Giacomo Sassaro

Progetto di Programmazione ad Oggetti

A.A. 2019/2020

Sassaro Giacomo - 1187566

Maher Ayoub - 1187406

Tovo Gianpiero Giuseppe - 1193350

Relazione di Sassaro Giacomo

QANALYTICS

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO MATEMATICA

Giacomo Sassaro

Abstract e funzionalità

L'applicazione QANALYTICS si occupa di analizzare e tracciare grafici sull'andamento di determinate statistiche relative ad un account social, attualmente supporta Facebook, Instagram e Youtube. L'applicazione dovrebbe raccogliere i dati attraverso delle API, raccoglierli secondo uno schema da noi studiato e successivamente mostrare all'utente un grafico rappresentativo dell'andamento del suo account rispetto a followers, likes, coverage, ecc. Nel nostro progetto l'input dei dati avviene tramite l'inserimento di un file Json anziché tramite API. QANALYTICS potrebbe anche analizzare l'andamento di un singolo “Content” di un account, per esempio di un post di Instagram. Attualmente il modello è predisposto per questa funzionalità, mentre la GUI verrà implementate eventualmente in successive versioni. L'applicazione rende inoltre disponibile il salvataggio di un grafico come immagine e l'esportazione di tutti i dati in essa contenuta in un file Json.

Istruzione di compilazione ed esecuzione

L'applicazione necessita del modulo QtChart, per installarlo su una macchina linux è necessario lanciare da terminale il comando :

sudo apt install libqt5charts5 libqt5charts5-dev

Questo modulo non è installato di default, ma è indispensabile installarlo per il corretto funzionamento di QANALYTICS.

Per compilare correttamente l’applicazione è necessario dare i comandi :

qmake QANALYTICS.pro
make

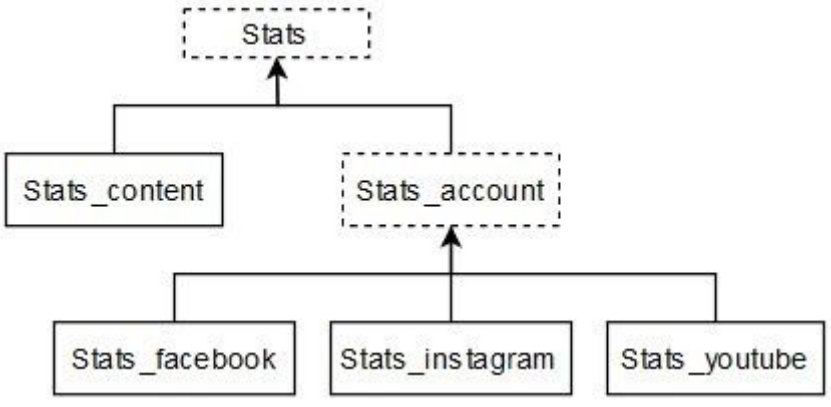
Successivamente si può lanciare l’applicazione con il comando *./QANALYTICS*

Progettazione e descrizione delle gerarchie usate

L’applicazione utilizza il modello MVC, così da separare il modello dalla GUI, le gerarchie potrebbero essere utilizzate per altre interfacce grafiche, restando comunque nel framework Qt in quanto per la gestione dei grafici, anche a livello logico e non solo grafico, siamo stati obbligati ad utilizzare classi di Qt.

Gerarchie

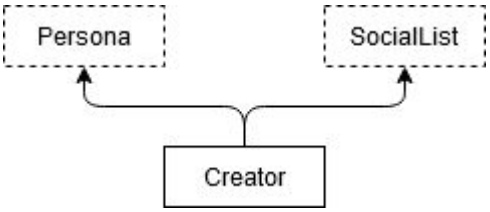
L’applicazione utilizza due gerarchie ed altre classi “singole”.



La seguente gerarchia è la gerarchia più profonda e rappresenta i diversi tipi di Statistiche che utilizza l’applicazione. La gerarchia si compone di una classe base astratta

Stats, dalla quale derivano due classi, quella per le Statistiche dei contenuti, classe concreta, e quella per le Statistiche per gli account, anche questa astratta. Da quest'ultima derivano le classi per le Statistiche per i vari tipi di account, quindi Facebook, Instagram e Youtube. La gerarchia è pensata per essere scalabile, infatti in futuro potrebbero essere implementate le statistiche di altri tipi di account.

Altra gerarchia implementata nell'applicazione è l'eredità multipla di Creator, la classe Creator eredita da Persona, classe astratta, i dati anagrafici, mentre da SocialList, altra classe astratta, la lista degli account Social di quel Creator con relativi metodi per la loro gestione. SocialList è una semplice classe contenitore, la quale implementa come campo privato un vettore che raccoglie i vari oggetti di tipo Account.



