Simone Barco
Matricola: 1052302
Anno Accademico: 2015/2016

Relazione Progetto Programmazione Ad Oggetti

Informazioni sullo sviluppo e considerazioni generali

Il progetto è stato sviluppato su un sistema Linux Mint versione 17.3. Per lo sviluppo è stato utilizzato l'IDE QtCreator v.3.2.1 con versione delle librerie Qt v.5.3.2 (64 bit) e compilatore g++ v.4.8.4.
Per la parte grafica non è stato usato lo strumento QtDesigner.
Il progetto è stato testato anche in laboratorio dove è presente un sistema Linux Ubuntu versione 12.04 LTS, con librerie Qt v.5.3.2 e compilatore g++ v.4.6.
In tutti i casi il progetto compila e funziona correttamente.

Il programma è stato sviluppato cercando di fornire all'utente un'interfaccia chiara e facilmente comprensibile. Inoltre, è stata utilizzata l'architettura model/view, che permette di separare i dati dal modo in cui essi sono presentati, fornendo un sistema più versatile e flessibile.

Scopo

Lo scopo del progetto è quello di sviluppare un'applicazione C++/Qt che permetta di interagire con un contenitore di dati. In particolare, deve essere possibile aggiungere, modificare, eliminare e ricercare elementi all'interno del suddetto container.

Logica del programma

Nel progetto è stato definito un template di classe `Container<K>`, che implementa un contenitore come lista di nodi, i quali memorizzano generiche informazioni `K`.
Il container possiede una classe `nodo`, annidata e privata, che memorizza l'informazione di tipo `K` e il puntatore al nodo successivo. In particolare, mantiene un riferimento al primo nodo della lista.
È stato implementato per gestire la memoria senza condivisione, quindi, è stato creato un metodo statico che effettua copie profonde della lista e i distruttori sono stati reimplementati in maniera tale da distruggerne ogni nodo.
È stata definita anche una classe `C_iterator`, annidata e pubblica, che implementa un iteratore per il contenitore, con la funzione di permettere l'accesso ai dati presenti in esso.

- Inserimento
  È stato implementato il metodo `add(const K&)` che inserisce un nuovo nodo in testa alla lista.

- Modifica
  La modifica è stata implementata con il metodo `replace(const K& k, const K& value)` che scorre la lista alla ricerca dell'oggetto `k` e lo sostituisce con `value`.

- Eliminazione

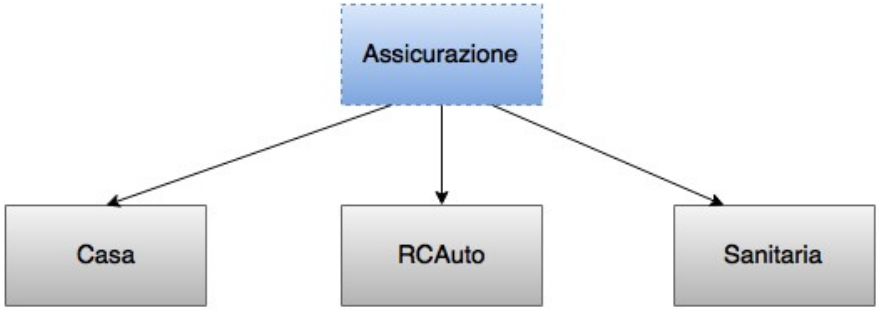
  La rimozione di un elemento dal contenitore, scorre i nodi della lista finché non trova quello da eliminare o finché non arriva alla fine. In caso venga trovato viene eliminato.

- Ricerca

  La ricerca può essere effettuata tramite due metodi:

  - `find`: prende come parametri un puntatore a funzione e un `K`. Ritorna un nuovo container con gli oggetti che soddisfano i termini di ricerca.
  - `findIndex`: prende come parametri un puntatore a funzione, un `Container<K>& k` e un `K& k2`. Ritorna una `QList<int>` contenente gli indici degli oggetti di `k` uguali a `k2`.

Gerarchia



La gerarchia che riempe il contenitore è composta da quattro classi, di cui una astratta e tre concrete.
La classe base *Assicurazione* è astratta e serve a rappresentare un generico oggetto di tipologie di assicurazioni. *Casa*, *RCAuto* e *Sanitaria* sono le sottoclassi, che oltre ai vari costruttori e metodi `get` e `set`, ereditano tre metodi virtuali puri di *Assicurazione*, che vengono reimplementati in tutte le classi della gerarchia:

- `clone()`: metodo di clonazione;
- `isEqual()`: controlla se due oggetti di tale classe sono uguali (è stato ridefinito anche l'operatore di uguaglianza `operator==` per poter implementare questo metodo);
- `prezzoAnnuo()`: restituisce il prezzo annuo del tipo di assicurazione scelta.

SmartP

Per popolare il container, è stata implementata una classe `SmartP`, che contiene un puntatore polimorfo alla classe base della gerarchia.

StaticCont

La classe `StaticCont` è stata creata, invece, per contenere un'istanziazione statica del container, e aggiunge dei metodi utili alla conversione degli indici da `Container` alle varie `QTableView`.

ContainerTableModel

La classe `ContainerTableModel`, derivata da `QAbstractTableModel`, implementa il modello usato dalla

view per accedere ai dati.
Ogni metodo all'interno di questa classe, prima di eseguire le varie operazioni, controlla da quale tipo di tabella è invocato, per poter richiamare le funzioni corrette.
La classe `ContainerTableModel` aggiunge a `QAbstractTableModel` cinque metodi, di cui quattro utili ad inserire una nuove voce nella tabella e uno ad aggiornare un elemento quando viene modificato.

