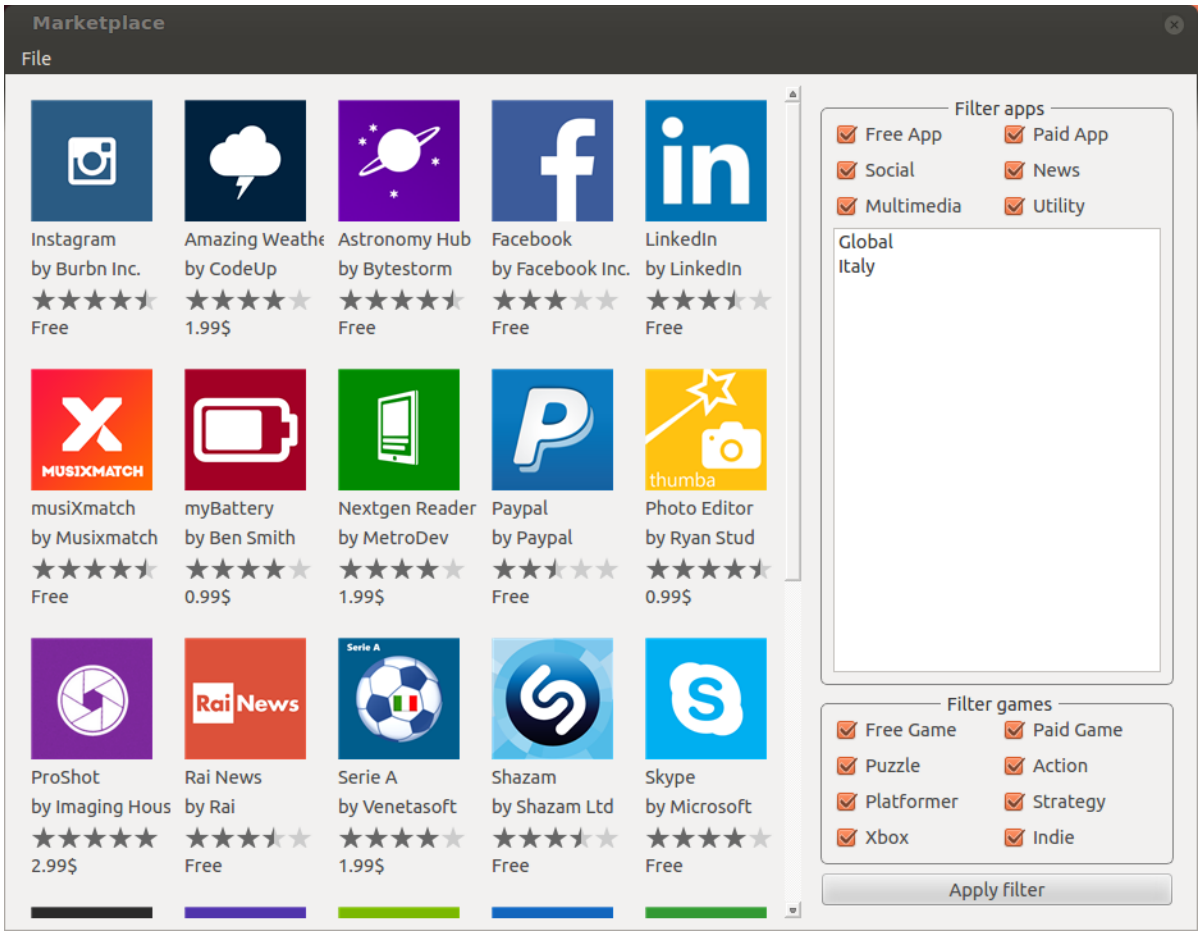
Progetto di Programmazione ad Oggetti

MARKETPLACE (qDB)

di Cusinato Giacomo - 1054137

Ambiente di sviluppo:
Windows 8.1 / Ubuntu 12.04
Versione librerie Qt: 4.8.0
IDE: Qt Creator 2.1.4



1 Premessa

Il progetto è stato realizzato in ambiente Windows (in versione 8.1 Pro 64bit) da casa e Linux dalle macchine del laboratorio. Per convenienza ed ordine è stato scelto di dividere i file riguardanti la grafica e quelli riguardanti la logica in due sub-directory differenti. Il file di progetto viene prodotto correttamente in ambiente Linux tramite il comando qmake mentre in Windows, dopo aver eseguito

lo stesso comando, è necessario aggiungere le directory sopracitate alla voce di inclusione delle directory di default (INCLUDEPATH += . \ Grafica \ Logica). Risulta inoltre presente un file projres.qrc, essenziale all'IDE per identificare le risorse locali presenti nel progetto, ed un file container.xml che contiene un database preimpostato (serializzato direttamente dal programma) che è possibile utilizzare per avere del contenuto iniziale. La parte grafica è stata interamente realizzata tramite l'editor di QtCreator, non è stato fatto uso, quindi, di QtDesigner.

