Alessandro Lovo
Matricola: 1142682

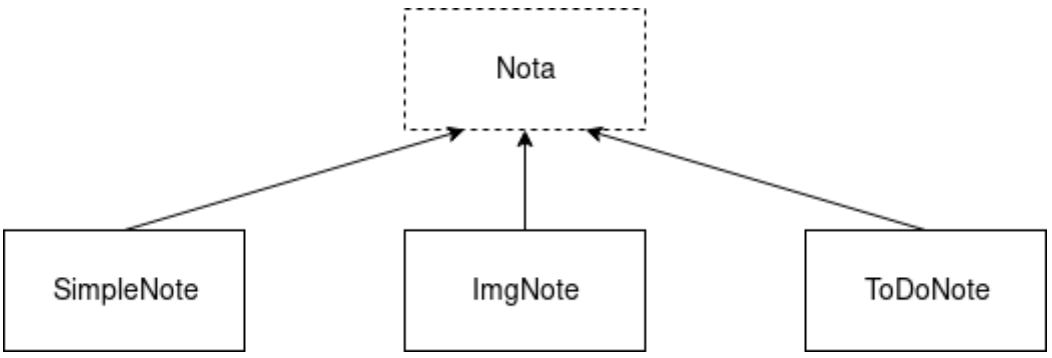
QNOTE

QONTAINER : Progetto di Programmazione ad Oggetti, a.a. 2018/2019

Introduzione

QNote è un'applicazione che consente di gestire delle note aggiunte dall'utente. È infatti possibile creare, modificare, ricercare ed eliminare note, oltre che importarle ed esportarle su file in formato JSON. Sono presenti tre tipologie di note: note semplici con titolo, testo e tag per la catalogazione delle note stesse; note con immagine che includono un'immagine selezionata dall'utente che verrà codificata in Base64 per il salvataggio su file; note con obiettivi che introducono una lista di elementi nella quale selezionare degli obiettivi raggiunti.

Gerarchia di tipi



La classe base della gerarchia è la classe base astratta polimorfa ***Nota***. Essa è caratterizzata da quattro campi dati privati che sono quindi sempre presenti in tutte le tipologie di nota: titolo e descrizione di tipo QString, tag di tipo QVector<QString> e dataUltimaModifica di tipo QDateTime, che indica data e ora in cui la nota ha subito l'ultima modifica dall'utente.

Nota possiede un metodo virtuale puro di clonazione polimorfa *clone* utilizzato per restituire una copia esatta di una nota qualsiasi.

Nota ha tre sottotipi concreti:

- ***SimpleNote****:* rappresenta una nota di testo semplice con i campi essenziali: titolo, descrizione, tag e data di ultima modifica.
- ***ImgNote****:* rappresenta una nota in grado di visualizzare un'immagine selezionata dall'utente. Contiene infatti il campo dati privato *img64* di tipo QString che racchiude la codifica in Base64 dell'immagine.
- ***ToDoNote****:* rappresenta una nota contenente una lista di obiettivi. Tale lista è indicata dal campo privato *toDoList* di tipo QList<ToDoItem>. L'utente può creare, rimuovere, modificare gli obiettivi e cambiarne lo stato in completato o da completare.

Come scritto, la lista di obiettivi nelle *ToDoNote* utilizza elementi di tipo ToDoItem, definito al di fuori della gerarchia. Un obiettivo (*ToDoItem*) ha due campi privati: *target*, ovvero il titolo di tipo QString e *completato* di tipo bool per indicare se l'obiettivo sia stato portato a termine o meno.

Container<T>

Il contenitore del progetto è stato implementato come un array dinamico. Nella sua parte privata possiede una struttura chiamata *base_vector*, la quale contiene il vettore di elementi di tipo generico T e la sua dimensione.

Il contenitore ospita la classe pubblica ***Iterator***, per scorrere il container stesso, e utile per la rimozione di un particolare elemento (in tempo O(1)). Iterator mantiene infatti nella sua parte privata un puntatore al *base_vector* del container e l'indice dell'elemento puntato dall'iteratore. Grazie a questo indice in fase di rimozione, passando l'iteratore, si conosce la posizione dell'elemento da eliminare dal vettore.

Sono presenti poi una classe base pubblica Ricerca e due classi base pubbliche astratte Serializzazione e Deserializzazione.

