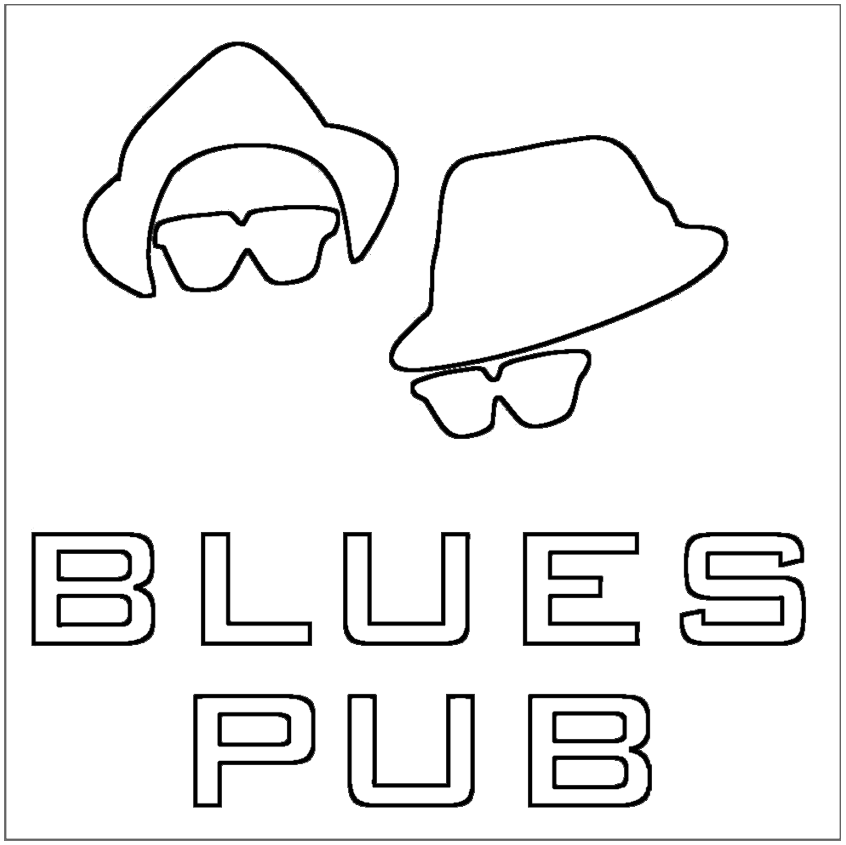
Programmazione ad Oggetti

a.a. 2016/2017

Relazione del progetto

Sara Feltrin – Mat. 1122453



Sara Feltrin – 1122453 – Progetto di programmazione ad oggetti *Università degli studi di Padova – Laurea in Informatica*

INDICE

1 - Scopo del progetto

L'idea del progetto BluesPub è la creazione di un'applicazione per la gestione delle ordinazioni all'interno dell'omonimo locale. Il cliente, comodamente da casa, può consultare il menù e vedere la vasta selezione di bevande e pietanze che il BluesPub può offrire e, dopo aver consumato un pasto, potrà valutare la propria esperienza, assegnando un punteggio (da 1 a 5) per il servizio e il cibo ricevuti. Effettuando il login, i dipendenti, invece, possono prendere le ordinazioni e calcolare il conto totale in un determinato del tavolo. Tali azioni sono disponibili anche per il titolare che, in più, ha la possibilità di modificare il listino aggiungendo, rimuovendo bevande e piatti o semplicemente cambiando nome, descrizione e/o prezzo di quelli già presenti. Un titolare, inoltre, può consultare la media delle recensioni lasciate dai clienti e, nel caso ci siano modifiche al personale, aggiungere i nuovi dipendenti assunti e rimuovere quelli licenziati.

2 - Descrizione delle gerarchie

Il progetto si compone di cinque gerarchie principali: vivande, utenti, database, controller e view.