2 Introduzione

Marketplace rappresenta un contenitore di applicazioni presenti nello store di un un sistema operativo mobile. Gli item del contenitore sono essenzialmente di due tipi: giochi ed applicazioni, con attributi comuni e disgiunti. Tutte gli elementi vengono rappresentati tramite una vista a tabella dove vengono fornite le informazioni essenziali per ognuno. L'interfaccia grafica offre la possibilità di filtrare e cercare gli elementi desiderati a seconda delle caratteristiche dei vari giochi ed applicazioni e la modifica degli stessi qualora presenti nella vista attuale. Risulta anche possibile aggiungere o rimuovere nuovi elementi e salvare lo stato corrente del database su file.

3 Scelte progettuali

Il progetto è stato realizzato cercando di mantenere una netta distinzione tra parte grafica e parte logica. Nello specifico, è stato scelto di seguire il pattern architetturale Model-View-Controller per il Container principale utilizzando le classi custom ContainerController, ContainerModel, ConteinerView e ContainerDelegate. Il tutto è stato realizzato ereditando, per le ultime tre citate, delle specifiche classi appartenenti al Model/View framework di Qt.

La classe Container, inoltre, è stata realizzata in modo tale da gestire in modo condiviso la memoria attraverso l'uso di puntatori smart. Questo scelta risulta più la efficace in quanto nel Controller vengono utilizzati più Container (quello originario, con tutti gli elementi presenti, e quello visuale, con gli elementi visualizzati in una certa situazione) e, in caso sia presente un elevato numero di elementi, eventuali copie risultano meno costose in termini di memoria.

4 Parte logica

La parte logica del progetto è composta delle tre classi che compongono la gerarchia di dati (StoreItem, App, Game), il contenitore ed il puntatore smart alla classe base della gerarchia, che andrà ad identificare il tipo del container (Container<T>, SmartPoiner), un gestore delle eccezioni e le parti logiche dell'approccio MVC (ContainerController, ContainerModel, ContainerDelegate).

Come richiesto, StoreItem è una classe astratta, contiene infatti il metodo virtuale puro clone() che funge da costruttore di copia per le classi che la ereditano utilizzando chiamate polimorfe. StoreItem identifica un elemento generico del Marketplace e contiene i campi dati title, developer, number of ratings, quality e price. Le app (classe App) sono caratterizzate da una localizzazione e da una categoria (campi dati localization e category) mentre i giochi (classe Game) si differiscono per categoria e brand (campi dati category e xbranding). In tutto il progetto, l'identificazione di un elemento (distinzione tra App e Game) viene effettuato tramite il costrutto dynamic_cast su di un puntatore alla classe astratta.

Tale puntatore è il campo dati di SmartPointer che identifica il tipo del template Container<T> (conversione implicita SmartPointer => StoreItem*). Il Container, come detto in precedenza, è stato progettato in modo tale da mantenere una gestione condivisa della memoria attraverso l'uso della classe smartp e l'overloding dei vari operatori che gesticono il conuting reference della classe item. Il contenitore è stato dotato dei metodi essenziali per la gestione dei dati (es. push_back, push_front, remove, ecc.) e di due iteratori (uno costante) per effettuare lo scorrimento della lista concatenata.

ContainerModel rappresenta il modello dei dati presenti nel container (solo quelli visibili in nella vista, ovvero quelli contenuti nel campo dati viewContainer). ContainerModel eredita la classe astratta QAbstractTableModel appartenente al framework model/view di Qt. Essendo QAbstractTableItem astratta, è stato necessario effettuare l'overloading dei metodi columnCount(..) che restituisce il numero di colonne del modello, rowCout(..) che restituisce le righe e data(..) che restituisce un oggetto QVariant che rappresenta i dati di un certo indice del contenitore e del modello (è stato scelto di rappresentare i campi di un elemento tramite la classe QStringList, in quanto compatibile con QVariant).

