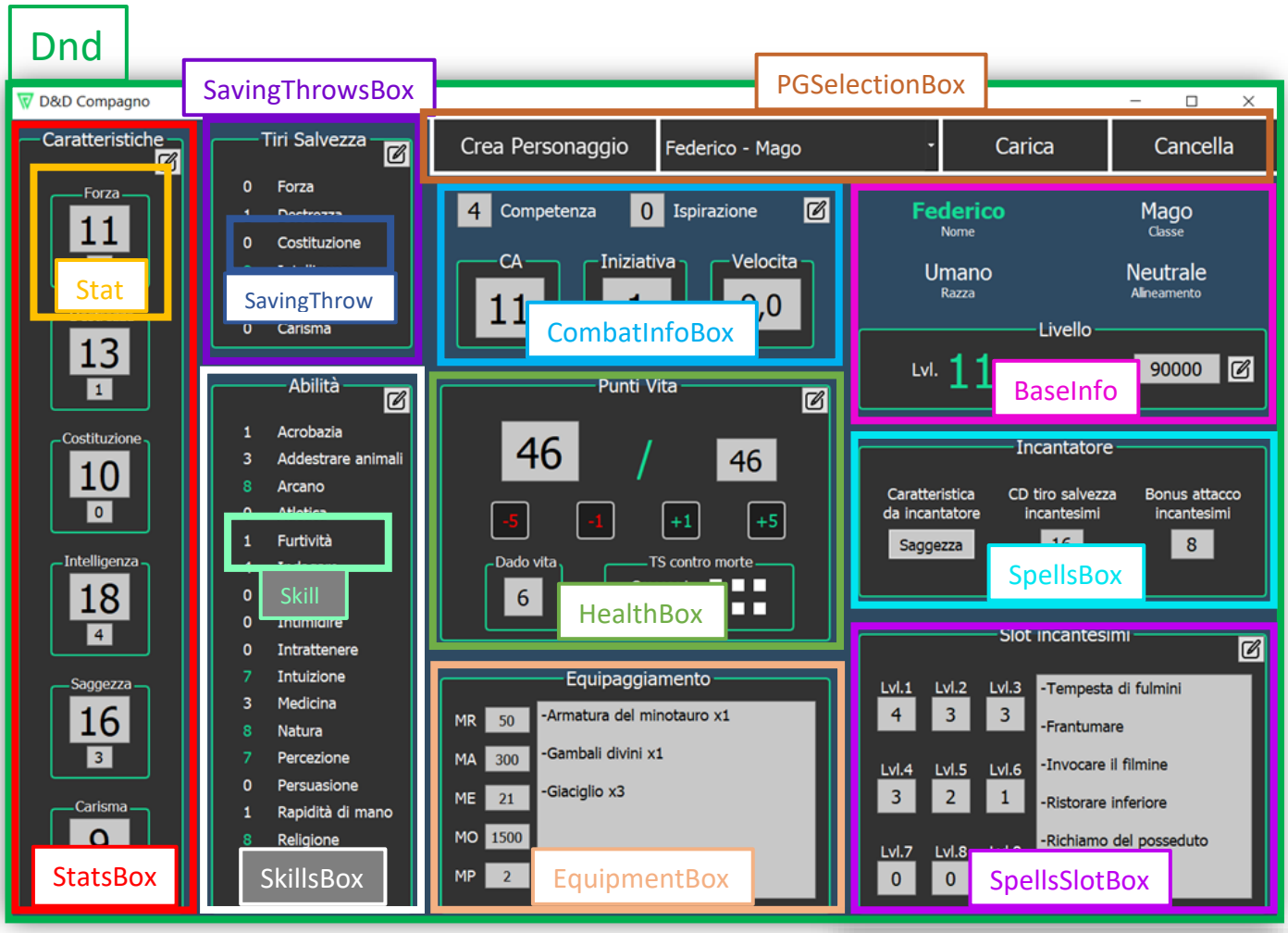
I puntatori che il modello restituisce quando si richiede un determinato personaggio non sono dei *DeepPointer* ma vengono estratti dei puntatori tradizionali a *Personaggio* prima di essere restituiti. Questo perché la modifica dei vari campi di un Personaggio non necessita di una gestione profonda della memoria, è compito dello modello gestire inserimento e cancellazione dei PG nel contenitore, quindi le funzionalità della classe *DeepPointer* non sarebbero utilizzate poi all'interno della vista.