2.1 Gerarchia vivande

La gerarchia delle vivande prevede alla radice la classe astratta *ListItem* da cui derivano le classi concrete *Food* e *Drink*. Nella base sono presenti i campi dati che definiscono una vivanda generica, i metodi per impostare tali campi dati e quelli pubblici per permetterne la lettura. Viene inoltre definito il metodo virtuale puro *load* che, implementato nelle sottoclassi concrete, permette di caricare dal database i dati di uno specifico item e uno virtuale *save* utile al salvataggio dei dati.

Le varie categorie di utenti dell'applicazione possono interagire in maniera differente con questi tipi di oggetti appartenenti al listino.

ListItem prevede al suo interno l'enumeratore *Type* per specificare la tipologia di vivanda (Bruschetta, Toast, Birra, ...). Questo particolare attributo è utilizzato principalmente per categorizzare le vivande nel listino. Tale scelta implementativa è stata fatta perché le differenti tipologie non presentano alcuna differenza sostanziale, infatti mantengo gli stessi campi dati e metodi. Quindi si è preferita questa strada rispetto alla creazione di una gerarchia più complessa che prevedesse una classe per ognuna delle categorie definite da *Type*. In futuro sarà possibile estendere la gerarchia delle vivande estendendo le classi concrete già esistenti e impostando automaticamente la categoria (es. creando una sottoclasse *Toast*, sottotipo di *Food e* impostare il tipo della vivanda a *ListItem::Type::TOAST*, senza richiedere il tipo nella costruzione dell'oggetto).

La base astratta prevede, inoltre, la ridefinizione dell'operatore di uguaglianza. Tale metodo è marcato come *virtual* , in modo tale che possa essere ridefinito nelle sottoclassi e che permetta il confronto tra differenti tipi di oggetti presenti nella gerarchia.

In *ListItem* troviamo tutti i metodi *setter* per i vari campi dati, marcati con visibilità *protected*: tale scelta deriva dall'idea che solo il metodo *load*, ridefinito nelle sottoclassi, possa accedervi, settando i valori presenti nel database come valore degli attributi di un oggetto. È comunque presente un costruttore che fornisce la possibilità di creare oggetti con i valori desiderati.

2.2 Gerarchia utenti

La gerarchia degli utenti prevede alla radice la classe astratta *User* da cui derivano le classi concrete *Client* e *Waiter.* estesa a sua volta dalla classe *Holder*. La classe base contiene i dati che definisco il singolo utente e possiede i relativi metodi accedere e modificare tali informazioni. *User* espone il metodo virtuale puro *getType()* , che restituisce la tipologia a cui appartiene l'utente su cui viene invocato.

Nel dettaglio:

Client	• Vedere il menù • Valutare servizio e consumazione • Modificare i propri dati • Eliminare il proprio account
Waiter	• Prendere un'ordinazione • Fare il conto
Holder	• Prendere un'ordinazione • Fare il conto • Vedere la medie delle valutazioni date dai clienti • Modificare il listino • Aggiungere dipendente • Rimuovere dipendente

Quindi i *Client* hanno permessi di accesso al database solamente in lettura, per vedere il listino del pub. Anche i *Waiter* possono accedervi in sola lettura, ma per uno scopo differente: infatti possono aggiungere le ordinazioni e successivamente rimuoverle chiudendo il conto. Gli *Holder*, invece, hanno accesso in lettura e scrittura, con la possibilità di aggiungere nuovi cibi e bevande al listino e eliminare o modificare quelle preesistenti.

La registrazione di nuovi *Client* può essere fatta semplicemente accedendo all'applicazione, mentre quella di nuovi *Waiter* può essere fatta solamente da un *Holder*: tale scelta è stata fatta pensando ad un contesto dove c'è solo il datore di lavoro che può aggiungere i nuovi dipendenti al sistema. Con la stessa idea, è stato dato all'*Holder* la capacità di rimuovere un *Waiter*. Non è stata creata una registrazione per nuovi *Holder* invece, pensando che questi siano aggiunti direttamente al database, alla creazione dell'applicazione, poiché è meno facile che questo tipo di utenti aumenti nel tempo. Solo per il *Client* è stata creata l'interfaccia per la modifica dei dati personali, ma le relative classi sono state create pensando alla possibilità che possano essere riutilizzate per anche gli altri tipi di utente.

