Progetto “Erboristeria”

RELAZIONE PROGETTO PAO

AA 2019/2020 | Zuccolo Giada | matr. 1193485

Sommario

INTRODUZIONE

Questo progetto è stato pensato per fornire ad un impiegato all’interno di un’erboristeria, un’applicazione in grado di gestire i clienti (nuovi e vecchi), i loro ordini e i loro punti, guadagnati con i vari ordini.

SPECIFICHE DI SVILUPPO

Il progetto è stato sviluppato con:

- Sistema operativo: Ubuntu 18.04.3 LTS (macchina virtuale fornita)
- Versione Qt Creator: 4.5.2, basato sulla versione Qt 5.9.5.
- Compilatore gcc 7.3.0

COMPILAZIONE

Per la compilazione, viene fornito il file progetto.pro. Da terminale deve essere usato per il comando qmake per generare il file Makefile, e poi il comando make, per compilare il progetto. Per poi poterlo eseguire, da terminale bisogna digitare ./progetto.

SUDDIVISIONE LAVORO

Il progetto è stato sviluppato con Sofia Chiarello (matr. 1187024). Insieme abbiamo analizzato il problema dato, ideato e poi e progettato il modello. Insieme abbiamo quindi curato la gerarchia e deciso cosa doveva fare ogni classe e quindi organizzare tutte le varie funzioni interconnesse. Abbiamo poi curato insieme l'idea grafica e quindi come gestire l'interfaccia. Successivamente abbiamo lavorato singolarmente ma contemporaneamente, suddividendoci il lavoro così:

Io	
integratori.h	integratori.cpp
fluidi.h	fluidi.cpp
sciroppobalsamico.h	sciroppobalsamico.cpp
goccetranquillanti.h	goccetranquillanti.cpp
bevande.h	bevande.cpp
tisana.h	tisana.cpp
lineate.h	lineate.cpp
cliente.h	cliente.cpp

Sofia	
prodotto.h	prodotto.cpp
apistica.h	apistica.cpp
miele.h	miele.cpp
pappareale.h	pappareale.cpp
capsule.h	capsule.cpp
propoli.h	propoli.cpp
echinacea.h	echinacea.cpp
ordine.h	ordine.cpp

I file che abbiamo creato insieme sono stati: handler.h e handler.cpp, home.h e home.cpp, clientela.h e clientela.cpp. Abbiamo strettamente collaborato e progettato insieme inserisciordine.h e inserisciordine.cpp, inseriscicliente.h e inseriscicliente.cpp, mostraclienti.h e mostraclienti.cpp. Infatti, abbiamo progettato insieme l'idea e poi io mi sono occupata della pratica ideazione e creazione degli oggetti grafici necessari e dell'aspetto prettamente estetico delle tre pagine, mentre Sofia si è occupata della modellazione ovvero di come collegare la parte grafica al modello progettato.

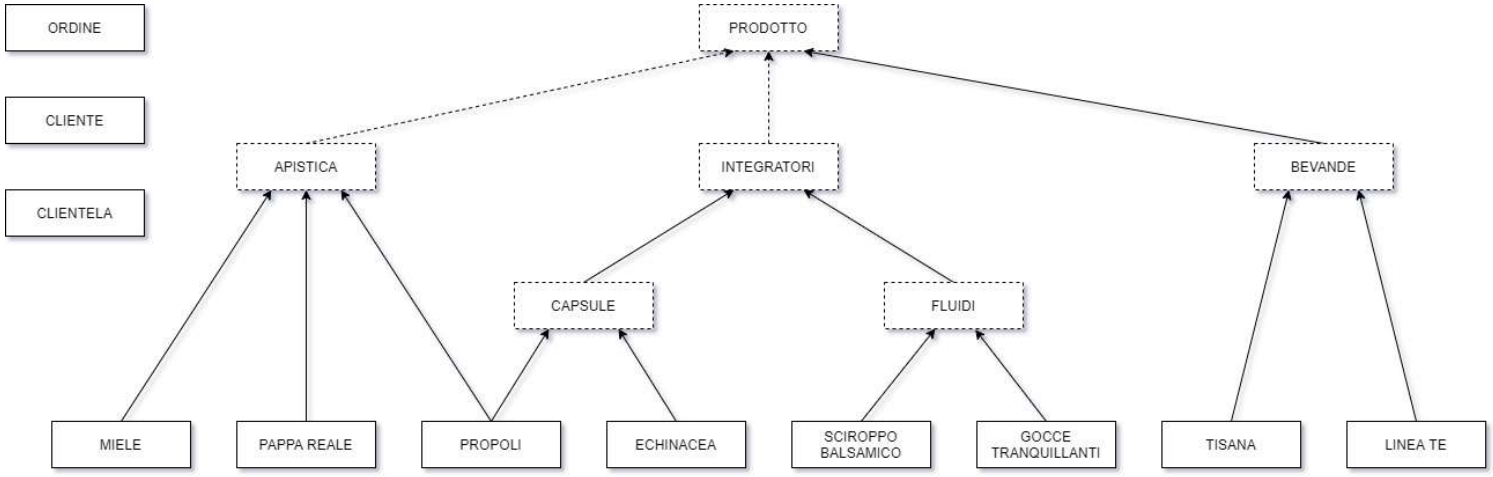
La stima delle ore impiegate da me si suddividono così:

