Relazione del Progetto di Programmazione ad Oggetti 2020/2021

Coppia: Alberto Lazari, Francesco Protopapa
Relazione della consegna di **Alberto Lazari**

1 Descrizione del progetto

Il progetto è la scheda personaggio interattiva di un gioco di ruolo (una versione estremamente semplificata della scheda personaggio della quinta edizione di Dungeons and Dragons).
Si pone l'obiettivo di fornire al giocatore le informazioni relative al suo personaggio, tra cui le sue statistiche, le sue classi e gli oggetti contenuti nel suo inventario, con l'aggiunta del calcolo automatico dei tiri dei dadi, che implicano spesso molti calcoli e numerose informazioni da tenere a mente, delegando il tutto al programma.
Le dinamiche del gioco sono infatti basate su tiri di dado a diverse facce (comunemente a 4, 6, 8, 10, 12 e 20 facce, denominati d'ora in poi "d+numero" delle facce, es: d20) che determinano la riuscita o meno di una qualche azione del proprio personaggio nel gioco. È importante notare come la scheda personaggio non possa fornire l'indicazione riguardo la riuscita o meno di un'azione, in quanto necessita un confronto con le statistiche di altri personaggi, dunque viene mostrato il risultato e il giocatore agisce di conseguenza. Questo spiega la libertà di personalizzazione delle statistiche all'interno del programma e la presenza di controlli limitati riguardo agli input dell'utente, perché il gioco di ruolo per sua natura è aperto all'implementazione di dinamiche personalizzate e modifiche eccezionali alle regole, ed essendo lo scopo del progetto sostituire una scheda personaggio cartacea (in cui ovviamente si può scrivere quello che si vuole e come si vuole) si è voluto mantenere questa possibilità, dove necessario e dov'era sensato farlo.

1.1 Die

La classe Die (dado) viene utilizzata per rappresentare le informazioni relative a un dado da tirare: le sue facce. In più fornisce un getter per le facce e il metodo unsigned roll() const che permette di tirare il dado, ritornando un numero casuale da 1 al numero delle facce. Fornisce anche gli operatori di confronto e uguaglianza, perché spesso vengono memorizzati in un vector che li ordina in ordine crescente.