Dati per l'accesso:

- **Client** username: *cliente* password: *cliente*
- **Waiter** username*: cameriere* password: *cameriere*
- **Holder** username*: titolare* password: *titolare*

I vari tipi di utenti non presentano metodi differenti (tranne quello che identifica il tipo), poiché tale gerarchia è pensata per contenere i dati degli utenti. Le varie classi differenziano i controller e le view che verranno create a seconda del tipo, in quanto la logica dell'applicazione e di ciò che ogni tipologia di utenti può fare è spostata sui controller.

2.3 Gerarchia database

La gerarchia database prevede alla radice la classe astratta *DbXml* da cui derivano le seguenti classi concrete: *db_user*, *db_list*, *db_table* e *db_valuation*. Si è deciso di creare questa gerarchia per racchiudere nella classe astratta tutti i metodi comuni alla gestione dei diversi database, in quanto tale gerarchia è stata pensata principalmente per il riuso del codice.

In *DbXml* è presente il metodo virtuale *save_db()* che viene implementato dalle diverse classi derivate e con il compito di salvare su file tutti i dati presenti nel database.

La classe *db_user* si occupa dei dati degli utenti. Si è scelto di caricare tali dati all'interno di una mappa, con chiave username, identificativo univoco dell'utente all'interno dell'applicazione. La scelta della mappa è stata fatta poiché semplificava le fasi di accesso e registrazione: infatti, grazie al nome utente è immediato il controllo dei dati di login o se un certo nome utente è già stato inserito nel sistema. Nella classe sono inoltre definiti i metodi di verifica per l'accesso, come

checkLogin() e *findUsername()* , che hanno il compito di verificare che all'interno del database sia presente lo username inserito e che la password corrisponda a quest'ultimo, e per la registrazione, come *checkExistingUsername()* che verifica se un certo nome utente è già stato scelto da altri utenti.

In *db_list* vengono implementati i metodi per la gestione degli oggetti che vanno a popolare il listino. L'insieme degli oggetti è gestito da un vettore e tale scelta è stata fatta poiché il vettore permette sia l'accesso randomico, utile per la rimozione e la modifica, sia di scorrerlo sfruttando un iteratore, sia di aggiungere oggetti in coda in maniera semplice.

La classe *db_table* espone i metodi per la gestione dei tavoli e delle relative ordinazioni. Per una migliore gestione degli ordini, i tavoli sono memorizzati in una mappa, dove la chiave è il numero del tavolo, mentre tramite il valore è possibile accedere alle informazioni sulle ordinazioni associate ad un certo tavolo .
Tutti i dati vengono salvati utilizzando il formato XML. Anche in questo caso la scelta è stata fatta per semplificare le operazioni di lettura e scrittura, che sono state implementate sfruttando le classe della libreria Qt *QXmlStreamReader* e *QXmlStreamWriter*.

2.4 Gerarchia controller

All'interno del progetto si è cercato di seguire il più possibile il pattern MVC separando l'interfaccia grafica, la logica dell'applicazione e i dati dell'applicazione. Per questo motivo troviamo le classi con nomi che hanno come suffissi "controller" e "view", i quali indicano appunto a che categoria di classi appartengono.

I controller all'interno del progetto fanno parte di una gerarchia, creata allo scopo di gestire in modo più semplice possibile il cambio di pagina dell'applicazione. La classe base è *Controller,* dalla quale derivano tutte le classi con suffisso "controller", che contiene la segnatura del segnale *changesignal* , emesso dai vari controller quando l'utente esegue un'azione che provoca il cambio di pagina. Questo è un segnale che richiede di specificare la nuova view da mostrare (che deve essere sottoclasse di *QWidget*) e il nuovo controller che la gestisce. Tale segnale verrà catturato dal *MainController,* il controller che effettivamente si occuperà del cambio dell'interfaccia grafica e dell'annesso gestore.