- Analisi problema 3 ore
- Progettazione 3 ore
- Apprendimento QT 9 ore
- Programmazione 35 ore
- Debug 4 ore
- Testing 2 ore

Il totale si avvicina alle 60 ore scarse, superando di non molto la soglia delle 50 ore prestabilite. Questo è dovuto al fatto che ho riscontrato problemi con la programmazione per lo sviluppo della parte grafica, poiché facevo molti errori e ho dovuto apprendere dedicare molto tempo all'apprendimento dei metodi e delle proprietà corrette dei vari oggetti e delle librerie.

DESCRIZIONE DELLA GERARCHIA DELLE CLASSI

La gerarchia che ora viene indicata, rappresenta tutti i vari prodotti venduti dall'erboristeria. Come si può notare questa gerarchia presenta un'ereditarietà multipla (a diamante). La gerarchia permette di istanziare prodotti veri e propri, ognuno di essi con le sue caratteristiche. Se diventasse necessario, potrebbe essere estesa con nuovi prodotti, derivando dalle classi virtuali proposte in questo modello.



PRODOTTO

È la classe base di tutta la gerarchia dalla quale derivano tutte le varie tipologie di prodotti acquistabili da un cliente. Ha la caratteristica di essere astratta e polimorfa. Al suo interno comprende gli attributi double prezzo e double peso. Come metodi, presenta il costruttore con ad un parametro (il peso), un metodo void setPrezzo(double) e due metodi definiti *const*, ovvero double getPeso() const per il ritorno del peso e double getPrezzo() const per il ritorno del prezzo. Presenta poi due metodi virtuali puri (double calcolaPrezzo() e std::string stampa()) e il distruttore virtuale, entrambi motivo per il quale la si può definire una classe astratta.

APISTICA

È una classe astratta, derivata virtualmente della principale classe base Prodotto. Il suo costruttore è a due parametri, precisamente il peso e il tempo di lavorazione. Infatti, al suo interno ha solo un campo dati, double tempoLavorazione, che viene definito in base al prodotto scelto tra le classi sottostanti, tramite il metodo virtuale puro virtual double calcolaLavorazione()=0 che calcola quanto è costato la lavorazione dei prodotti. Presenta poi due metodi definiti *const*, ovvero double getTempoLavorazione() const per il ritorno del tempo di lavorazione e double getTotaleLavorazione() const per il ritorno del costo della lavorazione.

MIELE

Classe che rappresenta il prodotto "*Miele*". La sua derivazione da Apistica non è virtuale. Al suo interno vengono dichiarati e poi inizializzati (nel file .cpp) due attributi statici, static double pMiele e static double PrezzoLavorazione. Il costruttore ha solo il peso come parametro, e nel file .cpp viene chiamato il costruttore di Apistica con, al suo interno, il parametro attuale che definisce l'effettivo tempo di lavorazione.

