UOZAPP

PROGETTO DI PROGRAMMAZIONE AD OGGETTI BASATO SU

“QONTAINER”

NICOLA SALVADORE, 1143452

TEMPO IMPIEGATO

- ANALISI DEI REQUISITI E PROGETTAZIONE: 2 ore;
- PROGETTAZIONE E REALIZZAZIONE DI QONTAINER: 15 ore;
- PROGETTAZIONE E REALIZZAZIONE GERARCHIA: 5 ore;
- PROGETTAZIONE E REALIZZAZIONE MODELLO MCV: 20 ore;
- IMPLEMENTAZIONE METODI E FUNZIONI PER LA GUI: 5 ore;
- DEBUGGING E MIGLIORIE GRAFICHE: 6 ore;

Il tempo segnato non tiene conto dei "trial and error" tipici durante l'apprendimento di un linguaggio e di nuove librerie. Diverse parti di codice sono state cancellate in tronco e riscritte e le ore indicate si riferiscono alla realizzazione dell'ultima versione del codice

STRUMENTI UTILIZZATI

Il progetto è stato realizzato interamente su Windows con Qt Creator 4.9.1 con Qt 5.12.3.

Compilato su Windows con MinGW 7.3.0.

È stato utilizzato il debugger GNU gdb 8.1.

È stato testato sulla macchina virtuale fornita dal professore e risulta funzionante.

COMPILAZIONE ED ESECUZIONE

Il codice contiene elementi di c++11 (lambda espressioni e keyword come *nullptr auto*…), pertanto il comando *qmake -project* non crea un *.pro* adatto. Consiglio per questo di utilizzare il file *uozapp.pro* fornito alla consegna.

ContainerList<T>

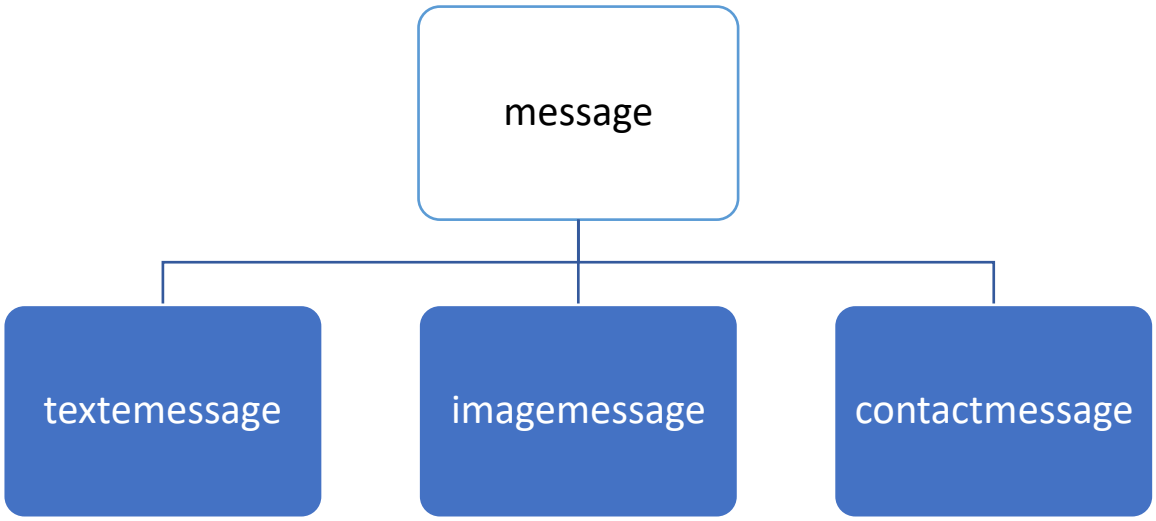
La classe *ContainerList* rappresenta un container templetizzato a lista senza puntatore smart (che era stato inizialmente inserito, ma poi rimosso), per semplicità di codice. È stata scelta la lista perché non avevo bisogno di un contenitore ad accesso casuale. La classe *nodo* contiene il dato e il puntatore al successivo e al precedente per rendere più veloci operazioni come la rimozione di un elemento.