Contenitori

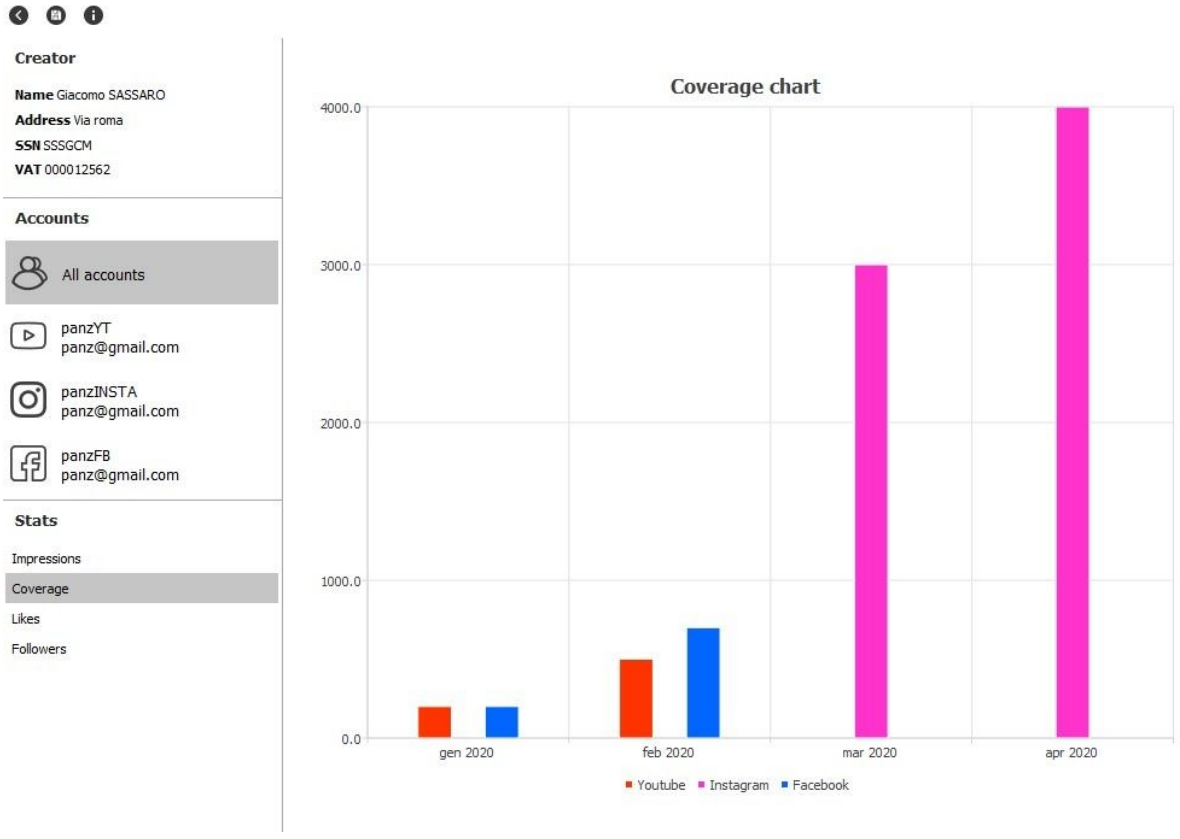
L'applicazione utilizza diversi contenitori, due più semplici, ossia SocialList e CreatorList, i quali non sono altro che delle classi con un vettore di oggetti Account e Creator come campo privato. Mentre il contenitore di oggetti polimorfi è la classe StatsList, la quale viene usata in Account e Content per memorizzare la lista di Stats, StatsList è una lista double linkata di nodi puntatori a Stats, così da poter implementare liste di qualsiasi tipo di Stats (Content, Facebook, Instagram e Youtube), la classe contiene anche due puntatori al primo e all'ultimo nodo, così da rendere più facile la gestione della lista. In StatsList è stato anche implementato un constiterator così da facilitare lo scorrimento della lista. StatsList utilizza costruttore, costruttore di copia, operatore di assegnazione e distruzione profonda.

Interfaccia grafica

L'applicazione si compone di due finestre principali : LandingWindow e GraphsWindow, le quali ereditano da QWidget, e da un dialog: QInfoDialog, che eredita da QDialog. LandingWindow è la prima finestra che si incontra all'apertura dell'applicazione, in seguito all'import dei dati nella finestra viene mostrata la lista di Creator analizzabili, al di sopra della lista è anche disponibile una barra di ricerca dove è possibile cercare il Creator per nome, cognome e codice fiscale. In alto a destra si possono vedere tre pulsanti, a partire da sinistra: importa, esporta e info. Quest'ultimo pulsante lancia

il dialog QInfoDialog in cui sono raccolte diverse informazioni sull'applicazione. Al click su un Creator viene lanciata la finestra GraphsWindow in cui si possono analizzare i dati del Creator selezionato.

GraphsWindow si compone di una barra laterale e di uno spazio sulla destra dedicato al grafico. Nella barra laterale a partire dall'alto si possono leggere i dati anagrafici della persona e subito al di sotto la lista degli account di questa persona. Nel caso in cui un creator abbia più di un account viene reso disponibile anche un pulsante "All Account", con il quale si possono vedere le statistiche comuni a tutti gli account in relazione tra di loro. In base all'account che viene selezionato la lista di stats sottostanti viene aggiornata in base a quelle disponibili per quel tipo di account. Con un click su di una stats viene mostrato il corrispettivo grafico sulla destra.
In alto a destra sono disponibili tre pulsanti, a partire da sinistra il pulsante "indietro" per tornare alla LandingWindow, il pulsante per salvare il grafico come immagine png, e il pulsante info che si comporta come il pulsante info spiegato in precedenza.