Dopo aver incluso <iostream> e <sstream>, viene poi definito all'interno della dichiarazione della classe un metodo friend per poter effettuare la stampa, ovvero friend std::ostringstream& operator<<(std::ostringstream&,const Miele&) che lavora insieme al metodo virtuale virtual std::string stampa() const. Vengono poi dichiarati altri due metodi virtuali, virtual double calcolaPrezzo() e virtual double calcolaLavorazione(), il primo che calcola il prezzo totale del prodotto sommandoci il prezzo di lavorazione ottenuto con il secondo metodo, il quale richiama il metodo getTotaleLavorazione() dalla classe Apistica.

PAPPA REALE
Classe che rappresenta il prodotto "*Pappa Reale*". È modellata come la classe Miele, con la derivazione da Apistica non virtuale e con la dichiarazione e inizializzazione di due attributi statici, static double pPappaReale e static double PrezzoLav.
Tutti i metodi presenti sono anch'essi analoghi a quelli della classe Miele.

PROPOLI
Classe che rappresenta il prodotto "*Propoli*". Chiude il diamante con Prodotto-Apistica-Integratori-Capsule. Per questo motivo è stato necessario derivare Apistica e Integratori come virtual da Prodotto. Al suo interno vengono dichiarati e poi inizializzati un attributo booleano bool naturale e tre attributi statici, static double pPropoli, static double prezzoLavoroP e static double pNaturale. Il costruttore presenta ben tre parametri, il peso e due bool, uno per indicare se il prodotto scelto è naturale (true se è naturale, false altrimenti, di default false), e il successivo per indicare se il prodotto scelto è venuto in sacchetto (true se è venduto in sacchetto, false altrimenti, di default false), caratteristica ereditata da Capsule.
Tutti gli altri metodi presenti all'interno sono gli stessi analoghi a quelli della classe Pappa Reale e Miele.

INTEGRATORI
È una classe astratta, derivata virtualmente della principale classe base Prodotto. Il suo costruttore presenta un solo parametro, precisamente il peso.

CAPSULE
È una classe astratta, derivata (ma non virtualmente) dalla classe Integratori. Il suo costruttore è a due parametri, precisamente il peso e il sacchetto, il quale è definito da uno dei suoi due campi dati bool sacchetto (true se è venduto in sacchetto, false altrimenti, di default false). L'altro attributo è static double scontoSacchetto. Sono poi presenti anche due metodi *const,* bool getSacchetto() const per far ritornare il valore booleano di sacchetto e double getScontoSacchetto() const che ritorna il valore di scontoSacchetto.

ECHINACEA
Classe che rappresenta il prodotto "*Echinacea*". È modellata come le altre classi che stanno nell'ultimo livello della gerarchia (tranne Propoli), con la derivazione dalla classe superiore, in questo caso Capsule, non virtuale, con la dichiarazione e inizializzazione di attributi statici, in questo caso solo uno, static double PEchinacea, con il costruttore a due parametri che in questo caso sono il peso e il sacchetto, e infine con la presenza di un metodo virtuale per il calcolo del prezzo e dei metodi per la stampa.

FLUIDI
È una classe astratta, derivata (ma non virtualmente) dalla classe Integratori. Il suo costruttore è a due parametri, precisamente il peso e un bool che specifica il tipo di confezione, che è o plastica o vetro. È definito dall'attributo bool confezionePlastica (true se è in plastica, false se è in vetro, di default true). Ci sono poi due attributi statici, static double prezzoPlastica e static double prezzoVetro, per definire il prezzo dei due tipi diversi di confezione. Presenta poi tre metodi di tipo *const,* bool getConfF() const per far ritornare il valore booleano della confezione, quindi di confezionePlastica, double getPrezzoPlastica() const e double getPrezzoVetro() const che ritornano i due valori richiesti.

SCIROPPO BALSAMICO
Classe che rappresenta il prodotto "*Sciroppo balsamico*". È modellata come le altre classi che stanno nell'ultimo livello della gerarchia (tranne Propoli), con la derivazione dalla classe superiore, in questo caso Fluidi, non virtuale, con la dichiarazione e inizializzazione dell'attributo statico static double prezzoScirop, con il costruttore a due parametri che in questo caso sono il peso e la confezione, e infine con la presenza di un metodo virtuale per il calcolo del prezzo e dei metodi per la stampa.

