MC BURGER

studente: Elvis Murtezan

matricola: 1096758

anno scol: 2019/2020

Indice

1. Scopo Progetto

Il progetto ha come scopo la creazione di un applicativo finalizzato alla gestione di un fast-food nelle sue varie funzionalità ovvero la gestione di un ordinazione dalla richiesta al completamento di essa . La versione implementata prevede un interazione (vocale) col cliente che effettua degli ordini, e i dipendenti che gestiscono tali ordini.

2. Funzionalità implementate

L'applicativo prevede 3 tipologie di utenti che sono i dipendenti: Cassiere, Cuoco, e Manager.

Nello specifico ogni dipendente avrà a disposizione un'area riservata dopo essersi autenticato,e da cui potrà in base al ruolo:

il **cassiere**:

- Crea degli ordini
- Conferma/annulla ordini

il **Cuoco** avrà le seguenti funzionalità principale:

- Conferma ordini in preparazione

il **Manager** ha le funzionalità sia di **cuoco** che di **cassiere** oltre ai seguenti compiti:

- modifica la disponibilità dei prodotti disponibili;
- avvia un'istanza di cucina (apre la GUI della cucina)

Il **cliente**, a cui viene associato un identificativo numerico, potrà invece visualizzare gli ordini pronti/in preparazione tramite uno schermo **(simulato con un widget)**

3. Progettazione

3.0 Progettazione architetturale

Per sviluppare l'applicazione, ho deciso di optare per un'architettura di tipo Model-View-Controller, perche' è un pattern molto utilizzato e volevo avere un idea di come le cose debbano essere fatte con questo pattern.

3.1 Analisi gerarchia principale

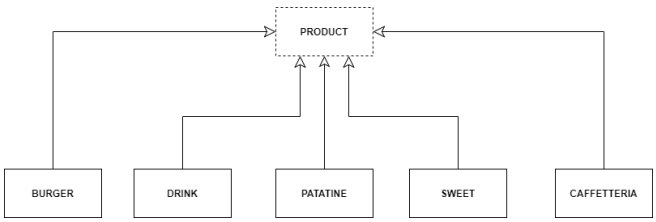

Figura 1: rappresentazione della gerarchia di Product (classe astratta)

Premessa: al fine di rappresentare il più fedelmente possibile una situazione reale si sono individuate 5 classi concrete di prodotto che derivano dalla classe base astratta **Product. (**

figura 1). Tale classe ha dei campi dati standard come il nome, il prezzo, la categoria, size, ecc. con relativi metodi di get() set(), e un costruttore che aiuta le classi concrete a inizzializzare campi dati per diverse funzionalita'.

Burger: rappresenta un panino con i suoi ingredienti

Drink: rappresenta tutte le bibite con possibilita' di selezionare una size tra small, medium, and tall

Patatine: rappresenta le varie tipologie di patatine

Sweet: rappresenta le varie tipologie di Dolci

Caffetteria: rappresenta oggetti come caffè,spremute,...

Nonostante molte classi possono sembrare molto simili alla classe base Product si è scelto non rendere Product concreta perchè comunque troppo generica e in un eventuale aggiunta di nuovi campi dati bisognerebbe andare a modificare tale classe ,cosa che creerebbe una dipendenza grave ed innefficente.

i metodi virtuali offerti da **Product** sono:

-**clone():** classico metodo di copia polimorfa molto utile in vari casi

-**readinfofromJSON():** lettura polimorfa dei campi dati da un oggetto JSON per inizializzare il prodotto con valori di default dentro il database

-**operator==():** utile per il confronto se un dato prodotto è già presente nel carello

3.2 class Employee

La creazione di questa gerarchia deriva dalla necessita di distinguere i vari ruoli di ciascun dipendente (**figura 2).**

La derivazione a diamante di Manager è dovuta dal fatto che il manager deve avere le funzionalità offerte sia dal Cassiere che dal Cuoco in quanto il manager è una sorta di tutto fare/figura di riferimento del fastfood. Le classi concrete, rappresentanti le tipologie di utente, derivano dalla classe astratta **Employee**.

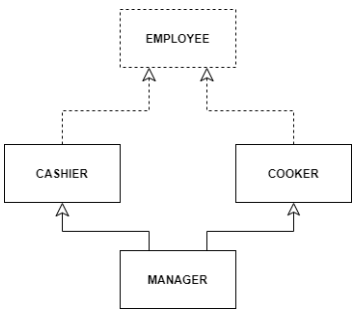

figura 2 rapp.grafica gerarchia Dipendenti

I metodi virtuali offerti dalla classe **Employee** sono:

- **clone():** utilizzato dalla classe Database (lettura da file)
- **readinfofromJson()**: metodo di inizializzazione dei campi dati da un oggetto JSON
- **getincome():** metodo che calcola lo stipendio in base a certi criteri.

Inoltre ciascuna classe ha dei metodi relativi alla funzione svolta come per esempio il cambio di stato di un ordine eseguito dal

cuoco e dal cassiere, o il cambio di disponibilita' di un prodotto
fatto dal manager.

