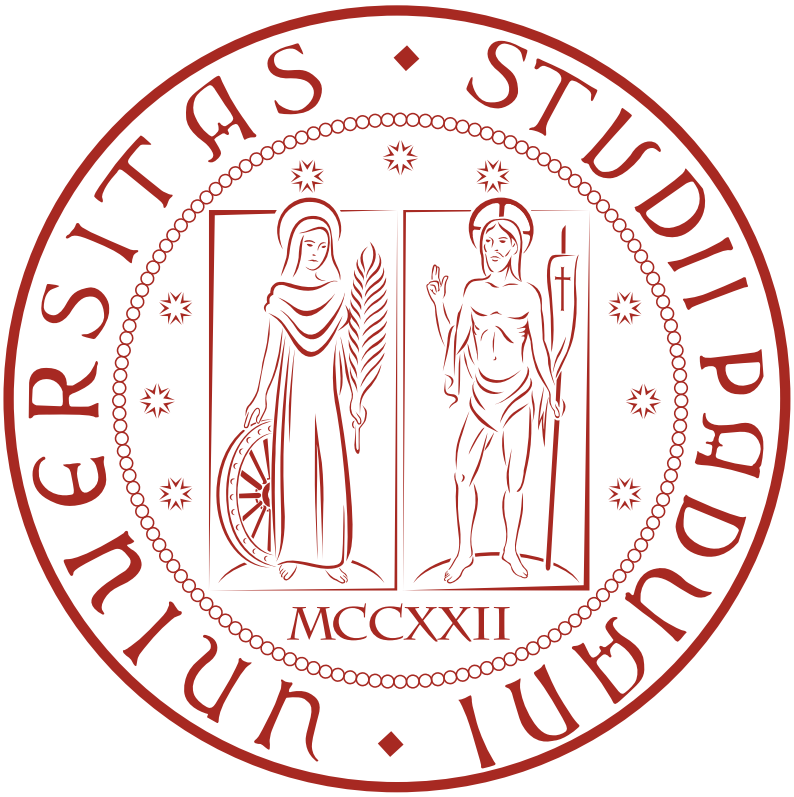

qDB

Relazione del Progetto di Programmazione ad Oggetti

Stefano Munari - 1031243

2 settembre 2014

Indice

1 Introduzione

Il progetto é stato sviluppato con un sistema operativo Linux Ubuntu 12.04. La versione del compilatore GNU utilizzata é la 4.6.3. É stata inoltre utilizzata la versione 2.4.1 dell'IDE QtCreator e la versione 4.8.1 delle librerie Qt.
É stato consegnato anche il file qDB.pro, per generare l'eseguibile dare i comandi qmake e make da terminale.
Tra i file di consegna sono presenti application.qrc e la cartella images, necessari per la compilazione.
Per semplicitá nel seguente documento il riferimento ai metodi sará fatto indicandone solamente il nome(senza parametri e tipo di ritorno) se il contesto risulterá inequivocabile. Quando ci si vorrá riferire ad un metodo di una specifica classe si indicherá come segue: [NomeClasse]NomeMetodo().

2 Descrizione del progetto

L'applicazione consente di gestire un database di personaggi di un role-playing video game(RPG).
Ho reso disponibili le seguenti funzionalitá:

- inserimento
- rimozione
- modifica
- ricerca
- salvataggio dei dati su file .xml
- caricamento dei dati da file .xml

Sono disponibili 3 tipi diversi di personaggi da inserire nel database: Alchemist, Berserker, Golem.
La gerarchia di classi riguardante i personaggi é stata implementata in modo da essere estendibile, é quindi possibile aggiungere alla gerarchia qualsiasi tipologia di nuovo personaggio. Inoltre la gerarchia di per sé offre delle funzionalitá in piú che permettono l'interazione tra gli oggetti attraverso alcuni metodi che sfruttano delle specifiche proprietá di OOP. Questi metodi non sono stati usati concretamente nell'implementazione del database in quanto esulano dallo scopo del progetto(creare un piccolo database).
Ho scelto di mantenerli comunque nel codice in quanto prevedo di sviluppare un ambiente grafico in cui i personaggi interagiscano tra loro.
Ho diviso completamente la parte logica dalla parte grafica per rendere il codice piú mantenibile e modulare possibile.
Ho cercato di usare il pattern Model-View-Controller(MVC) come suggerito a lezione. Per questo ho creato una classe Controller che ha come campo dati un' istanza del contenitore. Il Controller si frappone tra parte logica(Model) e parte grafica(View) manipolando la prima e aggiornando i risultati della seconda in base a ció che restituisce il Model.
Ho inoltre inserito dei file per la gestione delle eccezioni: excView.h, excContainer.h, excHierarchy.h.
Ho reso disponibile un file d'esempio(esempio.xml) contenente dei personaggi giá inseriti.
Ho scelto di usare dei file .xml per il salvataggio e il caricamento dei dati in quanto avevo giá avuto esperienza con questo tipo di file in precedenza e mi sembrava una soluzione semplice da adottare, considerando anche che Qt offre delle classi che ne permettono l'utilizzo.
Questa relazione vuole essere una breve e sintetica spiegazione delle scelte implementative attuate, altri dettagli si possono trovare come commenti direttamente nel codice.

