Corso di Programmazione ad Oggetti

Progetto dell' A.A. 2018/19

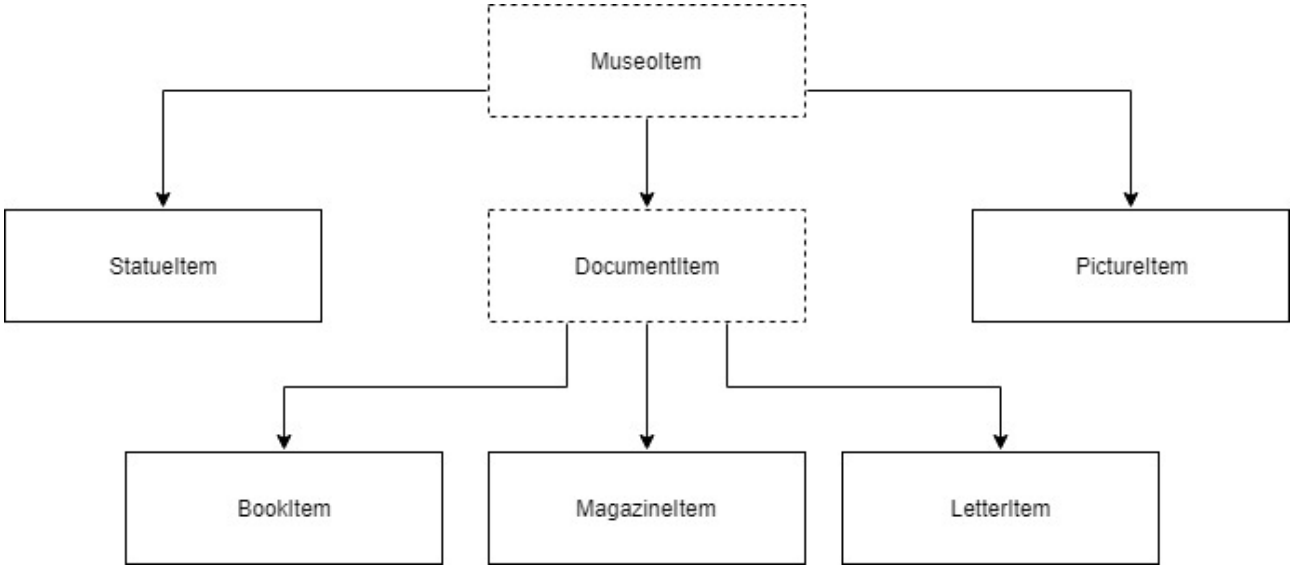
MuseoManager – Gestore archivio museo

Tassetto Riccardo – matr. 1145883

Abstract

Lo scopo del progetto è di creare un'applicazione, **MuseoManager**, la cui funzione principale è la gestione di un archivio di un museo.
Le opere contenute nel museo sono sculture, pitture, libri, lettere e riviste (d'epoca). La gerarchia è facilmente estensibile.
Le funzionalità disponibili sono la ricerca filtrata per nome, tipo o autore dell'opera; l'inserimento di un'opera; l'eliminazione di una o più opere; la modifica dei campi di un'opera già presente.

Gerarchia



MuseoItem: classe base polimorfa astratta.

Campi dati:

- *nome,* tipo string che descrive il nome dell'opera;
- *autore*, tipo string che descrive autore dell'opera;
- *descrizione,* tipo string che descrive brevemente l'opera;
- *dataScoperta,* tipo QDate che descrive data di ritrovamento.
-

DocumentItem: classe astratta derivata direttamente da MuseoItem che descrive opere di tipo testuale.

Campi dati:

- *dataDocumento,* tipo QDate che indica la data in cui il documento è stato creato.

BookItem: classe concreta derivata direttamente da DocumentItem che descrive dei libri.

Campi dati:

- *prefazione,* tipo string che indica la prefazione del libro;
- *copertina,* tipo string che contiene path dell'immagine relativa alla copertina del libro.

LetterItem: classe concreta derivata direttamente da DocumentItem che descrive delle lettere.

Campi dati:

- *testo,* tipo string che indica il testo della lettera;
- *destinatario,* tipo string che indica il destinatario della lettera.

MagazineItem: classe concreta derivata direttamente da DocumentItem che descrive periodici.

Campi dati:

- *cat,* tipo enum categoriaM che descrive i tipi di periodici (Giornale, Quotidiano, Rivista, Almanacco);
- *primaPagina,* tipo string che contiene path dell'immagine relativa alla prima pagina del periodico.

PictureItem: classe derivata concreta direttamente da MuseoItem che descrive opere grafiche.

Campi dati:

- *cat,* tipo enum categoriaP che descrive il tipo di opera (ritratto, paesaggio, natura morta, fotografia);
- *soggetto,* tipo string che descrive il soggetto dell'opera;
- *movimentoArtistico,* tipo string che descrive il movimento di cui fa parte l'opera;
- *foto,* tipo string che contiene il path relativo a un'immagine dell'opera.