3.3 Classe contenitrice Dlist

E' stata appositamente creata una classe contenitrice templetizzata: Dlist, implementata sotto forma di lista doppiamente linkata. Lo scopo principale è quello di far fronte alle mansioni di insert, delate, search e di implementare una gestione della memoria senza condivisione (deep memory).

Ovvero:

- un puntatore al primo e all'ultimo nodo di questa lista: classe annidata nodo –> first e last;
- vari metodi allo scopo di facilitare l'accesso e l'inserimento in coda e in testa alla lista: begin(), end(), push back...;
- costruttore che inizializza una lista vuota;
- ridefinizione di costruttore di copia, assegnazione e distruzione in modalità profonda;
- presenza sia di un iteratore costante che non,dentro alla classe (nested template);
- metodi come size() che mi fornisce la lunghezza della lista o get() che mi restituisce la lista stessa;
- ridefinizione di alcuni operatori: operator[], operator++, operator--, operator*, operator→, operator!=, operator== standard delle classi "container".

3.4 Classe Ristorante(Restorant)

Classe che mi rappresenta il fastfood in quanto entità fisica,che presenta:

- un contenitore di tutti i prodotti presenti nel FastFood;
- un contenitore dei dipendenti (Dlist)
- un contenitore delle ordinazioni.
- un costruttore di default e vari metodi concreti con le finalità di manipolare le informazioni del FastFood, aggiornare il menù, aggiornare le ordinazioni, convertire le informazioni per successivamente salvarle nel database(file JSON) e comunicare con le varie view dell'interfaccia.
- all'avvio dell'applicazione, chiama alcune funzioni statiche della classe Database per costruire i prodotti venduti dal ristorante e i dipendenti.

3.5 Class Order

Questa classe rappresenta un ordine effettuato da un cliente alla cassa, gestisce i prodotti ordinati dalla sua creazione(costruttore) al completamento di esso, tramite il cambiamento di stato effetuato dal cuoco e dal cassiere ,tramite i metodi offerti dalla classe.

4. Il Database

Il database e' rappresentato dalla cartella Database che contiene file JSON che hanno informazioni salvate relative ai prodotti venduti dal Fastfood e relative ai dipendenti. La classe **Database** e' una classe di aiuto per la lettura dei file Json presenti nel database, tramite funzioni statiche.

5. Interfaccia Utente

Per quanto riguarda l'implementazione della GUI si è cercato di utilizzare varie classi offerte da Qt, facendo eventualmente dei subtyping necessari per funzionalita' specifiche al ristorante. L'avvio dell'applicazione presenta la finestra di Login, dove il dipendente dovra' inserire le sue credenziali di accesso per accedere alla finestra che gli e' riservata.

I widgets vengono costruiti una sola volta con parente, e alcuni di questi contengono puntatori a classi astratte del modello o del controller, al fine di una comunicazione tra le parti con dipendenze lasche (uso di metodi delle classi astratte).

Finestra del manager

1. **il menu in alto (tasti blu):** ripristina il menu principale quello (figura3),apre la gestione dei ordini,visualizza il carrello,avvia una finestra di cucina se non è già presente solo manager(un cuoco non ha loggato)
2. **bottoni di sinistra:** filtrano i prodotti secondo la categoria che dice di rappresentare il bottone
3. **bottoni a destra:** filtrano i prodotti secondo certi parametri

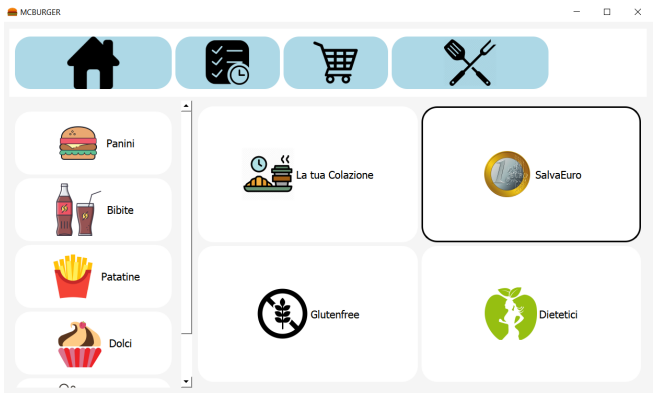

figura 3:GUI manager(menu principale)

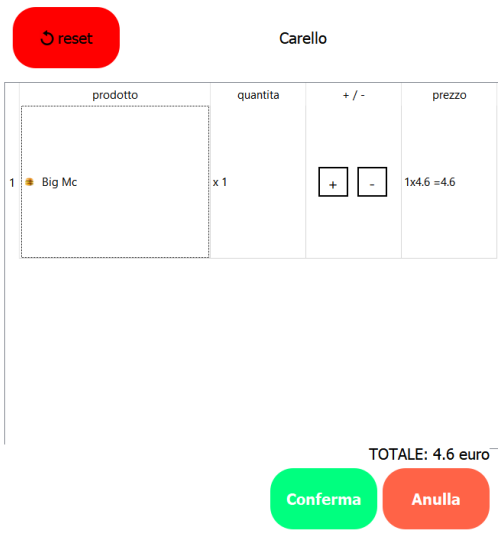

figura 4: GUI carello

Altre azioni:

Cliccato un bottone come quello in **figura 5** si attiva una finestra di dialogo che: ci farà selezionare la quantità e nel caso di prodotti con size si può scegliere anche una size **figura 6.**

Nel caso di manager con un click destro si disabilità la possibilita di selezionare il relativo prodotto/bottone.