1.2 Character e punteggi di caratteristica

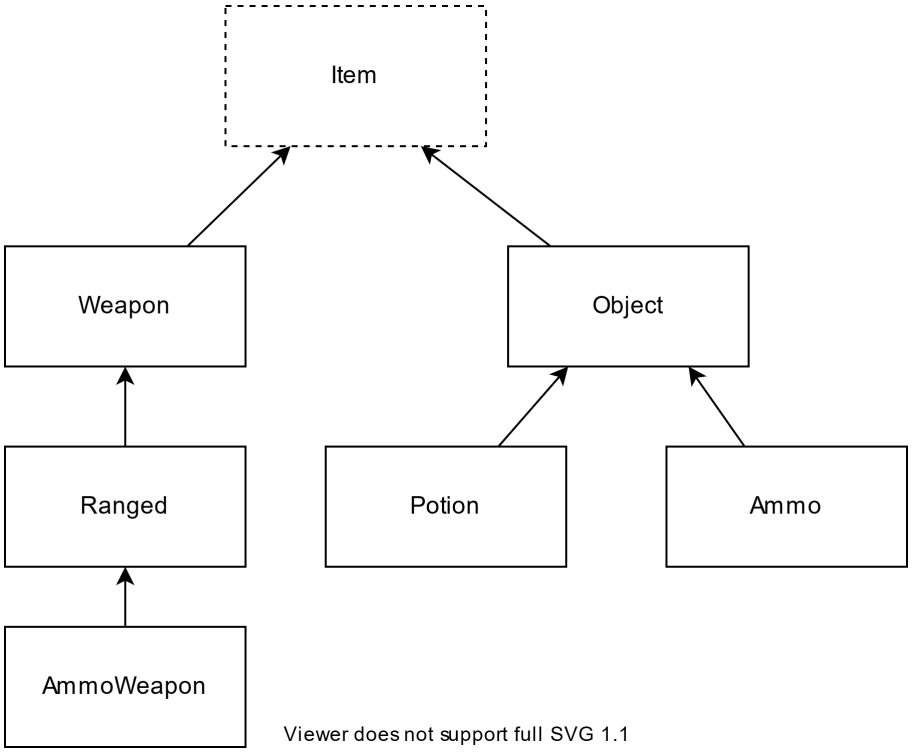
Character è la classe che rappresenta le principali informazioni del personaggio come il suo nome, la sua razza e il suo allineamento, incluse le informazioni relative alla sua salute, alle sue classi e ai suoi punteggi di caratteristica, competenze nei tiri salvezza e abilità. In più rappresenta gli oggetti tenuti in mano attraverso DeepPtr ad Items, descritti nel dettaglio successivamente. In questo caso si è optato per un array statico di due elementi (mano destra e sinistra) da indicizzare con due enum: leftHand e rightHand.
Di fondamentale importanza in tutto il progetto sono i punteggi di caratteristica. Sono 6: Forza, Destrezza, Costituzione, Saggezza, Intelligenza e Carisma e sono rappresentati da un punteggio > 0 (solitamente da 1 a 20). Da questi punteggi vengono calcolati i modificatori che vanno applicati ad ogni tiro su una specifica caratteristica, che possono essere positivi o anche negativi (con un punteggio < 10 il modificatore è un malus). In ogni caratteristica (Ability) il personaggio può essere competente, in tal caso se il tiro sulla caratteristica è un tiro salvezza viene sommato il bonus di competenza, che viene calcolato in base al livello del personaggio (al livello 1 è pari a +2). Ogni caratteristica si può specializzare in abilità più specifiche (Skills), ad esempio alla Destrezza sono associate le abilità Acrobazia, Rapidità di mano e Furtività. Quando il giocatore tira su un'abilità applica il modificatore della caratteristica associata. Per ogni abilità però può anche essere competente, in tal caso aggiunge il suo bonus di competenza; alcune classi forniscono la maestria per alcune abilità, in tal caso si aggiunge due volte il bonus di competenza. Vista la complessità delle Abilità (Skills, diverse dalle Ability che sono i punteggi di caratteristica) è stato deciso di rappresentarle come classe annidata alla classe Character. Le Caratteristiche e le Abilità vengono rappresentate attraverso enum nel file AbilitiesEnums.h. Sono racchiusi nel namespace enums, per non inquinare lo spazio globale dei nomi. In enums sono anche racchiusi due metodi per convertire da enum a string (tradotta per la View) e due costanti che specificano il numero di Caratteristiche e Abilità, che verranno usate soprattutto nella View.
L'altra classe annidata è Class, che rappresenta le classi del personaggio. Una classe è un ambito di specializzazione del personaggio, che gli conferisce particolari abilità (non rappresentate nel progetto). Ogni classe deve rappresentare le informazioni relative al suo nome, il livello attuale, il tipo di dado vita usato (Hit

die) e quanti ne sono stati usati. All'interno di Character le classi sono identificate dal loro nome e la somma dei livelli nelle sue classi determina il livello del personaggio. I dadi vita possono essere utilizzati per ripristinare la salute del personaggio e sono pari al livello di quella classe. Si ripristinano con un riposo (void rest()), che ripristina anche i punti ferita del personaggio al suo massimo.

1.3 Bag

Bag è la classe che rappresenta lo zaino contente l'inventario del giocatore. È composta da un vector (il template C<T> richiesto dal punto 2 della specifica) che contiene DeepPtr di Item (punti 3 e 4 della specifica). Fornisce i metodi per aggiungere o prelevare Items dalla borsa, eseguendo i controlli necessari a garantire che il peso totale trasportato non sfori il massimo.