3 Parte Logica

Consiste in una gerarchia di sei classi, di cui tre classi astratte e tre classi concrete. Per poter gestire meglio gli oggetti della gerarchia si é scelto di utilizzare un puntatore alla classe base incapsulandolo come campo dati della classe smartCharacter.

3.1 Character

É stata creata una classe base astratta **character** che rappresenta un personaggio generico, questa classe possiede diversi campi dati propri di un personaggio RPG(name,hp,strength...).
Il file include al suo interno **smartCharacter**. [smartCharacter]setter() é il metodo che si occupa di settare i valori che l'utente inserisce nel momento della modifica del personaggio. Per questo motivo setter() é una funzione amica della classe character e della classe human.
Maxfactor é un campo dati statico che viene usato come fattore moltiplicativo nei semplici algoritmi di calcolo del valore da assegnare ad alcuni campi dati.
DP rappresenta i damage points e sará usato dal metodo attack(), ho scelto di mantenerlo come campo dati di character in quanto caratteristica comune a tutti i personaggi.
Nel calcolo dei vari campi dati si puó inoltre notare come tutti(esclusi ovviamente name e maxfactor) dipendano da LV, cioé dal livello del personaggio. Non é possibile che un personaggio di livello inferiore possieda delle statistiche massime piú alte rispetto ad un personaggio di livello superiore. Riguardo al calcolo delle statistiche e dei punti vita(hp) é possibile, in uno scenario reale di videogame, che i valori diminuiscano durante un combattimento oppure vi siano delle variazioni riguardanti le statistiche.
Per questi motivi SetLV() e setDP() sono dei metodi privati mentre setStat() e setHP() sono pubblici.
Anche setName() é stato dichiarato pubblico, infatti é possibile che l'utente decida di cambiare il nome del personaggio durante il gioco.
isAlive() é un metodo protetto usato dalle classi derivate che ritorna un parametro booleano: true se il personaggio é vivo(HP >0), altrimenti false.
chooseStat() é un altro metodo protetto, serve per scegliere quale statistica modificare da parte di una specifica magia(spell()).
Un esempio del suo utilizzo si ha nella seguente sequenza di chiamate a funzione:
[berserker]trance()->[human]spell()->[human:overriding]chooseStat()->[character]chooseStat().
Character é astratta perché non puó essere costruita se non dalle classi derivate, in quanto il costruttore é protetto e contiene 2 metodi virtuali puri: attack() e clone(). Questi metodi servono rispettivamente per l'attacco e la costruzione di copia, utilizzano puntatori polimorfi come parametri formali(nel caso di attack()) e come parametro di ritorno.
In entrambi i casi si ha covarianza sul tipo di ritorno.
Il distruttore della classe base astratta é stato dichiarato virtuale in modo tale che le classi derivate ereditino il metodo di distruzione virtuale.

3.2 Human

Classe astratta derivata da **character**. Viene aggiunto il metodo spell() che rappresenta un attacco magico generico. Aggiunti anche i campi MP e wisdom, il primo permette ad un personaggio di usare spell() mentre il secondo serve come statistica nel calcolo dei danni di una specifica magia(vedere ad esempio [berserker]iperfury()). La classe é astratta in quanto non implementa i metodi virtuali puri di character ma li eredita soltanto. La classe fa overriding sui metodi setStat() e chooseStat() ereditati da character. Per semplicitá si é scelto di stampare a video le eccezioni della gerarchia, infatti quest'ultime riguardano dei metodi che non sono utilizzati nel programma con cui l'utente interagisce.

3.3 Demon

Classe astratta derivata da **character**. Viene aggiunto il metodo reborn() che controlla se il personaggio é stato ucciso con un solo colpo verificando il valore del campo dati booleano oneHit. Per lo stesso motivo di **human** anche **demon** é una classe astratta. Fa overriding su setHP() ereditato da character.

3.4 Alchemist

Classe concreta derivata da **human**. Vengono implementati i metodi virtuali puri attack() e clone(). Aggiunti due attacchi magici sulfur() e cure(). Si intendono attacchi magici tutti i metodi che utilizzano spell() per il calcolo degli

MP disponibili per l'attacco e wisdom per il calcolo dei punti danno magici. cure() é un attacco magico curativo, quindi i punti danno sono negativi perché hanno effetto contrario rispetto ad un attacco offensivo.

3.5 Berserker

Classe concreta derivata da **human**. Simile ad alchemist tranne per il fatto che implementa attacchi magici differenti.

3.6 Golem

Classe concreta derivata da **demon**.