In questa classe troviamo anche i metodi con i quali i controller possono ricavare i puntatori ai vari database, in modo tale che possano accedere ai dati che le varie "view" hanno il compito di mostrare.

2.5 Gerarchia view

Tutte le classi dell'interfaccia grafica, ovvero quelle con suffisso "view", derivano dalla classe *QWidget*, offerta da Qt. Ognuna si occupa della visualizzazione dei dati in una particolare schermata dell'applicazione. La maggior parte sono state create sfruttando *Qt Designer*, ad eccezione di quelle che si interfacciano con il listino, poiché sono pagine più "dinamiche". In questo casi si è preferito creare l'interfaccia scrivendone il codice manualmente, dal momento che i dati da mostrare cambiano a seconda del contenuto del database e possono cambiare in seguito all'interazione con l'utente.

3 - Descrizione dell'uso del codice polimorfo

Il polimorfismo all'interno dell'applicazione è stato utilizzato in vari punti. La maggior parte si trova nel salvataggio e nel recupero di dati. Il database degli utenti sfrutta il metodo polimorfo *getType()* per salvare l'informazione riguardante il tipo di utente da memorizzare o da caricare. In modo dissimile, invece, tali operazioni vengono effettuate dal database del listino. Infatti la classe *ListItem* espone i metodi virtuali per il salvataggio e il recupero dei dati, ridefiniti da *Food* e *Drink* e sfruttati da *db_list*. Quest'ultima possiede come membro di istanza un vettore di *ListItem* e quindi per effettuare il salvataggio e recupero dei dati su file invocherà *load* e *save* che, a seconda del tipo concreto, saranno implementate in maniera differente. Le operazioni di *db_list* vengono aiutate dai metodi virtuali di *ListItem* come *clone* e *operator==*.

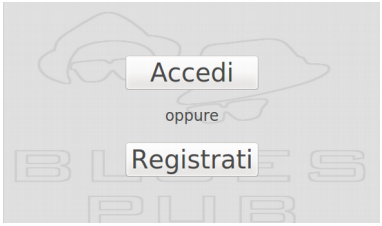
La classe *ListItem* è stata inoltre utilizzata nella gestione degli ordini e per il calcolo dei totali: poiché non è necessario sapere il tipo esatto dei vari elementi del listino è stato possibile sfruttare solamente i metodi della classe base astratta.

Il polimorfismo è utilizzato anche dalle sottoclassi di *DbXml*. In questa classe infatti è presente il metodo virtuale puro *save_db*, che ogni sottoclasse ridefinisce, in modo da permettere di salvare tutti i dati in maniera corretta.

4 – Compilazione ed esecuzione

Il progetto per essere compilato ed eseguito correttamente richiede il file *BluesPub.pro*: per la compilazione è necessario, quindi, dare il comando *qmake BluesPub.pro*, seguito dal comando *make*. Per l'esecuzione, in laboratorio, ultimata la compilazione, è sufficiente quindi dare da terminale il comando `./BluesPub` . Per il corretto funzionamento l'applicazione necessita dei file *databaserate.xml*, *databasetavoli.xml*, *databaseutenti.xml e listino.xml*. Nel caso in cui questi non siano presenti, il programma funzionerà comunque, in quanto è progettato per creare questi file, ma, ovviamente, non ci saranno dati per la visualizzazione del listino o per effettuare il login.