2 Gerarchia di Item



Oltre alle statistiche del personaggio si è posta una particolare attenzione a come sono rappresentati gli oggetti (Item) e come vengono trasportati e utilizzati.
L'astrazione di base di un oggetto trasportabile nello zaino o che si può prendere in mano è Item, quella che nel modello è la classe base astratta della gerarchia G richiesta nella specifica.
Item rappresenta le seguenti informazioni di un generico oggetto: il suo nome (name) e il suo peso (weight). Mette a disposizione i seguenti metodi virtuali (ad eccezione del getter per il nome: string getName() const): un getter per il peso (double getWeight() const), gli operatori di uguaglianza e disuguaglianza (che eseguono anche un controllo che il riferimento all'item da confrontare abbia lo stesso tipo dinamico dell'item di invocazione), e due metodi virtuali puri: Item* clone() const e int use().
clone() è il classico metodo di clonazione che restituisce, nei suoi override, un puntatore polimorfo ad una copia del (puntatore o reference polimorfi all') item di invocazione, mentre use() ha comportamenti diversi in base al tipo dinamico dell'item di invocazione. Di quest'ultimo metodo verranno descritti successivamente i suoi override, ma l'idea di base è quella di avere un metodo con cui il personaggio possa interagire con l'oggetto.
Gli item istanziabili si dividono in Weapon e Object.

2.1 Weapon

La sotto-gerarchia di **Weapon** rappresenta tutti quegli oggetti che vengono utilizzati come arma e possono provocare del danno diretto, ovvero utilizzando un'arma si attacca un nemico.
Una weapon deve memorizzare le informazioni relative a:
- i dadi da tirare per infliggere danno, rappresentati in vector<Die> dice;

- il tipo di danno inflitto (perforante, tagliente, contundente, da fuoco, da freddo, ecc…): string damageType;
- il bonus dell'arma che viene applicato sia al tiro per colpire che ai danni: int bonus;
- le proprietà di quell'arma, ad esempio se un arma è da lancio può essere lanciata, se due armi sono leggere possono essere usate contemporaneamente (una in una mano, una nell'altra), ma anche abilità particolari che potrebbe avere un'arma. Per semplicità le proprietà vengono solo utilizzate per essere memorizzate e non hanno effettive implicazioni sull'utilizzo di quell'arma, ad esempio non avviene il controllo che due armi siano leggere per essere usate in due mani contemporaneamente. Il controllo viene lasciato all'utente. Le proprietà vengono rappresentate come vector<string> Properties.
Ogni weapon mette a disposizione i getter per ogni suo campo (per valore o per riferimento costante a seconda che il tipo di ritorno sia rispettivamente un tipo primitivo o meno) ed esegue l'override di Weapon* clone() const e degli operatori di uguaglianza, che hanno un comportamento analogo a quelli di Item con la sola aggiunta del controllo che tutti i campi privati di Weapon siano uguali alla weapon su cui fare il confronto. Il modus operandi di override degli operator==/operator!= è sempre lo stesso in tutta la gerarchia di Item, dunque successivamente sarà sottinteso (a meno di eccezioni).
Il metodo int use() in una weapon restituisce la somma di tutti i tiri dei suoi dadi + il bonus dell'arma, ovvero il danno inflitto dall'arma senza contare i modificatori di caratteristica, che verranno aggiunti successivamente. Viene anche eseguito il controllo che il danno non sia negativo, eventualità non ammessa.

Weapon si specializza in **Ranged** nel caso in cui l'arma possa essere utilizzata anche a distanza. Un esempio è il giavellotto, che può essere utilizzato corpo a corpo, ma anche lanciato per colpire un nemico più distante. Di conseguenza aggiunge semplicemente l'informazione sulla distanza massima a cui può essere lanciata l'arma attraverso il campo privato unsigned int range (d'ora in poi il tipo unsigned int verrà sempre chiamato unsigned per praticità, come anche nel codice) e il rispettivo getter.