Presenta come campi dati un puntatore all'inizio e uno alla fine della lista. Altre caratteristiche come ad esempio la *size* ho preferito lasciarla come metodo dato che nel MVC non viene utilizzata molto.

Copia profonda è a carico della funzione statica copy, mentre la distruzione profonda è delegata al distruttore.

È presente un *const_iterator* e un *iterator* definito come classe annidata amica per l'accesso sequenziale ai dati del container.

LA GERARCHIA



message è la classe astratta padre della gerarchia e contiene le informazioni di base di un messaggio nella applicazione uozapp, come il nome del mittente e quello del ricevente, la data di creazione inizializzata durante la costruzione con la data di sistema e, infine, un flag che indica se il messaggio è in entrata o in uscita. Mittente e Destinatario hanno valore di default *uknown* se non specificato.

textmessage rappresenta un messaggio di testo derivato da *message*. Contiene come campo dati proprio il campo text, non constante. C'è infatti il metodo *write* che reimplementa nella classe il metodo *append* di QString sul testo di textmessage.

imagemessage rappresenta un'immagine derivata da message. Come campi dati propri contiene l'URL dell'immagine e la descrizione, tutti e due di tipo QString.

contactmessage rappresenta un contatto telefonico da poter salvare su un'ipotetica rubrica. È composto da nome, cognome, nickname, numero di telefono e prefisso.

CHIAMATE POLIMORFE

Una funzione virtuale della gerarchia è *sendmex*, un metodo di clonazione che setta il messaggio clonato come in entrata se l'originale è in uscita. Nella classe *message* è contrassegnato come metodo puro (rendendola astratta). Nelle classi derivate il metodo utilizza il costruttore di copia standard di *message* (non serve la copia profonda di DateTime). Il metodo è invocato dal Controller nel momento di inviare il messaggio nelle funzioni *sendATMessage*, *sendAIMessage* e *sendACMessage*.

L'altra funzione virtuale è *similar*, che implementa un confronto tra un messaggio della gerarchia e una stringa, che ritorna vero se la stringa è contenuta nel testo (nel caso del *textmessage*), nella descrizione (nel caso dell'*imagemessage*) o all'interno di uno dei campi dati di *contactmessage*. La funzione è utilizzata nel metodo *searchThis* del modello, che si occupa di creare un vettore con tutti i messaggi che restituiscono vero da *similar*.

