Alessandro Chimetto Matr.: 1142192

Qontainer

Abstract
Relazione del progetto individuale del corso di Programmazione ad Oggetti tenuto dal prof. Francesco Ranzato. Si è scelto di modellare la creazione di un software contenitore per articoli in vendita in un negozio di elettronica (es. Mediaword, Unieuro).

Istruzioni
Compilazione
La gestione delle dipendenze è totalmente affidata al file *Qontainer.pro* perciò per la compilazione sarà effettuata semplicemente tramite *qmake Qontainer.pro* => *make*.

Esecuzione


All'avvio si aprirà la finestra principale visibile a fianco ma senza nessuna entry. Nella barra degli strumenti sono presenti i seguenti pulsanti:
- *Salva*: salva il Container corrente se esiste altrimenti richiama *Salva con Nome.*
- *Salva con Nome*: apre una finestra in cui richiede dove salvare il Container corrente e lo salva in formato YAML.
- *Apri*: apre un Container in formato YAML. Un file *example1.yaml* è presente all'interno della cartella radice del progetto.
- *Ricerca*: apre un form di ricerca dal quale si possono fare ricerche fra gli articoli utilizzando i parametri mostrati; il pulsante *Cerca* fa partire la ricerca e restituisce il risultato sulla finestra principale. Il pulsante *Rimuovi Elementi Trovati* cancellerà dal container gli elementi risultanti dalla ricerca. La chiusura di questa finestra comporta l'annullamento della ricerca e il ritorno della finestra principale allo stato iniziale.
- *Nuovo Articolo*: verrà aperta una piccola form in cui verrà chiesto che tipo di articolo inserire e successivamente una Form di Inserimento. Una volta inseriti i dati e premuto il pulsante *Inserisci* la form rimarrà aperta divenendo una Form di Visualizzazione/Modifica.

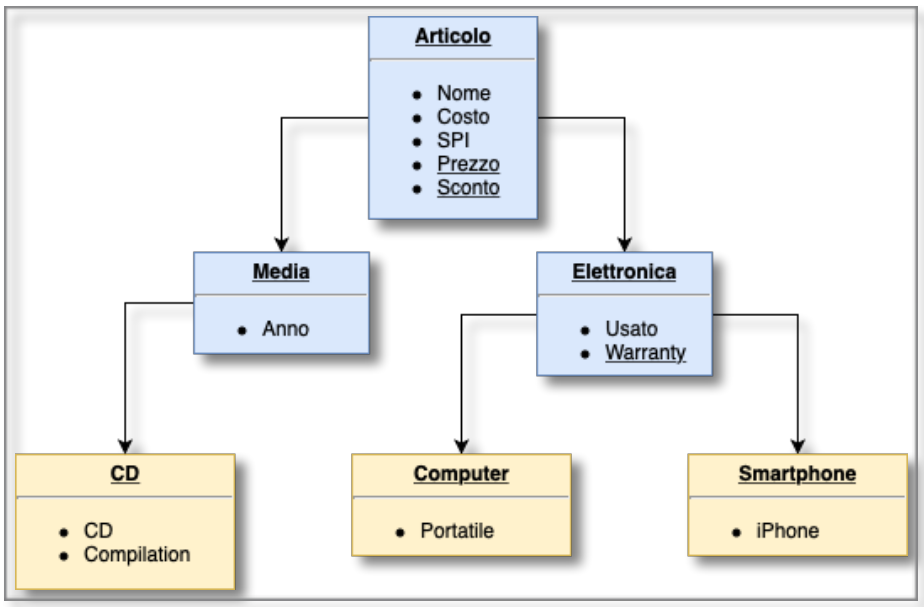
Facendo doppio click su una riga della tabella verrà aperta una Form di Visualizzazione/Modifica che permetterà di vedere in dettaglio i campi dati specifici dell'articolo in quella riga di tabella. È possibile modificare i dati e applicare le modifiche con il pulsante *Applica Mod* oppure eliminare l'articolo dal Container con il pulsante *Elimina*.

Ambiente di Sviluppo
Lo sviluppo è stato effettuato su macOS 10.14.5 Mojave con Emacs 26.2, clang-1001.0.46.4 (fornito con Apple Xcode), QtCreator 4.9.1 e Qt SDK 5.5.1 mentre il testing è stato effettuato sulla macchina virtuale non modificata fornita durante il corso.
La Gerarchia dei Tipi e il Container sono stati creati solamente utilizzando Emacs e clang in modo da garantire il loro funzionamento a prescindere dall'utilizzo di una specifica libreria grafica.