Ranged si specializza ancora in **AmmoWeapon**, che differisce locicamente sotto alcuni punti da ranged, in quanto un'ammoweapon è un'arma che può colpire solo a distanza (anche ravvicinata), attraverso l'utilizzo di munizioni. Un esempio immediato è l'arco, che utilizza come munizioni le frecce. Ovviamente un'arma a munizioni non può utilizzare qualsiasi tipo di munizioni: un arco non può tirare un dardo da balestra, dunque AmmoWeapon dovrà memorizzare il nome della munizione che utilizza attraverso string ammoUsed e fornire il relativo getter.

A questo punto si può già notare come tutti i punti del requisito 1 della specifica siano rispettati:
a) La gerarchia di Item include Weapon, Ranged e AmmoWeapon => almeno 3 tipi istanziabili;
b) Item è la classe base astratta della gerarchia;
c) Item ha altezza 3 (>= 2);

2.2 Object

Object rappresenta un oggetto generico che può essere trasportato in zaino, ma che non infligge danni direttamente, dunque ben diverso da una Weapon. A questo punto potrebbe essere lecito domandarsi perché una weapon non sia un sottotipo di Object. Il motivo è che un object memorizza in realtà più oggetti identici in un'unica entità, cioè memorizzando le informazioni comuni più la quantità degli oggetti rappresentati. Questo non è sensato farlo con delle armi, essendo che è raro avere nell'inventario due copie esatte della stessa arma e, in ogni caso, non sarebbe particolarmente sensato aggregarle in un'unica entità.
Un Object aggiunge, rispetto ad Item, una descrizione dell'oggetto, che può anche essere utilizzata per specificare proprietà aggiuntive di un oggetto, ad esempio in una corda di canapa si può memorizzare la lunghezza in metri. Questa viene rappresentata semplicemente da string description. La quantità di oggetti rappresentati è memorizzata in unsigned amount. Al solito Object fornisce i getter per i suoi campi privati, oltre all'override di int use(), che consuma uno degli oggetti, riducendo amount e ritorna la nuova quantità di oggetti rappresentati. Object esegue l'override di double getWeight() const restituendo il peso di tutti gli oggetti rappresentati, cioè il prodotto del peso di un singolo oggetto (l'attributo weight ereditato da Item) per la quantità degli oggetti. Inoltre fornisce un setter per la descrizione: void changeDescription(const string&) e i due metodi void add(unsigned amount = 1) e void remove(unsigned amount = 1) che permettono rispettivamente di aggiungere o rimuovere una quantità uguale al parametro formale amount dall'object, con l'eccezione di remove in cui se amount è maggiore della quantità di oggetti effettivi questa diventa 0.
Si noti che gli operatori di uguaglianza e disuguaglianza di Object non tengono conto della quantità di oggetti rappresentati, ma confrontano solamente le informazioni in comune fra gli oggetti.

Object si specializza in due sottotipi distinti: Potion e Ammo.

Potion rappresenta delle pozioni che, rispetto ad Object, forniscono l'informazione relativa ai dadi da tirare per la cura. Il metodo int use() di potion infatti restituisce il risultati di tutti i tiri dei dadi, procedendo poi a consumare la pozione, quindi diminuendo l'amount dell'object. Si noti che ad una cura si aggiunge anche il bonus di costituzione (si aggiunge sempre almeno 1 anche se il bonus è 0 o negativo), informazione che Potion non possiede, che quindi verrà aggiunta in seguito dal Model.