4 Contenitore

Come contenitore ho utilizzato una lista che sfrutta la tecnica della memoria condivisa attraverso puntatori smart. Ció avviene utilizzando un campo dati che conta i riferimenti (intesi come puntatori) a un dato oggetto che puó esser condiviso tra piú liste.
Poiché risulta necessario sapere se la lista su cui si sta operando é condivisa con altre la complessitá di un operazione di push_back() risulterá O(n).
Di seguito elenco le funzionalitá implementate con i relativi metodi:

- inserimento: push_front(), push_back(), insert_after()
- rimozione: pop_front(), pop_back(), erase_after(), clear()
- modifica: replace(), metodo reso disponibile ma non utilizzato.
- ricerca: find()

Si é scelto di stampare a video le eccezioni di Container, considerandole come delle informazioni che aiutano il programmatore durante la fase di testing. Quest'ultime non si possono verificare durante l'esecuzione del programma (ad esempio non puó esistere una pop_back() su un contenitore vuoto) e non danno informazioni utili all'utente.
Sono stati utilizzati i funtori(definiti nel file functor.h) all'interno del metodo find() per avere una ricerca piú specifica sui singoli campi dati.

4.1 Functor.h

Il file contiene una gerarchia formata da: template di classe base astratta, classi funtore concrete derivate dal template base. Le classi derivate servono a specificare il campo dati che si intende confrontare durante l'esecuzione di find(). Usando il dynamic binding sul parametro di find() riesco ad invocare sempre il funtore derivato che mi serve per eseguire il confronto.

5 Parte Grafica

5.1 MainWindow

Finestra principale che contiene tutti gli altri widget. Tutte le funzionalitá finora descritte (inserimento, modifica, ricerca...) vengono attivate dall'utente interagendo con campi dati propri di questa classe. Gestisce il salvataggio e l'apertura di file .xml. Contiene il campo dati controller. I campi modified, modifiyng e fileopen servono rispettivamente a: sapere se ci sono state delle modifiche non ancora salvate, sapere se c'é una modifica in corso, conoscere il nome del file .xml aperto. In alcune situazioni che potrebbero rivelarsi critiche (ad esempio la modifica di un elemento) disabilito alcune funzionalitá finché le precedenti operazioni non sono state completate.

5.2 Add

É una dialog che permette di scegliere il tipo di personaggio da inserire.

5.3 Form

É una dialog che permette di inserire un personaggio derivato da demon. Nel caso specifico l'unico personaggio derivato da demon é golem.

5.4 Humanform

Eredita da Form in quanto le due dialog sono molto simili. Ho pensato di usare l'ereditarietá visto che Humanform estende Form aggiungendo dei campi dati e dei metodi in piú oltre a utilizzare quelli messi a disposizione dalla superclasse. Permette di inserire personaggi derivati da human.

5.5 Tab

Eredita da TabWidget. É il widget che contiene Table e Searchtable.

5.6 Table

Eredita da TableWidget. Rappresenta tutti gli elementi presenti nel database su una tabella. Ho inserito un campo dati puntatore a controller per avere un accesso facile e semplice al campo dati controller. Nel metodo setXML() scrivo sul file .xml da salvare solamente i valori dei campi dati che sono necessari per ricaricare il personaggio nel database(hp non viene scritto perché sará ricalcolato ogni volta). ModifiedRow tiene traccia della riga che si sta modificando in tabella. Per modificare un elemento é necessario selezionare la riga cliccando su di essa.

5.7 Searchtable

Eredita da TableWidget. Rappresenta i risultati di find() su tabella. Quando viene invocato [searchtable]setRow() TableRow viene condivisa tra SearchTable::searchlist e Table::datalist attraverso l'uso di puntatori(sono entrambe liste di puntatori a TableRow). Per questo motivo quando distruggo Searchtable dealloco solamente i puntatori da searchlist e non gli oggetti puntati.

5.8 Tablerow

É la riga di una tabella. Ogni riga contiene tutti i valori dei campi dati che interessano di uno specifico personaggio. Questi sono gestiti attraverso un array dinamico. L'inserimento dei valori(che avviene in form) cosí come la loro modifica sono ristretti a un preciso range controllato attraverso l'uso di espressioni regolari.
Il simbolo \ indica un campo dati vuoto e quindi non esistente. Con il metodo enableValues() ho permesso la modifica di quasi tutti i campi dati escludendo:
-HP: perché esiste giá un algoritmo che lo calcola in base ad altri parametri.
-Type: perché non é stata ritenuta una modifica sensata.

6 Gestione della memoria

Ho usato la tecnica della memoria condivisa all'interno di Container. Riguardo la parte grafica ho utilizzato il sistema di gestione automatica della memoria di Qt. Non é sempre stato necessario ridefinire i distruttori in quanto, quando viene definito un widget come padre, alla distruzione del padre vengono automaticamente distrutti anche tutti i suoi figli.