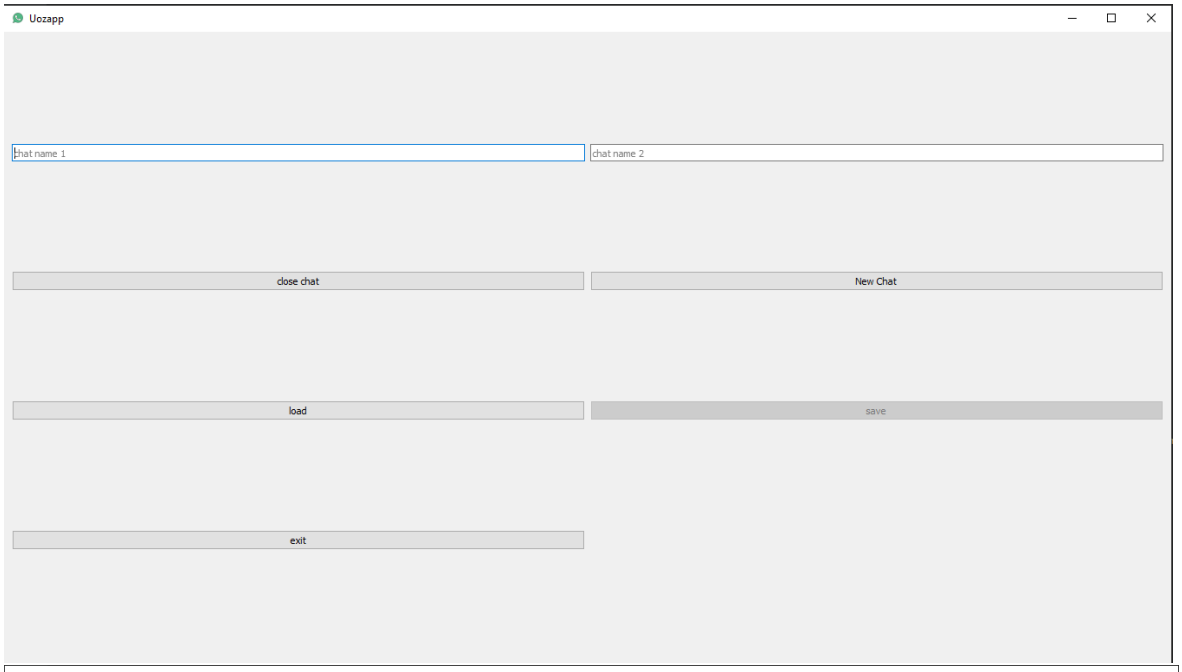
LA GUI

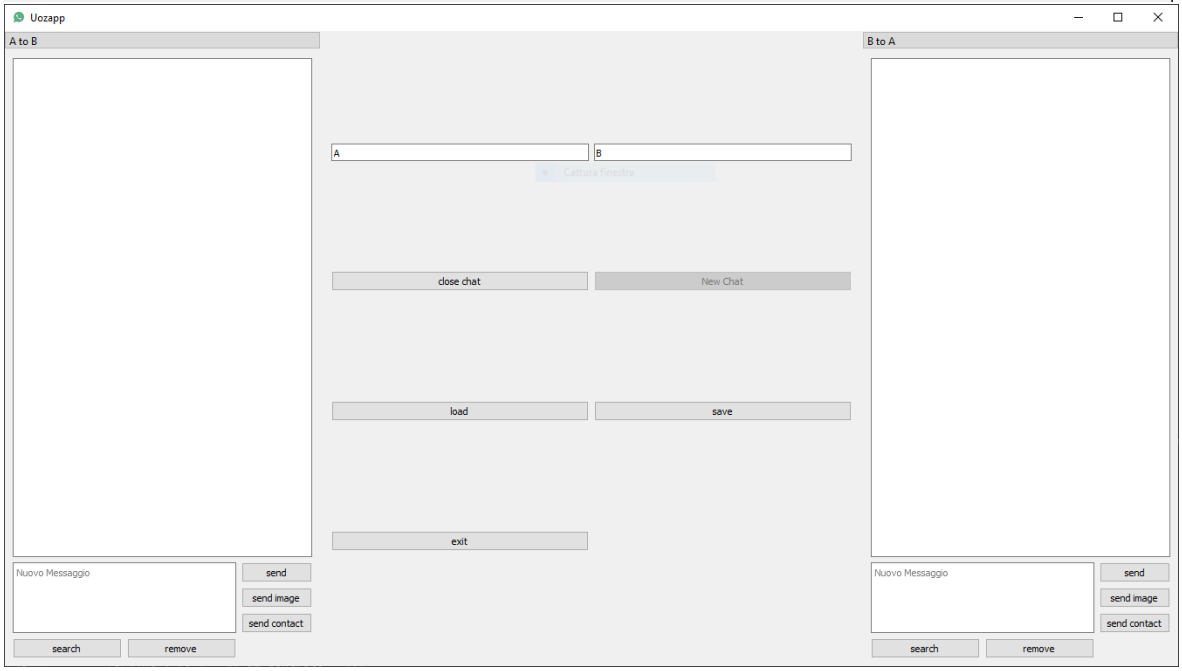
La GUI è stata costruita secondo il modello MCV, dove il modello è formata dal contenitore di messaggi all'interno dell'applicazione uozapp.

