Relazione di progetto
Programmazione ad Oggetti
A.A. 2018/2019

Enrico Cancelli 1143080

February 1, 2019

Contents

1 Introduzione

Il software realizzato consente la gestione di un contenitore di RPGItem. Un RPGItem rappresenta un oggetto nell'inventario di un ipotetico gioco di ruolo. La realizzazione è suddivisa in 3 parti:

- Realizzazione delle strutture di supporto(classi Container<T>, RPGContainer e DeepPtr<T>)
- Realizzazione della gerarchia di oggetti RPGItem
- Realizzazione della GUI e processi di serializzazione e de-serializzazione

2 Strutture di Supporto

2.1 Classe Container<T>

La classe *Container<T>* è una struttura dati lineare del tutto simile a *std::vector<T>* ed è stata realizzata in modo da rappresentarne una versione essenziale. Essa mette a disposizione dell'utente metodi pubblici con firme molto simili a quelle della classe da cui prende ispirazione, mantenedo invariate pre e post-condizioni e complessità computazionale in termini di spazio e tempo. La memoria è gestita tramite un'array dinamico allocato nello heap di capacità k con $k = 2^{\lceil \log_2 size() \rceil}$ dove size() è il numero di elementi inizializzati nell'array. Nella pratica ogni volta che la size supererà la capacità verrà riallocato un'array di dimensioni doppie rispetto all'originale e con una coppia degli elementi del vecchio array nella stessa posizione. Ciò consente di avere costo ammortizzato $O(k)$ per l'inserimento in coda. L'inserimento in qualsiasi altra posizione dell'array invece avra costo lineare agli elementi successivi alla posizione di inserimento a causa della traslazione necessaria. Ciò è valido anche per l'eliminazione che richiederà che l'array venga ricompattato. Oltre alle funzionalità di inserimento e cancellazione e ai metodi size() e capacity(), è possibile accedere e modificare gli elementi del container utilizzando le classi interne *Container<T>::const_iterator* e *Container<T>::iterator*. Queste due classi rappresentano iteratori ad accesso randomico(è pertanto disponibile l'overloading dell'operatore di subscripting). Essi aderiscono rispettivamente alle interfacce *LegacyRandomAccessIterator* e *LegacyInputIterator* definita nella libreria standard e pertanto mettono a disposizione operatori di confronto, somma, differenza e una serie di tipi richiesti dal template di struttura *std::iterator_traits*:

- value_type: tipo dell'oggetto referito dell'iteratore
- reference: tipo del riferimento
- pointer:tipo del puntatore

- difference_type: tipo di ritorno per operazioni di differenza tra iteratori
- iterator_category: tag che identifica la natura dell'iteratore

L'adesione a tali interfacce consente l'utilizzo di tutti (o parte nel caso di InputIterator) gli algoritmi generici presenti nell'header file *<algorithm>*. Questi ultimi non necessitano di alcun riferimento al tipo di struttura dati su cui operare ma soltanto uno o più iteratori che puntino a un elemento del contenitore e spesso anche di un funtore (o oggetto std::function o lambda-espressione o puntatore a funzione). L'adesione a tali interfacce è stata sfruttata in particolare per verificare la presenza di un'oggetto con determinate caratteristiche all'interno del container tramite la funzione generica *std::find_if*. Il metodo *sort* in particolare utilizza iterativamente questa funzione per riempire un nuovo contenitore (che verrà successivamente ritornato) con copie degli oggetti che soddisfano il predicato passato come parametro.

2.2 Classe DeepPtr<T>

Essendo il comportamento del template di classe Container indipendente dal tipo di dato in esso contenuto, la gestione della memoria profonda (cancellazione della memoria allocata nello heap, creazione di copie profonde e clonazione) deve essere demandata ad un oggetto intermedio che funga da wrapper e consenta allo stesso tempo di gestire il polimorfismo. DeepPtr è un template di classe che, incapsulando un puntatore alla classe base RPGItem, soddisfa queste esigenze. Per rendere il comportamento di tale classe trasparente e quindi del tutto simile ad un normale puntatore, sono stati ridefiniti gli operatori * e → in modo opportuno.

2.3 Classe RPGContainer

Questa classe è una specializzazione dell'istanza specifica:
Container<DeepPtr<RPGItem>>. All'interno di essa sono definiti, oltre agli opportuni costruttori, dei metodi specifici di ricerca per elementi di tipo RPGItem all'interno del contenitore. Essi non sono altro che specializzazioni del metodo *sort* generico presente in Container e implementano ricerche specifiche basate su caratteristiche proprie della classe RPGItem. In particolare il metodo *searchByWildCardName(std::string s)* utilizza il tipo *std::regex* messo a disposizione dalla libreria standard per cercare tutti gli RPGItem con nome che inizia con s (la valutazione non è case sensitive). Il metodo *searchAllByType* invece cerca tutti gli elementi appartenenti ad un sottoinsieme (non necessariamente proprio) di categorie di RPGItem (tale sottoinsieme è mappato dai valori booleani passati come parametro).