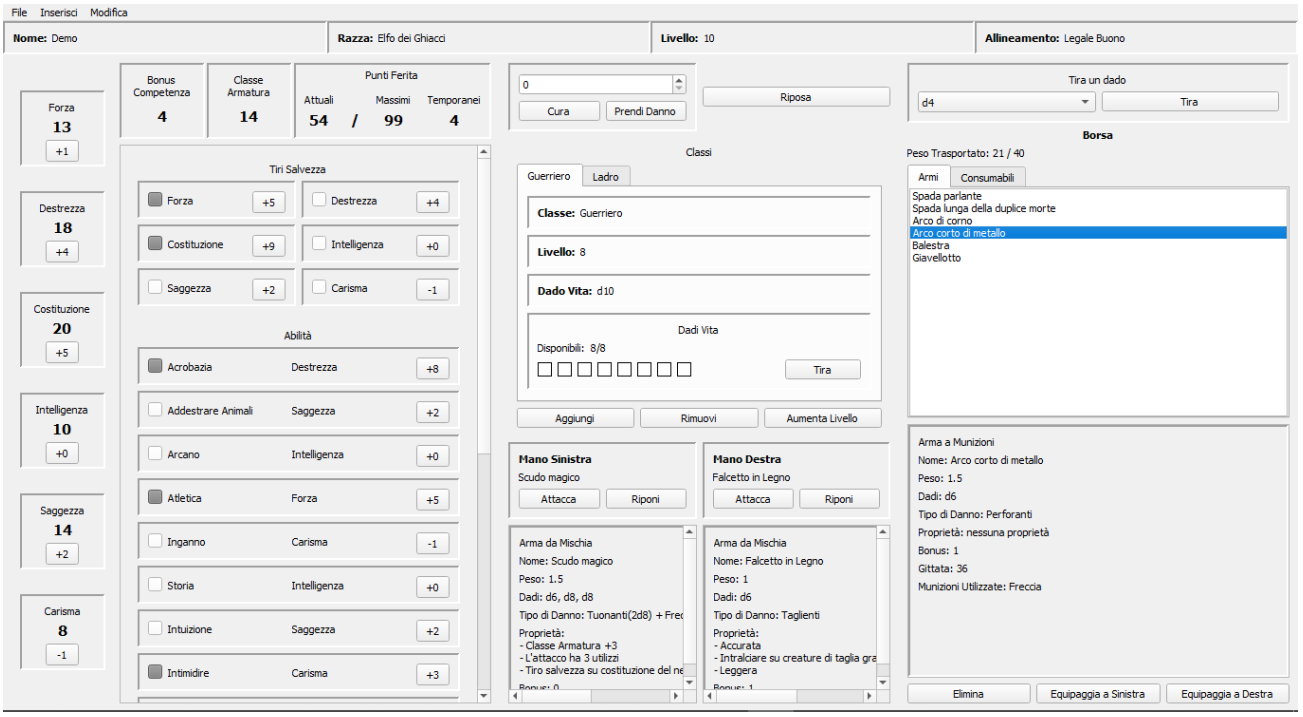
Ammo rappresenta le munizioni per le AmmoWeapon. Sono identificate dal loro nome (come già visto in AmmoWeapon), quindi ammo diverse con lo stesso nome vanno bene per la stessa arma. Oltre all'informazione di essere delle munizioni possono infliggere del danno bonus, dunque aggiungono alle informazioni di Object un booleano per vedere se hanno un danno bonus (bool hasBonusDamage), il tipo di danno inflitto (string damageType) e il dado da tirare per il danno bonus. Non tutte le munizioni infliggono danno bonus, infatti una freccia comune infliggerà il solo danno dell'arco, mentre una freccia infuocata o avvelenata potrebbe infliggere ad esempio un d4 di danni da Fuoco o Veleno, ma è importante che tutte queste possano essere utilizzate dallo stesso arco. L'override di int use() in questo caso si comporta in modo diverso in base al tipo di Ammo: se infligge danno bonus restituisce il risultato del dado, altrimenti si comporta come un Object; in entrambi i casi l'ammo viene consumata. Anche nel caso di Ammo operator==/operator!= hanno un comportamento diverso dal solito: Nel caso in cui la munizione non infligga danno bonus non vengono confrontati il tipo di danno e il dado per il danno (che ci si aspetterebbe che fossero rispettivamente una stringa vuota e un dado a 0 facce, ma non avrebbe importanza in ogni caso). Nel caso in cui invece infligga danno bonus gli operatori confrontano al solto l'uguaglianza di tutte le informazioni rappresentate (stesso tipo di danno e stesso dado).