Internals
Qontainer è stato sviluppato utilizzando il design pattern Model-View-Controller in modo da garantire totale separazione fra logica e GUI. La View è totalmente slegata dal Model e tutte le comunicazioni passano per il Controller. I file riguardanti Model si trovano nella cartella *Model* nella radice della cartella di progetto e lo stesso vale rispettivamente per View e Controller.

Model
Il *Model* (file Model/Model.hpp e Model/Model.cpp) è una classe derivata da *QObject* che funge da interfaccia al Container (operando per esempio tutte le conversioni dai tipi Qt a STL e viceversa), si occupa della ricerca di elementi all'interno di esso e del suo salvataggio/caricamento in file YAML.

Gerarchia dei Tipi
Interfaccia e implementazione si trovano nei file Model/Gerarchia.hpp e Model/Gerarchia.cpp ed è rappresentata dallo schema nell'immagine.



In blu sono segnate le classi base astratte mentre in giallo abbiamo tipi istanziabili. Nell'elenco puntato all'interno di ogni tipo sono segnati i campi dati dei rispettivi oggetti accessibili con metodi pubblici del tipo *is[NomeCampo]();* se booleani e *get[NomeCampo]();* in tutti gli altri casi. I campi sottolineati sono in realtà dei metodi virtuali puri in quanto si ritiene che quelle informazioni debbano essere calcolate in base al tipo considerato; più precisamente, ogni classe istanziabile ha delle regole specifiche su come calcolare il proprio prezzo in base al proprio costo, così come ogni classe istanziabile ha delle regole specifiche su come calcolare il proprio sconto in base a SPI (che sta per "Settimane dalla Prima Inscaffalatura") e così come ogni classe istanziabile derivata da *Elettronica* ha delle specifiche regole per calcolare il numero di anni di garanzia.

Questa organizzazione gerarchica dei tipi garantisce una buona estensibilità del codice in quanto nella realtà che si sta modellando è conveniente avere classi organizzate per categorie. Queste categorie sono qui rappresentate dalle classi *Media* ed *Elettronica* ovvero classi che rendono più specifico il concetto di articolo pur non essendo istanziabili e possono quindi enunciare caratteristiche comuni a tutte le classi derivate e istanziabili. Alcune di queste caratteristiche comuni potrebbero necessitare dei calcoli specifici nelle classi derivate e in questo caso si dichiareranno dei metodi virtuali puri.
Per esempio: dalla classe *Articolo* si deriva una classe astratta *Elettrodomestico* con campo RisparmioEnergetico e da quest'ultima si derivano le classi concrete *Lavatrice* e *Aspirapolvere*. La valutazione di RisparmioEnergetico dipende da caratteristiche intrinseche (ovvero dai campi dati) della classe che richiama il metodo *getRisparmioEnergetico()* e quindi converrà dichiarare tale metodo come virtuale puro nella classe *Elettrodomestico* per poi fare overriding. Ciò mette anche a disposizione delle chiamate polimorfe avendo un puntatore ad una classe astratta di categoria dando al codice ulteriore flessibilità nelle possibilità di estensione.

La gerarchia è utilizzabile indipendentemente da qualsiasi libreria grafica.

Container<T>
Data la realtà da modellare precedentemente descritta, si è ritenuto opportuno l'uso di un Container di tipo tabella hash. L'implementazione presente in questo progetto (file Model/Container.hpp) è una versione semplificata e "statica" del concetto standard di tabella hash che prevede:
- tabella di lunghezza fissa;
- gestione delle collisioni con lista di trabocco;
- ricerca in tempo costante mediante metodo della divisione.

In Container<T> ogni tipo T inserito al suo interno è accoppiato ad una chiave univoca. Questa chiave di default è di tipo *std::string* ma può essere un qualsiasi tipo K utilizzando la notazione *Container<T, K>*.

Container<T> mette inoltre a disposizione un iteratore istanziabile con il metodo begin() e dotato della ridefinizione degli operatori di incremento al fine di poter scorrere in modo sequenziale tutti gli elementi presenti nella tabella al momento della creazione dell'iteratore.

Container <T> è utilizzabile indipendentemente da qualsiasi libreria grafica.

Formato di salvataggio
Si è scelto il formato YAML come formato di salvataggio del Container. Questo perché, per la realtà considerata, non si necessità di categorizzare e immagazzinare dati in modo complesso quindi l'eccessiva

quantità di regole necessarie per creare file XML o JSON conformi ai rispettivi standard avrebbe reso inutilmente complicata la programmazione e dilatato i tempi in modo eccessivo.

YAML, al contrario, ha poche ma essenziali regole per la sua costruzione il che lo rende incredibilmente semplice da costruire; inoltre, vista la sua essenzialità, è incredibilmente semplice da leggere una volta aperto con un editor di testo.