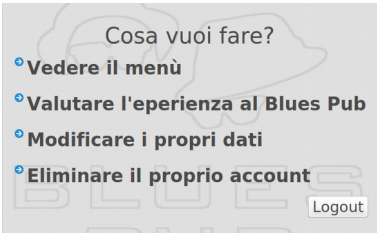
5 – Manuale utente



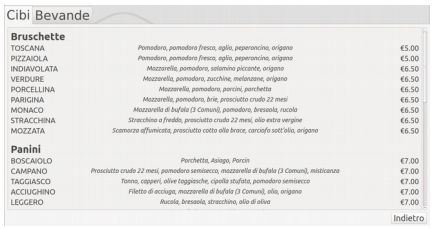
La schermata iniziale dell'applicazione chiede all'utente se vuole effettuare il login o registrarsi, nel caso non sia provvisto dei dati per l'accesso. A seconda della scelta verranno mostrate le schermate qui sopra: la prima nel caso in cui l'utente scelga di accedere, la seconda nel caso in cui voglia procedere alla registrazione.

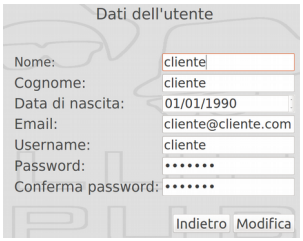
5.1 Utente

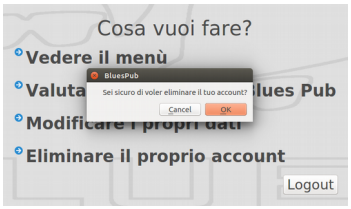
Nel caso venga effettuato con successo il login con i dati di un utente verrà mostrata la seguente schermata:



A seconda della scelta fatta verranno visualizzate le seguenti pagine:



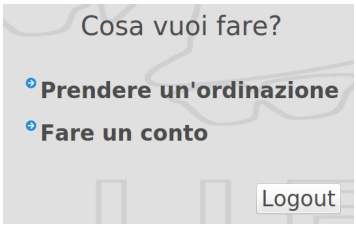


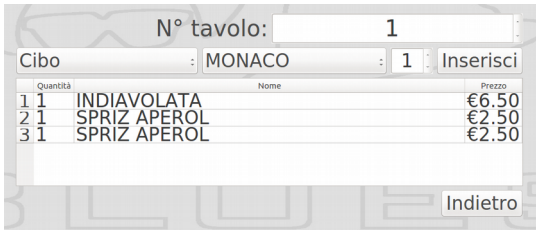


In particolare nella schermata di modifica dei dati è possibile riscrivere i propri dati (per rendere la modifica effettiva devono essere tutti immessi e validi, altrimenti verranno visualizzati dei messaggi di errore), mentre se si vuole eliminare il proprio account verrà visualizzato un pop-up, mostrato nell'ultima immagine a destra, che chiede la conferma per l'operazione.

5.2 Cameriere

Nel caso venga effettuato con successo il login con i dati di un cameriere verrà mostrata la prima delle schermate di seguito.



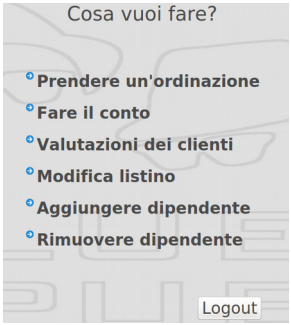


Per prendere un'ordinazione ad un certo tavolo, è necessario inserire il numero del tavolo nella parte in alto della schermata e successivamente selezionare il tipo di vivanda (cibo o bevanda). Fatto ciò appariranno i nomi associati alla tipologia di vivanda selezionata nella combo box a fianco. Infine è possibile specificare la quantità. Quando viene premuto il tasto inserisci, l'ordine verrà mostrato nella tabella al di sotto delle precedenti combo box.

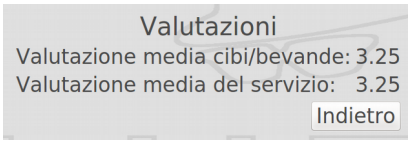
Per chiudere un conto è sufficiente inserire il numero del tavolo per mostrare le ordinazioni ad esso associate con il totale dovuto e premere il bottone "Chiudi conto".