3 CharacterSheet e resto del modello

A questo punto si può introdurre **CharacterSheet**, l'interfaccia che permette di interagire con il modello nel suo inseme dall'esterno e che mette in comunicazione il personaggio con il suo inventario. In particolare si occupa di equipaggiare gli oggetti dell'inventario in mano al personaggio, o di riporre quelli che ha già in mano di nuovo nella borsa. In più Permette di utilizzare gli oggetti che il personaggio ha in mano. In Character, infatti, è possibile solo avere informazioni sugli oggetti che tiene in mano, ma senza poterci interagire. Essendo CharacterSheet e Character strettamente collegati si è deciso di rendere l'accesso non const alle mani esclusivo a CharacterSheet (con una friend), delegando a CharacterSheet controlli aggiuntivi sulle operazioni da eseguire in base all'Item. In particolare controlla se è un Object, in tal caso lo consuma, oppure nel caso fosse una pozione la utilizza per curare il personaggio.

Il modello fornisce anche la classe NormalThrow, da cui eredita AttackThrow, che servono a memorizzare varie informazioni sui tiri, di Abilità o per gli attacchi, in modo da poter essere facilmente visualizzabili nella View, mostrando i passaggi effettuati per arrivare al risultato finale (che spesso implica diverse somme e molti bonus applicati). AttackThrow è particolarmente complicato, perché deve rappresentare due tiri in uno. Per attaccare infatti prima si esegue il tiro per colpire, il cui risultato decide la riuscita o meno dell'attaco. Come già accennato non è possibile eseguire il controllo della riuscita effettiva dell'attacco, in quanto implica un confronto tra i giocatori (il totale deve essere maggiore o uguale alla Classe Armatura del personaggio da colpire), dunque si procede in ogni caso al tiro per i danni inflitti dall'attacco (il secondo tiro) che, eventualmente, il giocatore scarterà in caso di fallimento dell'attacco. Si noti che, nel caso di un attacco con un'arma a munizioni, la munizione verrà consumata in ogni caso. Questo è un comportamento atteso, in quanto, anche se l'attacco non è andato a buon fine, la munizione è stata tirata comunque e deve di conseguenza essere consumata.

5 Descrizione della GUI



Per l'interfaccia grafica è stato scelto il design pattern Model-View-Controller, rispettivamente caratterizzati dalle classi CharacterSheet, MainWindow e Controller. Come richiesto dalla specifica il modello non fa alcun utilizzo della libreria Qt, mentre la vista fa un utilizzo limitato del modello. La comunicazione tra le due parti è gestita dal Controller, che fa un ampio utilizzo di segnali e slot. L'aderire a questo design pattern ha favorito la separazione tra View e Model, ma ha causato qualche problema nella gestione di alcuni eventi, soprattutto la connect degli slot del controller con alcuni pulsanti che, essendo annidati a diverse classi, hanno richiesto l'utilizzo di approcci che, a posteriori e grazie all'esperienza acquisita attraverso questo progetto, avrebbero potuto essere sostituiti da soluzioni certamente più eleganti.

La GUI è composta da tre sezioni principali: CharacterInfo, MiddleWidget e BagInfo, oltre alle quattro Label in alto che mostrano il nome, la razza, il livello e l'allineamento del personaggio.
Ad una prima impressione l'interfaccia potrebbe risultare molto caotica e piena di informazioni, ma nella pratica avere a disposizione tutti i dettagli del personaggio e della borsa, riducendo la necessità di muoversi tra troppe schede e menù, è in realtà estremamente utile e conveniente.