Per la lettura e il salvataggio di file in formato YAML si è utilizzata la libreria yaml-cpp che, come suggerito dal nome, è perfettamente integrata nell'ambiente C++ e permette operazioni su file utilizzando sintassi familiare; in particolare risulta essere semplice e molto espressivo l'uso dell'overloading di *operator<<*.

DeepPtr<T>
Per facilitare il trasporto di puntatori all'interno dei vari oggetti dell'applicativo è stato implementato DeepPtr<T> (file Model/DeepPtr.hpp) ovvero un wrapper attorno ad un *T** che ne garantisce costruzione di copia profonda, assegnazione profonda e distruzione profonda. Per assottigliare il divario in termini sintattici e semantici fra *T** e DeepPtr<T>, quest'ultimo permette la costruzione a partire da *const T** e *const T&* e ridefinisce *operator**, *operator->* e *operator&*.

Controller/View
Ogni View ha un suo controller che si occupa della comunicazione con il Model e con altri Controller (quindi di conseguenza con altre View). Al fine di implementale il design pattern MVC sono state create le classi base astratte *Controller* (file Controller/Controller.hpp e .cpp) e *View* (file View/View.hpp e .cpp), derivate rispettivamente da *QObject* e *QWidget*, utilizzate per derivare classi concrete che saranno effettivamente le viste e i rispettivi controllori.
Da View derivano: MainView (finestra principale), CDView (finestra Visualizzazione/Modifica CD), ComputerView (finestra Visualizzazione/Modifica Computer), SmartphoneView (finestra Visualizzazione/Modifica Smartphone), SearchView (form di ricerca).
Da Controller derivano: MainController, CDController, CompurerController, SmartphoneController che sono i controllori associati alle rispettive viste appena elencate.

Dal punto di vista implementativo si potrebbe dire che "comanda la View" ovvero, dovendo visualizzare una nuova finestra, verrà sempre creata per prima la View e poi con il metodo statico *View::initialize(View*, Model*)* le verrà assegnato il suo Controller. Inoltre, è possibile utilizzare la direttiva *#include* solo dalla View al suo Controller e non viceversa: le View hanno infatti bisogno di conoscere appieno l'interfaccia del proprio Controller per poter operare (soprattutto a causa del metodo *makeController* che verrà spiegato più avanti). Includere la View nel rispettivo Controller a questo punto porterebbe ad una condizione di inclusione incrociata che non permette la compilazione; teoricamente si potrebbero evitare problemi di inclusione incrociata utilizzando il forwarding ma in questo caso non è possibile dato che il forwarding non dà informazioni sulla gerarchia dei tipi rendendo la cosa inutile. Il problema viene risolto mediante l'utilizzo del sistema di messaggistica interno alla libreria Qt detto *Signal*/*Slot* che funziona fra segnali e slot (o segnali e segnali) appartenenti a qualsiasi tipo di dato *QObject* o un suo derivato indipendentemente dall'ereditarietà.

Chiamate Polimorfe
Vedremo ora delle chiamate polimorfe significative all'interno di Qontainer:

- *getPrezzo()* è un metodo virtuale puro di Articolo viene richiamato da un Articolo* in Model.cpp:74 per fare un confronto fra i prezzi di tutti gli articoli e in MainView.cpp:83 per popolare la tabella nella schermata principale indipendentemente da quale sia lo specifico derivato di Articolo che si sta rappresentando.
- *makeController()* è un metodo virtuale puro privato della classe View che viene richiamato dal metodo statico *View::initialize(View *v, Model *m)* su *v* per creare il Controller, connetterlo al Model *m* e associarlo alla vista *v*. Il Controller creato sarà sempre quello corretto da associare alla propria View dato che con il late binding verrà richiamato il metodo basato sul tipo dinamico di *v*.

Rendiconto Ore
Il progetto ha richiesto in totale 53 ore di lavoro distribuite secondo la tabella che segue. Nelle 3 ore in più rispetto alle 50 previste sono stati condotti testing e debugging finali a progetto concluso prima della consegna.

Come si osserva dalla tabella ciò che ha richiesto più tempo sono stati la scelta di un problema da modellare comprensiva di analisi preliminare e l'implementazione della GUI vista la complessità architetturale della stessa.

MOTIVAZIONE	NUMERO ORE
Analisi preliminare del problema	8
Ricerca informazioni	5,5
Progettazione Modello	4
Progettazione GUI	5
Implementazione Modello	7
Implementazione GUI	10,5
Debugging	8
Testing	5
ORE TOTALI	53