ContainerController rappresenta l'interfaccia che funge da tramite tra ContainerModel e ContainerView. Eredita QObject in modo da poter definire Signals e Slots e contiene come campo dati il container principale (mainContainer) oltre ad ulteriori oggetti utili per gestire il contenitore a seconda dell'interazione dell'utente con l'interfaccia grafica. Il contruttore di Container, come prevedibile, accetta un oggetto di tipo ContainerView ed uno di tipo ContainerModel. Nel controller sono inoltre presenti gli eventi per serializzare e deserializzare il container tramite le classi QXmlStreamWriter e QXmlStremReader.

ContainerDelegatate eredita da QStyledItemDelegate (appartenente al model/view framework di Qt) la quale permette di gestire la rappresentazione grafica di ogni singolo elemento in modo separato della view. Nella classe è stato effettuato l'overloading del metodo paint(..) per renderizzare un custom Widget con i dati di un elemento del contenitore in un certo indice del modello.

5 Parte grafica

La parte grafica è composta da una MainWindow che ospita principalmente tutti gli elementi grafici del programma, come la vista del container (ContainerView), ItemFilterView, AddAppView, AddGameView e la classica MenuBar da cui è possibile aggiungere o rimuovere elementi ma anche il salvataggio su file. Tra i campi dati di questa classe compare anche il Controller del container: tutta

l'interfaccia grafica, infatti, è collegata al Controller tramite il sistema dei Signals&Slots per modellare il database a seconda dei cambiamenti effettuati dell'utente tramite la GUI.

ContainerView rappresenta la vista del container ed eredita la classe QTableView appartenente al Model/View framework di Qt. Tale classe presenta solamente alcune personalizzazioni grafiche inserite nel costruttore ed il segnale emitIndex(..) che avverte il controller dell'inizio della modifica di un elemento del container una volta che l'utente ha interagito con la view.

ItemFilterView è la vista principale per filtrare e cercare gli elementi all'interno del database e, come tutte le classi della parte grafica, deriva da QWidget. ItemFilterView presenta una serie di QCheckBox raggruppati a seconda della tipologia ed un bottone per confermare i filtri applicati. Uno volta applicato il filto, viene emesso al controller un segnale che il filtro degli elementi è cambiato, il tutto rappresentato tramite un dizionario chiave-valore (QHash<string, int>) che contiene informazioni sul bottone e sul suo stato e verrà codificato dal controller. Per rappresentare la scelta della localizzazione è stato scelto di utilizzare una QListView (sempre utilizzando l'approccio Model/View) che viene aggiornata con tutte le diverse localizzazioni delle applicazioni a seconda del modello emesso del controller del contenitore.

TableItem, invece, rappresenta la grafica di un elemento del container, ovvero un'app o un gioco. Per questo accetta nel costruttore un puntatore ad un oggetto di tipo StoreItem oppure una lista di stringhe (il modo in cui vengono rappresentati i campi dati dell'oggetto in ContainerModel). TableItem viene costruito in ContainerDelegate dove viene renderizzato nell'evento paint(..) dell'elemento del modello. Le icone di ogni elemento sono ottenute dal file di risorse (è presente anche un'icona di default per i nuovi elementi) come le stelle che rappresentano la qualità ma queste, al contrario, vengono renderizzate in una QLabel nella classe RatingLabel sempre effettuando l'overloading dell'evento paint(..).

AddAppView e AddGameView permettono all'utente di aggiungere una nuova app o un nuovo gioco al contenitore. Vengono entrambi evocati tramite le rispettive Action della barra dei menù. Uno volta processati i dati, viene emesso un segnale contenente un oggetto di tipo App o Game che verrà accolto da uno slot del controller aggiunto al database.

ItemEditor, invece, è il widget per editare o rimuovere qualsiasi elemento. Viene evocato con un click sull'elemento desiderato. ContainerView, infatti, emette un segnale passando al controller un oggetto di tipo QModelIndex con il quale ContainerController può ottenere l'elemento richiesto nel modello. Una volta confermati i cambiamenti, viene ritornata la classica QListString per aggiornare i campi dati dell'elemento presente nel mainContainer. Nel caso venga premuto il tasto Remove, verrà ritornata una lista vuota ed il controller procederà con la rimozione dell'elemento.