- La classe base ***Ricerca*** fornisce una ricerca "di default" che seleziona tutti gli elementi nell'ordine naturale del container. L'override di *bool operator()(const T&)* nelle classi da essa derivate serve a stabilire se un elemento (in questo caso una Nota) vada incluso oppure no tra i risultati di una ricerca (nel caso della classe derivata ***VisualizzazioneOrdinata*** non si effettua override perché ogni elemento viene incluso nei risultati, mentre per la classe ***RicercaTesto*** viene verificata la presenza di un pattern nel titolo, nella descrizione o nei tag delle note e solo se vi è un riscontro la nota viene aggiunta ai risultati). Il metodo *getResults* fornisce invece l'implementazione dell'algoritmo per ordinare i risultati di ricerca nel modo desiderato. Nel caso della *VisualizzazioneOrdinata* delle note, queste vengono ordinate in base alla data di ultima modifica dalla più recente.
- La classe base astratta **Serializzazione** viene utilizzata nella sua implementazione concreta ***SerializzaNote*** per serializzare un elemento (in questo caso una Nota da serializzare in formato JSON). L'override del metodo virtuale puro *void operator() (const*

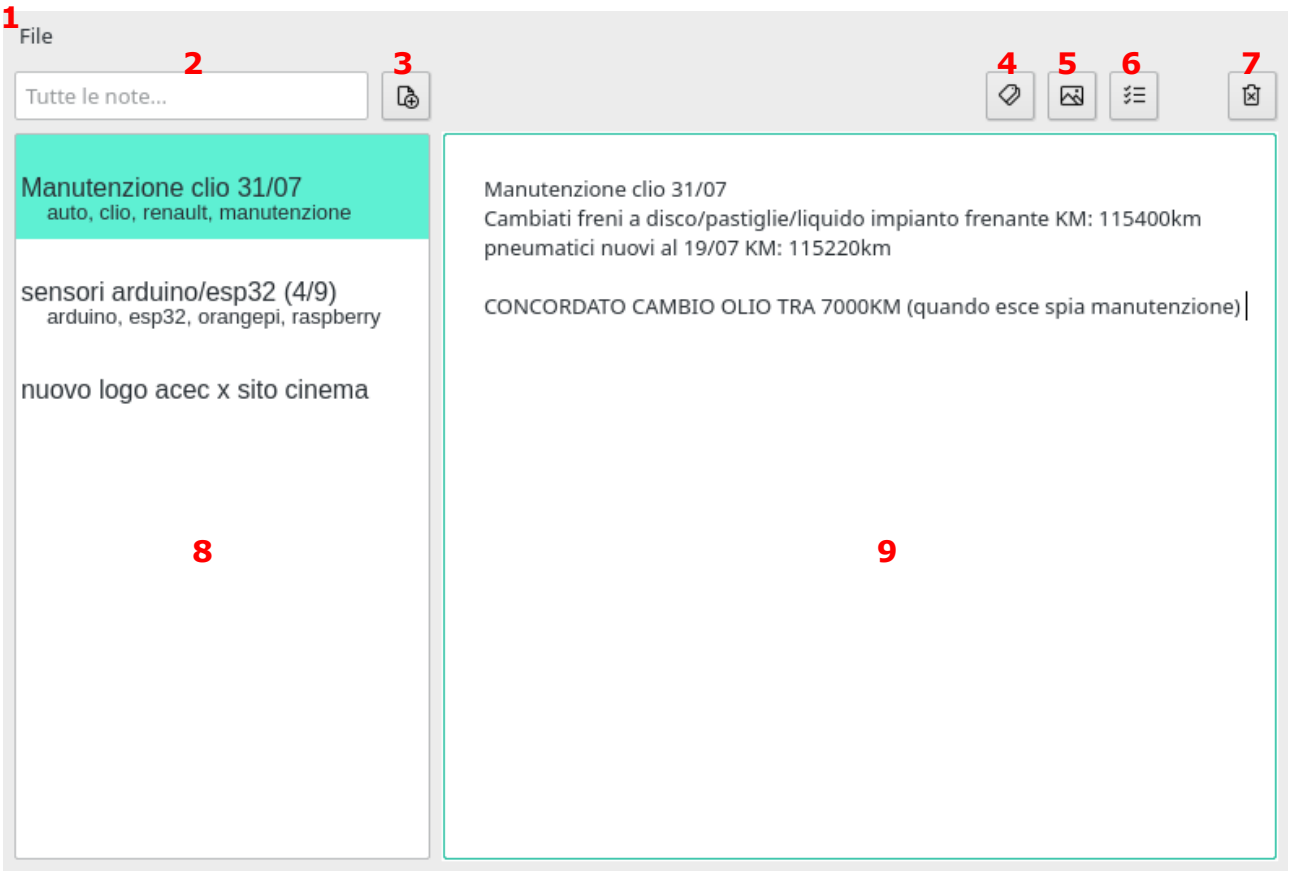
T& elemento) permette infatti in SerializzaNote di verificare la tipologia della nota e procedere al salvataggio appropriato della stessa aggiungendola al file JSON prodotto. Il metodo *void serializza(Serializzazione& cs)* presente nel container non fa altro che applicare la serializzazione per ogni elemento del vettore.