StatueItem: classe concreta derivata direttamente da MuseoItem che descrive opere scultoree.

Campi dati:

- *cat,* tipo enum categoriaS che descrive il tipo di opera (busto, bassorilievo, altorilievo, scultura equestre);
- *soggetto,* tipo string che descrive il soggetto dell'opera;
- *materiale,* tipo string che descrive il materiale utilizzato per l'opera;
- *foto,* tipo string che contiene il path relativo alla foto della statua.

Tutte le classi della gerarchia hanno inoltre ridefiniti gli operatori di uguaglianza (==) e di disuguaglianza (!=).
In tutte le classi sono presenti dei metodi *get* in modo da rendere le informazioni disponibili anche all'esterno della classe. Lo stesso vale per i metodi *set* disponibili per tutte le classi della gerarchia e per tutti i campi dati, ad eccezione del *tipo* e della *categoria* per le classi che la prevedono. Questo per scelta di implementazione, in quanto una modifica del tipo di un oggetto (una statua che diventa un libro) ne modificherebbe l'essenza stessa, di conseguenza se ci si dovesse trovare in una circostanza simile la procedura da attuare sarebbe quella di eliminare l'oggetto indesiderato e crearne uno nuovo del tipo voluto.

Qontainer

Il container è stato sviluppato implementando un array dinamico a tutti gli effetti, quindi provvisto dei campi *size*, *capacity* e del metodo *resize()*, che provvede a raddoppiare la capacità dell'array quando lo spazio disponibile è terminato.
Inoltre, è provvisto di iteratore e iteratore costante, usati per scorrerlo durante le varie operazioni.
Questa struttura permette inserimento, rimozione in coda e indicizzazione in tempo costante.
Il container è però fornito anche di un metodo di eliminazione nel mezzo, quindi non conoscendo l'indice si dovrà scorrere l'array fino a trovare corrispondenza; questo metodo ha una complessità lineare.

Considerando che la maggior parte delle operazioni saranno inserimenti (in coda) e accessi indicizzati per modifiche, la miglior soluzione sembra quella dell'array dinamico.

Progettazione

Per quanto riguarda il design pattern, è stato scelto di utilizzare la tecnologia Model-View, questo perché Qt supporta questa architettura grazie al suo sistema di *segnali* e *slot*.

La view principale sfrutta la classe QTableView fornita da Qt, preferita a QListView in quanto rispetto quest'ultima permette di fornire più dettagli dell'opera.

Per interfacciare il modello dei dati alla view, è stata utilizzata la classe QTableModel, derivata da QAbstractTableModel fornita da Qt. Inoltre, grazie a QProxyModel, derivata da QSortFilterProxyModel, che fa da modello custom a QTableModel, si riesce a filtrare ed effettuare ricerche sul modello evitando sprechi di spazio non dovendo utilizzare altri contenitori di supporto per salvare temporaneamente dati.
Le modifiche effettuate sul modello dei dati, grazie a questo collegamento, si riflettono sulla view e viceversa.

Polimorfismo

Metodi virtuali utilizzati:

- *virtual string getTipo() const* : metodo dichiarato virtuale puro nella classe base astratta MuseoItem. Ne viene fatto l'override in tutte le classi concrete della gerarchia, altrimenti non istanziabili (BookItem, LetterItem, MagazineItem, PictureItem e StatueItem).

  Il metodo restituisce una stringa che indica il tipo derivato dell'oggetto di invocazione.

Gestione file

La gestione del salvataggio e caricamento del file viene sviluppata attraverso la lettura e scrittura di file in formato XML. Questo formato è stato scelto sia per la comprensibilità del file generato, sia perché Qt offre diverse classi che ne permettono la costruzione e manipolazione in modo relativamente facilitato, come *QXmlStreamReader* e *QXmlStreamWriter*, usate rispettivamente per lettura (load) e scrittura (write) del file.
La classe XmlIo è stata creata per gestire le funzioni di *read* e *write* del file XML.

Compilazione e note utili

Nella cartella di consegna è presente il file *MuseoManager.pro* che è stato modificato con l'aggiunta dei comandi "CONFIG +=c++11" e "QMAKE_CXXFLAGS += -std=c++11", in modo da poter includere le funzionalità di c++11, come le keyword *default* e *override*.

Per il salvataggio file, specificare nella riga del nome anche l'estensione: es. *"nomefile.XML"*.

L'intero progetto è stato sviluppato su macchina virtuale VirtualBox con immagine fornita dal docente.
Versione Qt 5.9.5. Versione Qt Creator 4.5.2.

Ore di lavoro

Analisi preliminare del problema: 5 ore.

Progettazione modello e GUI: 13 ore.

Apprendimento libreria Qt: 12 ore.

Codifica modello e GUI: 10 ore.

Debugging: 6 ore.

Testing: 4 ore.

Totale: 50 ore.