Model & Controller

Il Controller ha il compito di mettere in comunicazione la View con il Model grazie alla presenza di un loro puntatore nel campo dati.
Il Model utilizza un oggetto di tipo CreatorList in cui sono contenuti tutti i creator, e di un campo *selected* che punta al creator selezionato, così da rendere più semplice le iterazioni con il suddetto account nella graphswindow. Il modello si occupa inoltre dell'analisi dei dati

per la costruzione di tutti i grafici, che successivamente passano per il controller che li manda alla graphswindow dove vengono mostrati. Il modello possiede anche due metodi per l'import e l'export dei dati in un file di testo Json.

Descrizione chiamate polimorfe

Unica chiamata polimorfa presente nell'applicazione è *clone()*, presente in StatsList nella sottoclasse *node* dove viene usato per la costruzione di un nuovo oggetto che viene puntato dalla lista. Essendo questa una chiamata polimorfa il tipo dell'oggetto da costruire viene stabilito a runtime. Lo stesso vale per il distruttore della gerarchia Stats.
La scarsità di chiamata polimorfe è data dal fatto che la nostra gerarchia non possiede molti comportamenti comuni, gli oggetti in essa contenuti sono prevalentemente differenziati dagli attributi. I metodi presenti in queste classi, visto anche che l'applicazione non fa altro che leggere dati da questi classi, sono per lo più metodi del tipo *getAttributo()* non overloadati. Avremmo potuto ridefinire l'operatore di stampa di queste classi per avere un'ulteriore ipotetica chiamata polimorfa, ma non la avremmo utilizzata in quanto superflua per lo scopo dell'applicazione.

Descrizione I/O

Il programma permette il caricamento e l'esportazione dei dati attraverso l'utilizzo di file JSON. Abbiamo scelto JSON come formato per la sua intuibilità e per la facile gestione di questi tipi di file in Qt.
Import ed Export vengono gestiti tramite due metodi implementati in ogni classe, per l'import il metodo *read()*, mentre per l'export il metodo *write()*.
All'apertura dell'applicazione viene richiesto di importare un file, nel caso in cui l'import fallisca viene notificato all'utente l'errore e l'applicazione crasha. Nello specifico,nel caso di 'import il controller notifica al modello il path del file selezionato dall'utente. Qui viene chiamato il metodo *read()* secondo lo schema in cui abbiamo deciso di salvare i dati, viene quindi prima chiamato in CreatorList, da cui successivamente viene chiamato in Creator, da Creator viene chiamato in Persona e in SocialList, continua così a fare chiamate annidate fino a quando non vengono letti tutti i dati e viene popolata la lista di Creator. Lo stesso vale per l'export, cambia solamente il metodo utilizzato che è appunto il metodo *write()*.

Ambiente di sviluppo

Lo sviluppo è stato effettuato principalmente su :

- Sistema operativo : Windows 10
- Compilatore: MinGW 5.3.0
- Qt : 5.9.5

Note aggiuntive

Le stats sono identificate dalla data, abbiamo deciso infatti che l'applicazione legge una singola stats per mese, ipoteticamente le statistiche vengono raccolte il primo giorno di ogni mese. Nei grafici si leggerà infatti solamente mese e anno.

Giacomo Sassaro

Per la condivisione e il versionamento del progetto abbiamo utilizzato GitHub, link alla repository : *https://github.com/gianpics/ProgettoPAO2020*

Suddivisione dei compiti e ore di lavoro

Suddivisione dei compiti

- **Gerarchia e contenitore**: Gianpiero Tovo, Giacomo Sassaro, Ayoub Maher
- **Modello**: Gianpiero Tovo, Giacomo Sassaro
- **Vista**:
    - GUI : Gianpiero Tovo, Giacomo Sassaro, Ayoub Maher
    - Graphs: Giacomo Sassaro
    - Image Export: Gianpiero Tovo
- **Controller** : Gianpiero Tovo, Giacomo Sassaro
- **I/O su file** : Ayoub Maher

Suddivisione ore di lavoro

Mi sono occupato principalmente dello sviluppo della parte logica, quali gerarchia Stats, classi Account e Content e del contenitore StatsList. Ho lavorato anche alla parte grafica, in particolare a GraphsWindow e a QInfoDialog. Mi sono occupato di gestire il modello e la creazione dei grafici.

- **Ideazione progetto** : 4 ore
- **Progettazione gerarchie e gui** : 3 ore
- **Creazione repository e studio github** : 3 ore
- **Sviluppo parte logica** : 25 ore
- **Sviluppo parte grafica** : 10 ore
- **Prima revisione fix** : 7 ore
- **Debug e test finali**: 8 ore

  Totale : 60 ore