- La classe base astratta **Deserializzazione** viene utilizzata nella sua implementazione concreta ***DeserializzaNote*** per deserializzare un container di elementi (in questo caso un container di elementi di tipo Nota salvato in formato JSON).

Serializzazione e deserializzazione

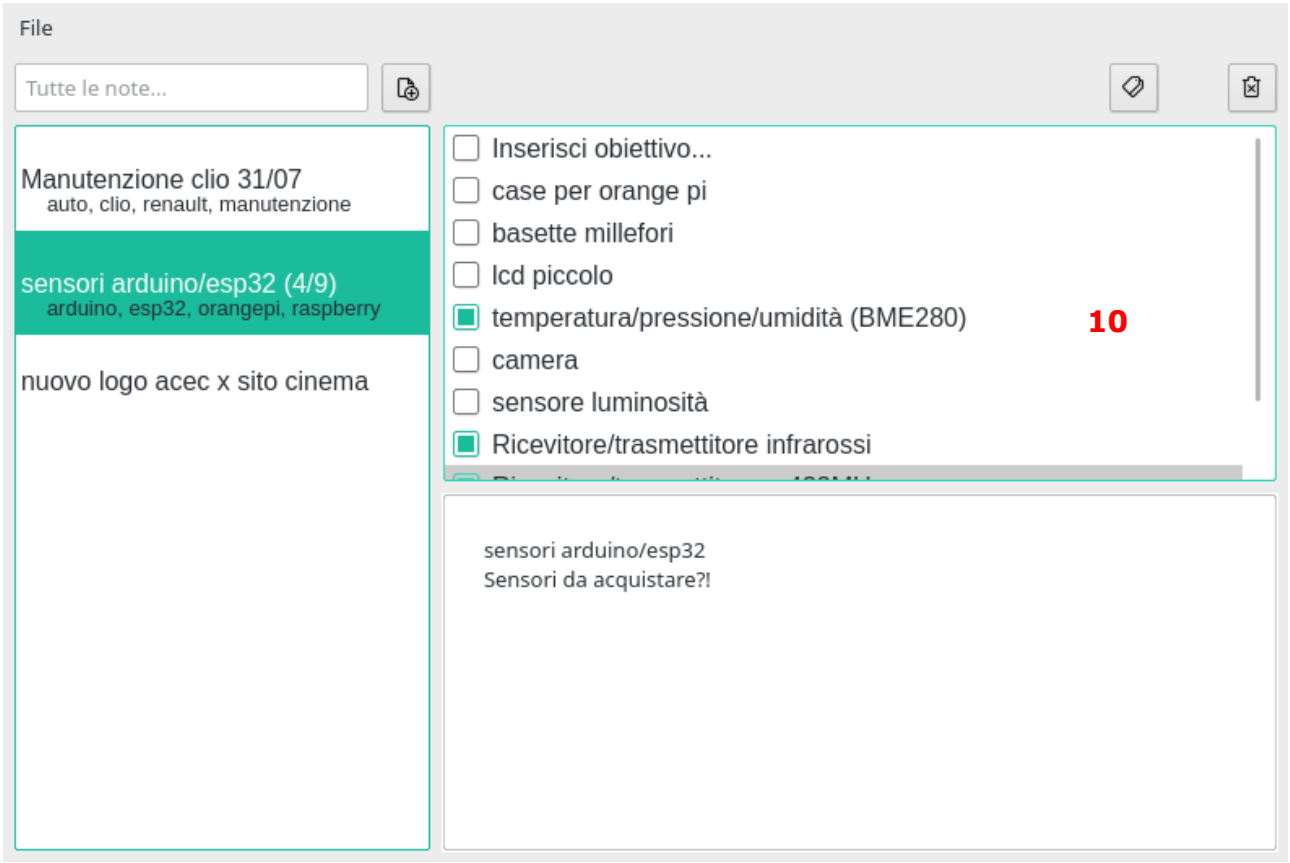
Come già anticipato, QNote permette il salvataggio e l'apertura di file in formato JSON che rappresentano contenitori di note. Tale formato è supportato da una grandissima moltitudine di linguaggi e Qt mette a disposizione diverse classi che ne rendono l'utilizzo intuitivo.
All'interno dei file di salvataggio prodotti dal programma troviamo l'array JSON "note" che racchiude la lista di note del contenitore. Ogni oggetto JSON di questo array contiene quindi sempre la coppia nome/valore dei campi: "dataUltimaModifica", "descrizione", "tag" (che è un array JSON), "tipo" (utilizzato in fase di deserializzazione per comprendere la tipologia della nota da ricostruire) e "titolo". Le note con immagine hanno in aggiunta il valore "image" che contiene la codifica in Base64 dell'immagine salvata, mentre le note con lista di obiettivi sono caratterizzate dall'array "toDoList" di oggetti JSON aventi i campi "status" e "target", ovvero lo stato e il nome dell'obiettivo.