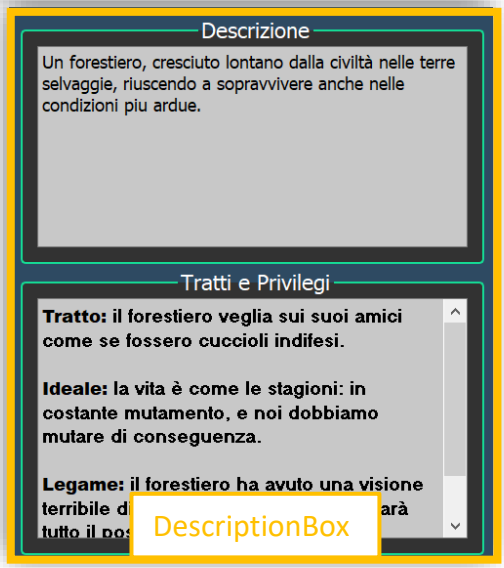
Interfaccia grafica

Per la realizzazione dell'interfaccia grafica è stato utilizzato il pattern Model-View al fine di separare il modello logico dalla GUI, la flessibilità di questo pattern, e il limite di ore per la realizzazione sono i principali motivi per cui è stata fatta questa scelta, nonostante la consapevolezza della possibilità offerta dal pattern Model-View-Controller di separare anche la view dal controller.

La GUI è stata realizzata cercando di implementare le varie componenti nel modo più modulare possibile, ogni singola componente è completamente funzionante in maniera autonoma, indipendente dalle altre, sta alla classe base della GUI *Dnd* metterle assieme e farle comunicare tra di loro sfruttando comunque funzionalità offerte dalle singole. Le singole componenti dialogano con il modello logico attraverso un puntatore al modello che è condiviso in ogni componente.

L'interfaccia utilizza volutamente un design particolare per risultare accattivante ad un target presunto di giocatori; Si sta parlando di un programma che dovrebbe essere utilizzato durante una sessione di un gioco da tavolo, non in ambiti professionali.





Lo schema mostra i nomi di tutte le classi widget che sono state realizzate per lo sviluppo della GUI. Il widget DescritionBox viene mostrato solo nel caso il personaggio selezionato non sia una classe derivata da Incantatore, andando a sostituire i widget che si occupano della gestione dei poteri magici del personaggio in quanto non ne possiede.

Note sulla qualità del codice

Si può notare che i nomi delle classi utilizzate per la realizzazione del modello siano in italiano a differenza della vista che è stata realizzata invece in lingua inglese, scelta fatta in quanto l'obbiettivo era quello di realizzare il programma per la versione italiana del gioco da tavolo, sono già presenti prodotti simili al progetto proposto in lingua inglese. Essendo la terminologia di gioco molto specifica e ben definita all'interno del manuale questa scelta ha facilitato non poco la realizzazione delle varie funzionalità riguardanti le regole del gioco e il confronto con il manuale.

Riconosco però che dal punto di vista della qualità e leggibilità del codice l'uso di più lingue possa risultare sgradevole e poco favorevole all'estendibilità del prodotto.

Note sulla compilazione

Il progetto è stato sviluppato utilizzando alcune funzionalità di c++11 quindi per la corretta compilazione vanno aggiunte nel .pro generato di default le due seguenti righe di codice:

```
CONFIG += c++11
QT += widgets
```

Nella cartella è stato consegnato ugualmente un file .pro funzionante per sicurezza, anche se non dovrebbe essere necessario.

Le caratteristiche dell'ambiente di sviluppo sono le seguenti:

- Sistema operativo: Windows 10 Home, vers. 1909
- Compilatore: MinGW 8.1.0 32-bit
- Versione libreria Qt: 5.14.0

Come richiesto dalle specifiche il progetto compila correttamente anche nella VM linux con i comandi:

```
qmake -project
qmake DnDCompagno.pro
make
```

Conteggio delle ore

Lo sviluppo del progetto ha richiesto approssimativamente 55 ore di lavoro complessivo:

- Analisi del progetto: 2 ore
- Progettazione del modello: 5 ore
- Progettazione GUI: 5 ore
- Apprendimento libreria Qt: 10 ore
- Codifica modello e contenitore: 10 ore
- Codifica GUI: 14 ore
- Testing e debugging: 6 ore
- Stesura relazione: 3 ore

Molto utile in questo contesto sarebbero state delle funzionalità di input/output su file per permettere il salvataggio e il caricamento dei personaggi creati, ma a causa del superamento delle ore consigliate per la realizzazione del progetto si è preferito ometterle.