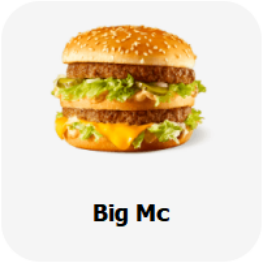

Figura 5: Bottone prodotto

Figura 6: finestra dialogo aggiungi prodotto

Finestra del Cassiere

-Per **cassiere** la UI è molto simile alla UI di Manager in quanto ha le stesse funzionalità del manager escludendo l'avvio di una cucina,la possibilità di "preparare" un ordinazione e la possibilità di disabilitare la disponibilità di un prodotto.

Finestra del Cuoco

-Per il **cuoco** la UI è molto simile al widget relativo alla visualizzazione dei ordini di manager senza la parte dei ordini pronti.

-UI vista dei clienti **figura 7** è il display che verrebbe visto dai clienti in attesa di ricevere la loro ordinazione,uguale a quella del manager ma senza funzionalità.

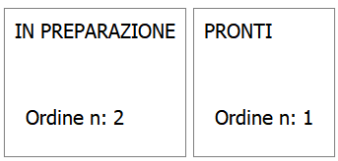

Figura 7 UI interfaccia per i clienti

Le modifiche(esempio gli ordini fatti,la disponibilità di un ordine non vengono salvate) non rimangono salvate perchè le funzioni di scrittura su file(update/write) non sono state implementate in quanto non era strettamente necessario, ovvero chiudendo l'app si ripristina il tutto.

6. Tempo di Lavoro

Le parti che nel complesso hanno richiesto maggiore dispendio temporale sono state quelle relative:

- allo sviluppo delle classi del modello, tale da essere il più flessibile possibile, visto anche il grande numero di tipologie di prodotti e operazioni che in un futuro potrebbero andare ad ampliare la gerarchia. Più volte infatti è stata necessaria una rivisitazione per capire bene cosa andasse implementato.
- allo sviluppo delle classi della view, per effetto soprattutto del numero di viste che poi si è andato a creare. Si è cercato di fare in modo che la comunicazione fra queste ultime e il modello fosse il più semplice ma efficiente possibile, si è reimplementato sfruttando alcuni widget già creati in precedenza non ad uso esclusivo (ex GenericOrderItem).

Nel complesso si sono spese un numero di ore leggermente superiore a ciò che inizialmente era stato preventivato, soprattutto a causa della mancanza di conoscenze iniziali della libreria Qt che ha richiesto il suo tempo per essere compresa.

Stime ore totali impiegate

Aa Attività	# Ore Svolte
Analisi del Problema	6
Progettazione	4
Codifica e implementazione del modello gerarchia	17
implementazione e codifica GUI (incluso apprendimento Qt)	31
Test Generali e Operazioni di Debug	6
Stesura relazione	3
ORE TOTALI	70

7. Credenziali di Accesso

username: manager **password:** manager **NB: loggare con manager per testare l'intera app**

username: cashier **password:** cashier ***NB: non permette la costruzione di un ordine nel caso non ci sia un cuoco attivo***

username: cooker **password:** cooker ***NB: il cuoco non può fare ordini finché non riceve ordini da un cassiere***

8. Ambiente di Sviluppo Utilizzato

SO: Windows 10 Pro 64bit , **versione Qt framework:** 9.5 **compilatore** MINGW -32bit

9. Compilazione ed esecuzione dei file sorgente

La gerarchia delle varie directory è la seguente:
MCBURGER è la cartella contenente il progetto.

che a sua volta si compone delle cartelle:

- Modello: contenente tutti i files .cpp e .h riguardanti l'implementazione del modello;

- View: contenente tutti i files riguardanti le viste
- Control: contiene il fle .cpp/.h del controller utilizzato
- Resources: contenente tutte le immagini e file css che l'applicazione usa.

e dei files:

- main.cpp;
- MCBURGER.pro
- relazione.pdf contenente la relazione richiesta.

E' stato fornito il file MCBURGER.pro in quanto si necessita di includere i moduli di QWidget.

10. Conclusioni

Questo progetto mi ha dato conoscenze e competenze sul framework Qt che non avevo in passato, mi ha anche permesso di mettere in pratica la programmazione ad oggetti. In particolare con un po di tempo in più avrei implementato la classe Menu nel suo complesso con le relative GUI, e le funzioni di aggiornamento e scrittura su file in modo da garantire le modifiche una volta spenta l'applicazione.