È stato scelto il modello MVC anziché l'MV, perché si sposa meglio con l'idea di chat dove idealmente le due viste sono in due dispositivi differenti e il controller con i due modelli associati nei server. Inoltre, è molto complicato, se non impossibile, gestire due viste e due modelli accoppiati che comunicano tra loro senza un controller centrale a mediare.

Il Controller è unico e contiene i due puntatori ai due modelli (che sono i due attori della chat) e i due puntatori alle viste e si occupa di fare comunicare i vari oggetti, oltre a contenere le informazioni di una chat come il nome dei due utenti. La vista, invece, è una derivata di *DockWidget* dato che è inserita all'interno di una mainwindow.

È proprio la ***mainwindow*** il cuore dell'applicazione dato che sarà la prima finestra ad apparire e a creare poi l'MVC.





Inserendo i nomi dei due corrispondenti nella *mainwindow* è possibile cominciare una nuova chat e si apriranno le due chat sottoforma di *DockWidget*.

La chat dà la possibilità di inviare un messaggio di testo digitando nella *QTextEdit* e premendo il pulsante *send*. Il pulsante *send image* permette di inviare un'immagine caricandola dal locale, l'eventuale descrizione va scritta nella *QTextEdit* prima di caricare l'immagine. Infine, *send contact* apre un widget dove è possibile inserire i dati del contatto.

FORMATO DEL FILE LOAD/SAVE

Per il salvataggio non è stato utilizzato nessun formato di altre librerie, ma ho voluto svilupparne uno utilizzando le librerie di output/input di con *QDataStream*.

Per ogni tipo di oggetto ho ridefinito infatti l'operatore di output, che stampa in sequenza i suoi campi dati. Quest'ultimo viene quindi chiamato da tutti i messaggi nel vettore contenuto nel Modello, separando uno dall'altro dalla stringa *continue*. Dato che nella mia versione i due modelli condividono lo stesso file di salvataggio c'è una stringa di *stop*, che indica il cambio di modello. Nel formato *.chf* viene quindi stampato i nomi dei corrispondenti e i due modelli.

La funzione di load segue lo stesso procedimento nello stesso verso tramite l'operatore di input utilizzando sempre il formato QDataStream. Nel modello una volta letto tutti i dati di un messaggio viene chiamato un suo metodo *sendmessage* che si occupa di caricarlo nel container e stamparlo a video.

L'operatore di output poteva essere ridefinito nel modello e non nella gerarchia attraverso i metodi di get, ma per non incorrere a problemi dovuti all'incapsulamento ho preferito posizionarlo lì. Quello di input invece è stato ridefinito direttamente nel modello.

Ho riscontrato un po' di difficoltà con il tipo char del prefisso e del numero telefonico di *contactmessage* probabilmente con più tempo sarei riuscito a sistemarlo, potrebbe darsi che nella visualizzazione compaiano quindi caratteri non codificati su quei campi dati.

SEARCH E REMOVE

Il pulsante della GUI search fornisce un tool di ricerca di un messaggio. Il controller chiama il metodo *searchThis* del modello già descritto sopra nelle chiamate polimorfe. Il vettore di messaggi trovati viene ritornato e visualizzato nel widget.

La remove potenzia il tool precedente, dato che prima di rimuovere il messaggio questo va cercato. Infatti, una volta trovati e visualizzati i messaggi sarà possibile rimuoverli. In questo caso per comodità ho sviluppato un metodo della vista che va a fare un refresh della chat (cioè viene cancellato e ricaricato il testo nella QTextBrowser) con i messaggi effettivamente contenuti nel modello. La rimozione di un messaggio su una chat è indipendente dall'altra, infatti se viene eliminato per A questo rimane nella chat B.