Grafica del programma

Visualizzazione

La classe `MainWindow` deriva da `QMainWindow` ed è quella che crea l'interfaccia principale dell'applicazione.
Questa classe, oltre a creare `QMenu` e `QAction`, istanzia anche una `QLineEdit` utile per la ricerca.
Per la visualizzazione, modifica ed eliminazione dei dati, sono state istanziate (per ognuna delle tre classi concrete della gerarchia) una `QTableView`, un `ContainerTableModel` e due `QPushButton`. La scelta di separare le tre tipologie di assicurazioni in tabelle differenti è pensata per permettere una maggiore chiarezza ed usabilità dell'applicazione stessa.
Nelle tabelle vengono visualizzate solamente le colonne più importanti, con la possibilità di vedere i dettagli più specifici aprendo un'altra finestra, che permette anche di modificarli.

Inserimento

Per inserire nuovi oggetti nel contenitore, si accede al menu *Modifica*, in cui si sceglie il tipo di assicurazione che si vuole aggiungere. Tramite segnali e slot, si agisce sul corrispondente model che aggiungerà in testa alla tabella la voce richiesta, e tramite il metodo `add(const K&)` lo aggiungerà anche all'interno che contenitore (che è unico per tutti e tre i modelli).
Ogni volta che viene creata una nuova riga, questa viene selezionata.

Selezione

Per gestire la selezione, sono stati istanziati tre `QItemSelectionModel`, in modo da tenere traccia degli oggetti selezionati per ciascuna tabella.
Quando viene selezionata una riga, vengono visualizzati due pulsanti: `detailBtn` e `deleteBtn`, se invece vengono selezionate più righe, sempre della stessa tabella, verrà visualizzato solo il pulsante `deleteBtn` permettendo così l'eliminazione di un insieme di voci.

Modifica

Il pulsante `detailBtn` è collegato allo slot `openDetail()`; grazie al metodo `StaticCont::index(int k)`, a partire dalla riga selezionata, viene calcolato l'indice reale all'interno del container di tale elemento.
I vari campi dati vengono salvati e passati, assieme a un `QString` che identifica il tipo di assicurazione, a `DetailsDialog`, derivata da `QDialog`, che permette la creazione di una modal dialog, bloccando gli input ad altre finestre visibili della stessa applicazione.
`DetailsDialog` istanzia vari widget (`QDateEdit`, `QLineEdit`, `QSpinBox`, `QComboBox`,

QCheckBox e QRadioButton) che permettono la modifica di questi campi dati.
Alla creazione di DetailsDialog viene controllata la stringa che identifica la tipologia dell'assicurazione in modo da permettere la visualizzazione dei widget corretti per ogni categoria.
I campi dati non presenti nella tabella vengono ricavati utilizzando i metodi get e i widget vengono impostati con i suddetti valori.
Nel caso in cui l'assicurazione che si sta modificando è di tipo *RCAuto*, se il campo *Età* viene impostato a 18 viene visualizzato un QMessageBox con un warning e il campo *Cavalli Fiscali* viene bloccato al valore 11.
Alla chiusura del dialog, in caso il valore ritornato sia QDialog::Accepted, vengono aggiornati nel contenitore i campi dati modificati, di quell'oggetto. Infine viene aggiornata la tabella.
Per quanto riguarda la modifica nel container, ho implementato il metodo replace, ma non ho trovato casi di utilizzo, in quanto uso già uno SmartP con cui posso facilmente modificare l'oggetto.

Ricerca

Per quando riguarda la ricerca, è stata istanziata una QLineEdit per cercare un oggetto in base al nome identificativo dell'assicurazione.
Quando viene inserito del testo nella QLineEdit e viene premuto *Enter*, viene richiamato lo slot search() che effettua la ricerca. Se essa ha esito negativo viene visualizzato un QMessageBox il quale avvisa che non è stata trovata nessuna assicurazione, all'interno del contenitore, con il nome richiesto.
Se invece la ricerca ha esito positivo viene creata una SearchDialog (derivata anch'essa da QDialog), che istanzia una nuova QTableView e un nuovo ContainerTableModel, da cui vengono nascosti quelli che non soddisfano la ricerca. In questo caso l'istanza di ContainerTableModel ha un nome diverso dalle altre in quanto verrà visualizzata una colonna in più con la tipologia dell'assicurazione, scelta compiuta in quanto è possibile avere voci di tipologie diverse ma con lo stesso nome.
Selezionando una delle voci di questa tabella verrà visualizzato un pulsante che chiuderà la dialog e selezionerà la voce che è stata cercata all'interno della MainWindow.
A livello di sviluppo, si è scelto di utilizzare il metodo findIndex in quanto find avrebbe richiesto di istanziare un'ulteriore contenitore statico solamente con i risultati (che potenzialmente poteva essere sempre di grandezza pari a uno) portando quindi ad un dispendio di memoria. Il metodo find è stato comunque implementato per permettere differenti modalità di ricerca.

Eliminazione

Il pulsante deleteBtn è connesso allo slot deleteObj() che memorizza, in una lista, gli indici selezionati. Tramite il metodo qSort essi vengono ordinati in maniera ascendente. Ciclando, quindi, la lista dalla sua fine all'inizio, è possibile eliminare le righe dal basso verso l'alto in modo da non dover aggiornare gli indici dopo che una riga è stata eliminata.deleteObj chiama removeRows(), metodo ridefinito in ContainerTableModel, che deve essere reimplementato nei modelli che supportano la rimozione di righe. removeRows chiama a sua volta il metodo removeOne di Container che elimina l'oggetto selezionato.
Alla fine della rimozione viene emesso il signal layoutChanged() per forzare l'aggiornamento del layout delle tabelle.