CharacterInfo è il widget di sinistra e comprende le statistiche relative ai punteggi di caratteristica, i tiri salvezza, le abilità, il bonus di competenza, la classe armatura e la salute del personaggio.
Per ogni tiro, che sia su una caratteristica pura, un tiro salvezza o un'abilità fornisce un pulsante che mostra il bonus totale applicato a quel tiro e che permette di tirare il dado (d20) e mostrarne il risultato.
Nell'area a scorrimento al centro di questo widget è anche possibile modificare la competenza dei tiri salvezza e delle abilità. Il piccolo pulsante a sinistra di ogni tiro salvezza o abilità infatti, se cliccato, modifica la competenza in quel tiro. Nel caso dei tiri salvezza se il pulsante è grigio il personaggio ha la competenza in quel tiro, se è bianco no. Discorso a parte va fatto per le abilità che, avendo anche la maestria, sono più complesse. In questo caso se il pulsante è grigio il personaggio ha la competenza, cliccando un'altra volta sul pulsante questo diventerà nero, indicando che ora il personaggio ha la maestria in quell'abilità. Cliccando di nuovo il pulsante diventa bianco, indicando che non vi è più né competenza né maestria. I bonus applicati al tiro cambieranno di conseguenza, come si può notare dai pulsanti a destra.

MiddleWidget è il widget centrale. Il suo nome poco fantasioso è dato dall'eterogeneità delle informazioni rappresentate al suo interno: in alto una zona che permette di curare o far prendere danno al proprio personaggio e di farlo riposare, in mezzo le informazioni sulle sue classi, all'interno di un widget a tab, che permette di vedere i dadi vita utilizzati e disponibili (attraverso dei quadrati che diventano neri se il dado è

stato già utilizzato) e un pulsante per tirarli. In più sono disponibili i pulsanti per aggiungere una nuova classe, per rimuovere quella attualmente selezionata e per aumentare il livello della classe selezionata.
In basso poi vengono mostrate le informazioni sugli Item che il personaggio ha in mano, oltre ai pulsanti per usare l'oggetto equipaggiato o riporlo nell'inventario.

BagInfo è il widget che mostra le informazioni sulla borsa, con i relativi Items trasportati (oltre al widget per tirare un dado qualsiasi senza applicare bonus che può sempre tornare utile).
La borsa viene rappresentata con due tab contenenti una lista di Items. I tab contengono le due principali gerarchie istanziabili di Item: Weapon e Object. Sono presenti anche i pulsanti per eliminare l'Item selezionato o per equipaggiarlo nella mano destra o sinistra. Se un Item viene equipaggiato in una mano, l'oggetto che prima era in mano viene automaticamente riposto nella borsa (se la mano non era vuota). Ovviamente è disponibile anche una finestra per mostrare i dettagli dell'Item selezionato.

Tutte le azioni che non è possibile eseguire attraverso i pulsanti presenti nell'interfaccia sono nel menù contestuale in alto. In particolare sono presenti i menù per aggiungere degli Item alla Borsa in Inserisci e un menù Modifica da cui si possono modificare le varie statistiche del personaggio e il peso massimo trasportabile dalla borsa.

6 Salvataggi su file

Nel menù File sono presenti le azioni che agiscono sul file attualmente caricato. All'avvio del programma verrà creata una nuova scheda personaggio non ancora salvata su un file. La si può modificare a proprio piacimento e, prima di chiudere il programma, si possono salvare tutte le modifiche effettuate su un file, attraverso File->Salva o File->Salva con Nome. Il file verrà salvato con estensione .character (in ambiente GNU/Linux è necessario scrivere esplicitamente l'estensione, su Windows basta solo il nome del file) nella directory scelta. La cartella che viene mostrata di default è "Saved Character" presente nella root della directory del progetto. Questa cartella contiene già un file di esempio "Esempio.character". Una volta salvato il file può essere caricato attraverso File->Apri.
I file *.character sono dei semplici file di testo che contengono tutte le informazioni sul personaggio salvato. Per semplicità è stato scelto di leggere e scrivere questi file attraverso la libreria di input/output standard, utilizzando i metodi statici privati di Controller string getLine(ifstream&) per leggere un'intera riga del file, restituendone la stringa risultante, Item* getItem(ifstream&) che legge tutte le informazioni di un Item e restituisce il puntatore polimorfo all'oggetto allocato sullo heap e CharacterSheet* loadCharacter(const QString&) che utilizza le prime due funzioni per caricare il file e restituisce la scheda personaggio completa risultante o solleva un'eccezione runtime_errror nel caso qualcosa sia andato storto durante il caricamento. Per quanto riguarda il salvataggio è stata utilizzato il metodo statico privato di Controller void printItemDetails(ofstream, const Item*) per scrivere su file le informazioni riguardanti un Item e il metodo privato saveCharacter() che utilizza anche printItemDetails per scrivere su file tutte le informazioni relative al personaggio attuale.