Manuale utente GUI



1. Accesso al salvataggio rapido, importazione ed esportazione note ed uscita dal programma.

2. Casella di ricerca: il testo inserito verrà cercato nei titoli, nei testi e nei tag delle note, mostrando la lista di risultati trovati. Cancellando il testo cercato, verranno nuovamente mostrate tutte le note.
3. Crea una nuova nota.
4. Aggiunge un tag alla nota selezionata (verrà controllato se il tag esiste già nella nota).
5. Apre una finestra di dialogo per la selezione di un file immagine da aggiungere alla nota (visibile solo selezionando una nota semplice).
6. Trasforma la nota corrente in una nota con lista di obiettivi (visibile solo selezionando una nota semplice).
7. Elimina la nota selezionata.
8. Lista di tutte le note (in fase di ricerca mostra solo le note desiderate). Vengono visualizzati il titolo (per le note con lista di obiettivi anche il conteggio degli obiettivi completati) e gli eventuali tag.
9. Area di testo. La prima riga dell'area di testo diventa automaticamente il titolo della nota, mentre tutto il resto ne forma la sua descrizione. In caso di note con immagine, sopra l'area di testo viene visualizzata l'immagine, mentre in caso di note con lista di obiettivi viene visualizzata la to-do list.



10. Lista di obiettivi (TODO list). Viene visualizzata sopra l'area di testo quando è selezionata una nota con lista di obiettivi. Il primo elemento della lista viene sempre visualizzato in automatico con il nome "**Inserisci obiettivo...**". Non appena il titolo o lo stato di questo elemento cambiano, verrà creato un nuovo elemento "Inserisci obiettivo…" all'inizio della lista. Un obiettivo viene considerato completato se il corrispondente checkbox è attivo. Per modificare il titolo di un elemento della lista è necessario fare **doppio click** sopra di esso. Per cancellare un obiettivo, fare doppio click per entrare in modifica e cancellarne completamente il nome: l'obiettivo sarà rimosso dalla lista.

Compilazione ed esecuzione

Il progetto richiede il file QNote.pro fornito per poter compilare. Infatti, è necessario che nel file .pro sia specificata l'inclusione dei widgets di Qt e l'utilizzo dello standard C++11.
La compilazione può quindi essere lanciata con i comandi *qmake* e *make* e il programma eseguito con il comando *./QNote*.
Il programma all'avvio tenta di eseguire l'importazione delle note da un file qNote.json: se il file non è presente aggiunge una nota di default. In questo caso se si procede al successivo salvataggio delle modifiche senza esportare su un file scelto dall'utente (e senza aver importato nel frattempo note da qualche file .json), le note saranno serializzate proprio in un file chiamato qNote.json, per essere caricate automaticamente all'avvio successivo.

Libreria Qt

Nel progetto viene fatto uso di alcune classi di Qt: QString, QVector, QList, QDateTime, QJsonDocument, QJsonArray e QJsonObject. Se per le prime tre la sostituzione per non dipendere da Qt è semplice, è necessario implementare adeguate alternative negli altri casi.

Tempistiche di sviluppo

Sono state impiegate all'incirca 50 ore per completare il progetto, così distribuite:

- Analisi preliminare del problema: 2 ore
- Progettazione modello: 3 ore
- Progettazione GUI: 4 ore
- Apprendimento libreria Qt: 6 ore
- Codifica modello: 20 ore
- Codifica GUI: 8 ore
- Debugging: 3 ore
- Testing: 2 ore

Ambiente di sviluppo

Il progetto è stato sviluppato sul seguente ambiente:

- Sistema operativo: Manjaro Linux
- Versione di Qt: 5.13.0
- Versione g++: 9.1.0