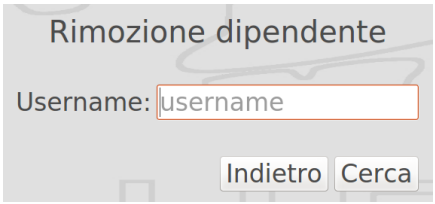
5.3 Titolare

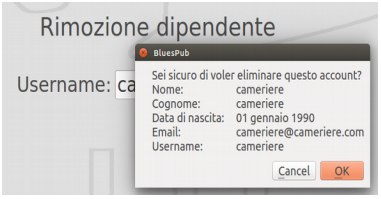
Nel caso venga effettuato con successo il login con i dati di un titolare verrà mostrata la seguente schermata.

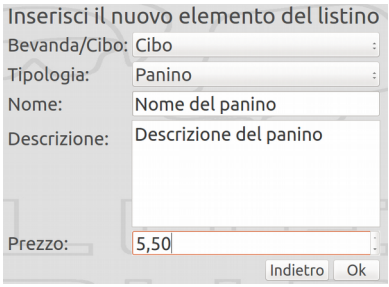


Le schermate per prendere un'ordinazione e fare il conto sono le stesse del cameriere, come quella per registrare un nuovo dipendente è uguale a quella per la registrazione di un cliente. Le altre schermate sono riportate a seguire.









Nel caso si voglia eliminare un dipendente è necessario inserire il suo username all'interno della casella di ricerca. Qualora venga trovato un cameriere con tale nome utente ne verranno mostrati i dati in un pop-up (un esempio è riportato sopra) nel quale viene richiesto se cancellare i dati del cameriere. Nella schermata di modifica del listino vengono mostrare tutte le informazioni delle varie vivande e per fare una modifica è sufficiente posizionarsi nella casella e modificare il campo del cibo o della bevanda e, successivamente, premere invio per rendere effettiva la modifica. L'aggiunta di una nuova vivanda richiede invece la corretta compilazione del form riportato nell'ultima figura in alto.

6 – Indicazioni conclusive

6.1 Problemi noti

La scelto di utilizzare le classi *QXmlStreamReader* e *QXmlStreamWriter* per la gestione dei file XML che permettono la persistenza dei dati dell'applicazione da un lato ha permesso una gestione della scrittura e della lettura dei file semplice, dall'altro però tali classi non permettono di effettuare la modifica di un file. Ciò implica una modifica dei dati che comporti una riscrittura completa del file. Questo problema è stato mitigato nel caso di aggiunta di un nuovo dato al file XML, posizionando l'oggetto *QXmlStreamWriter* alla fine del file (ciò è implementato in *DbXml*). Questo comporta inevitabilmente un calo nell'efficienza e, nel caso della rimozione di un elemento del listino da parte di un *Holder*, è possibile notare un "freeze" della GUI per qualche istante. È stato deciso comunque di mantenere tale implementazione per questioni di semplicità e tempo.

Nel progetto è stata creata una classe *Utility*, che espone solamente metodi statici di utilità. È stata implementata in questo modo poiché non aveva senso che tali metodi appartenessero alle altre classi del progetto, in quanto implementano solamente alcune funzioni generiche sulle stringhe.

6.2 Tempo di lavoro

Il tempo di lavoro richiesto ha superato non di molto le ore previste dal progetto. Questo perché, durante lo sviluppo, alcune scelte progettuali non si sono rivelate adatte allo scopo del progetto, dal momento che non erano in linea con le specifiche del progetto e con i criteri di valutazione.

Progettazione:	4-5	h.
Implementazione:	52-53	h.
Debugging e miglioramenti:	8-9	h.

Per un totale di circa 66h.

6.3 Ambiente di sviluppo

Sistema operativo:	Windows 10
Versione Qt:	Qt Creator 4.3.1 Based on Qt 5.9.1
Compilatore:	Qt 5.9.1 MinGW 32bit