7 Suddivisione del lavoro progettuale

Il lavoro non è stato da subito diviso in modo ben preciso, soprattuto nella parte del model. Nel fare vector e DeepPtr abbiamo lavorato insieme per tutto il tempo, alternandoci per fare ciascuno circa metà del lavoro, mentre chi non stava effettivamente scrivendo il codice guardava e dava consigli, in questo modo ci siamo sempre confrontati sulle scelte da effettuare. Abbiamo utilizzato questa modalità di collaborazione inizialmente, perché questo era il primo vero progetto di programmazione per entrambi, dunque volevamo affrontarlo insieme confrontandoci man mano per capire come proseguire, anche perché non avremmo nemmeno saputo come spezzare il lavoro. Quando abbiamo cominciato a lavorare sulla gerarchia di Item abbiamo cominciato a dividerci i compiti, io mi sono occupato di Weapon e Francesco di Object. Successivamente abbiamo mantenuto questa divisione per quasi tutto il progetto: io mi sono occupato di Character e delle statistiche, lasciando la gestione dell'inventario a Francesco. Ovviamente in qualche occasione è capitato di scambiarsi i ruoli, o comunque di collaborare per risolvere qualche problema avuto dal compagno, ma in linea di massima la suddivisione è stata questa anche durante tutto il lavoro sulla View. Per Quanto riguarda la fase del lavoro sull'input/output su file io mi sono occupato del input, quindi della lettura dei file di salvataggio.

La collaborazione quasi totale durante lo sviluppo del modello e l'assenza di esperienza nello sviluppo di un progetto di dimensioni significative hanno causato una certa dilatazione dei tempi, così suddivisi (nel mio

caso):
- 3 ore di progettazione preliminare
- 31 ore di sviluppo del modello (compreso debug e testing)
- 19 ore di sviluppo della vista
- 21.5 ore di sviluppo del controller, salvataggi e testing finale

Per un totale di circa 74 ore e mezza. Le 24 ore e mezza in più rispetto alle 50 ore previste, come già detto, sono state causate da una stima errata del tempo che avrebbe richiesto l'idea del progetto e il fatto che la parte del modello sia stata fatta in collaborazione. In particolare è stato sottovalutato lo sviluppo della vista e del controller, essendo che nessuno di noi due aveva mai sviluppato prima un'interfaccia grafica, tanto più complessa come nel nostro caso, quindi non avevamo un'idea chiarissima di quanto tempo avrebbe potuto richiedere. Quando ci siamo resi conto che il progetto stava diventando più grande del previsto non c'era molto da poter tagliare, quindi abbiamo proseguito, consapevoli di aver sforato con i tempi, ma determinati nel portare a termine l'idea originale.

8 Ambiente di sviluppo

Lo sviluppo del progetto è avvenuto sul sistema operativo Microsoft Windows versione 10.0.19042.804, utilizzando la versione 5.15.2 della libreria di Qt, utilizzando la versione 8.1.0 x64 del porting del compilatore GCC per Windows, MinGW, incluso nel pacchetto di installazione di Qt.

9 Istruzioni di compilazione aggiuntive

Nel file .zip consegnato contenente il codice è presente anche il file "progetto.pro", necessario per la compilazione del progetto, essendo diverso da quello ottenibile con il comando "qmake -project". Per eseguire la build del progetto, dunque, basterà eseguire il comando "qmake" per generare il Makefile necessario e successivamente "make". A questo punto, sulla macchina virtuale Linux, make avrà generato l'eseguibile "progetto" nella root della cartella del progetto (nella stessa directory del file .pro, su Windows il Makefile prodotto fa eseguire la build nella sottocartella ./release/), che può essere eseguito con il comando "./progetto".