3 Gerarchia di Tipi RPGItem

3.1 Classe base astratta RPGItem

Essa rappresenta genericamente un oggetto di inventario di un gioco di ruolo caratterizzato da un nome, una descrizione e un'indicatore booleano di unicità. Questa classe inoltre mette a disposizione 3 metodi virtuali puri: *clone*,*getCategory* e *getPrice*. Il metodo *getCategory* ritorna la categoria di appartenenza dell'oggetto di invocazione. E' stato deciso di implementare tale caratteristica come metodo virtuale puro per due ragioni: evitare catene di dynamic_cast per riconoscere il tipo dinamico di un'oggetto e slegare il concetto di categoria dal concetto di classe lasciando la scelta all'utente programmatore della libreria se fare in modo che le due denominazioni corrispondano o meno. Nelle 3 classi che derivano da RPGItem, l'overriding di tale metodo è marcato *final* in modo da evitare che una qualsiasi sottoclasse definisca una categoria propria. Teoricamente un'ulteriore classe derivata da RPGItem potrebbe tuttavia decidere di lasciare la scelta a classi più in profondità nella gerarchia. Il metodo *getPrice* invece ritorna il prezzo di un determinato oggetto.

3.2 Classe RPGWeapon

Rappresenta un'arma di un gioco di ruolo. E' caratterizzata da un valore base per il danno, un livello e dall'essere brandita con una o due mani e campi dati statici costanti come *unique_boost* e *increase_per_level* che rappresentano rispettivamente l'aumento di prezzo causato dall'unicità dell'item in questione e il coefficiente lineare con cui aumenta il prezzo e il danno in base al livello. I metodi degni di nota sono il metodo *damage* che ritorna il danno totale calcolato utilizzando la seguente formula:

$$danno\_base + increase\_per\_level * livello \tag{1}$$

(*2 se brandita a due mani)

e l'overriding del metodo *getPrice* che definisce il prezzo di un oggetto arma usando la formula

$$livello * increase\_per\_level \tag{2}$$

(+ unique_boost se l'arma è unica)

.

3.3 Classe RPGArmor

Rappresenta un'armatura o un oggetto difensivo. E' caratterizzato da un livello e dal materiale con cui è stata costruita. Tale caratteristica è im-

plementata utilizzando un tipo enum pubblico dichiarato all'interno della classe chiamato *armorclass*. I materiali disponibili sono i seguenti:

- Legno
- Bronzo
- Ferro
- Acciaio
- Mithril

E' fornito un metodo *defence* che calcola la difesa complessiva fornita dall'oggetto calcolandola nel seguente modo:

$$indice\_materiale * increase\_per\_level * livello \tag{3}$$

L'overriding di *getPrice* calcola il prezzo facendo una semplice somma tra livello, indice del materiale e unique_boost (se l'armatura è unica). Altri metodi degni di nota sono *FromInt* e *FromString* che fungono da convertitori da int e string a *armorclass*.

3.4 Classe RPGConsumable

Rappresenta un consumabile (come pozioni, veleni e cibi) ed è l'oggetto più semplice della gerarchia. Nessun consumabile può essere unico e ognuno di essi è caratterizzato da un costo di base e dal fatto che abbia un effetto positivo o negativo. Il costo è uguale al costo base ed è triplicato in caso abbia un effetto positivo.

4 Realizzazione della GUI

La GUI è stata realizzata utilizzando il design pattern Model/View di Qt e facendo quindi uso delle classi messe a disposizione dal framework per la realizzazione del Modello (*QAbstractListModel*) e della View (*QListView*).

4.1 Serializzazione/Deserializzazione

Per le funzionalità di salvataggio e caricamento da file è stato deciso di utilizzare il formato JSON. Tale decisione è dovuto in gran parte dal fatto che tale formato oltre ad essere human-readable (e quindi modificabile manualmente anche senza l'ausilio di strumenti particolari) è molto leggero e poco verboso (specie se comparato ad XML). Tuttavia, a differenza di XML, non esiste nessun modo di stabilire durante il processo di deserializzazione se il documento in lettura sia valido oltre che ben formato (ciò è possibile in XML tramite la definizione di una precisa grammatica in un apposito file DTD).

Perciò è necessario definire dei controlli a livello di interfaccia subito dopo la deserializzazione del file per assicurarsi che siano presenti i giusti campi con il giusto tipo. Un file leggibile da Qontainer si presenta come un array JSON di oggetti con un campo category che definisce il tipo e tutti i campi necessari per la costruzione di quella specifica istanza di quello specifico tipo.

4.2 Chiamate polimorfe nella GUI

Le chiamate polimorfe ai metodi virtuali del modello sono utilizzati dalla GUI per capire quale specifiche informazioni mostrare a schermo. In ogni momento l'interfaccia mostra a schermo tutte le informazioni disponibili e calcolabili per l'oggetto attualmente selezionato (tralasciando ovviamente quelle non informative ed utilizzate per puri scopi implementativi) facendo uso di *getCategory* per capire la struttura dei dati informativi che gli vengono forniti e di *getPrice* per il calcolo del prezzo.

5 Istruzioni per l'uso e compilazione

Il progetto è organizzato in cartelle secondo una struttura gerarchica e fa uso di file per risorse esterne di Qt. Per la compilazione è perciò necessario utilizzare il file .pro fornito (senza eseguire il comando qmake –project). E' inoltre fornito un file rpg_items.json nella cartella Resources contenente oggetti di esempio caricabili attraverso l'opportuna funzionalità "salva" fornita dall'interfaccia.

6 Elenco delle ore effettive

- analisi preliminare del problema: 5h
- progettazione modello e GUI: 8h
- apprendimento libreria Qt: 6h
- codifica modello e GUI: 19h
- debugging: 5h
- testing: 5h

Nota: il tempo di apprendimento della libreria è basso a causa di conoscenze pregresse del framework.