GOCCE TRANQUILLANTI

Classe che rappresenta il prodotto "*Gocce tranquillanti*". È modellata come le altre classi che stanno nell'ultimo livello della gerarchia (tranne Propoli), con la derivazione dalla classe superiore, in questo caso Fluidi, non virtuale, con la dichiarazione e inizializzazione dell'attributo statico static double prezzogocce, con il costruttore a due parametri che in questo caso sono il peso e la confezione, e infine con la presenza di un metodo virtuale per il calcolo del prezzo e dei metodi per la stampa.

BEVANDE

È una classe astratta, derivata (ma non virtualmente) della principale classe base Prodotto. Il suo costruttore è a due parametri, precisamente il peso e il tipo di confezione, che può essere confezione normale o confezione regalo. Presenta due campi dati, bool tipoConfezione e static double prezzoConfReg.
Presenta poi due metodi definiti *const*, ovvero bool getTipoConf() const per il ritorno del tipo di confezione e double getPrezzoConReg() const per il ritorno del suo prezzo.

TISANA

Classe che rappresenta il prodotto "*Tisana*". È modellata come le altre classi che stanno nell'ultimo livello della gerarchia (tranne Propoli), con la derivazione dalla classe superiore, in questo caso Bevande, non virtuale. Presenta poi la dichiarazione e inizializzazione dei suoi vari campi dati: bool filtro, static double prezzofiltro e static double pTisana. Il costruttore è a tre parametri, con peso, confezione (true se è regalo, false se è in normale, di default false) e filtro (true se c'è, false se non c'è, di default false). Infine, presenta un metodo virtuale per il calcolo del prezzo e i metodi per la stampa.

LINEA TE

Classe che rappresenta il prodotto "*Linea te*", ovvero una confezione di *te*. È modellata come le altre classi che stanno nell'ultimo livello della gerarchia (tranne Propoli), con la derivazione dalla classe superiore, in questo caso Bevande, non virtuale.
Presenta poi la dichiarazione e inizializzazione dei suoi vari campi dati: bool aromatizzato, static double prezzoaromatizzato e static double prezzoTe. Il costruttore è a tre parametri, con peso, confezione (true se è regalo, false se è in normale, di default false) e aromatizzato (true se aromatizzato, false se non lo è, di default false). Infine, presenta un metodo virtuale per il calcolo del prezzo e i metodi per la stampa.

La classe **List** è il contenitore templatizzato. È modellata sulla base di quella mostrata a lezione, e viene usata per contenere i prodotti List<Prodotto*> prodotti, gli ordini List<Ordine*> ordini e i clienti List<Cliente*> clienti. Sono presenti poi tre classi: **Ordine**, **Cliente** e **Clientela**. Infatti, il modello si sviluppa mettendo al centro il cliente: con una lista di prodotti si crea un ordine, il quale viene assegnato al cliente, che è un elemento di una lista clienti.

La classe **Clientela** rappresenta la lista dei clienti registrati nell'applicazione, infatti contiene una lista di Clienti* e metodi per la loro gestione, come la creazione di un nuovo cliente, la creazione di un nuovo ordine e l'aggiornamento di un cliente.

La classe **Cliente** rappresenta il cliente che va a effettuare un ordine e contiene per l'appunto una lista di Ordine* (per memorizzare tutti gli ordini del cliente) e i metodi per la gestione degli attributi del cliente stesso, come ad esempio per i punti. Per quanto riguarda la scelta del campo std::string nome, si è scelto di crearne uno unico per nome e cognome, per lasciare una possibilità all'utente dell'applicazione di inserire il cliente con il formato "*nome cognome*" oppure con un *nome utente* (quindi in un'unica stringa). Sulla base di questa gerarchia non sono implementati meccanismi di controllo per l'inserimento di un nome univoco per ogni cliente. Questo perché si è pensato di creare un'applicazione per un'erboristeria che non conta un numero elevato di clienti, e che quindi, solo in previsione di un notevole aumento della clientela, si potrà pensare all'implementazione di un sistema di controllo.

La classe **Ordine** rappresenta il singolo ordine effettuato da un cliente e contiene al suo interno una lista di Prodotti* e di metodi per la gestione dell'ordine, quindi il prezzo totale, l'IVA, l'inserimento di un prodotto e i metodi per la stampa.

POLIMORFISMO

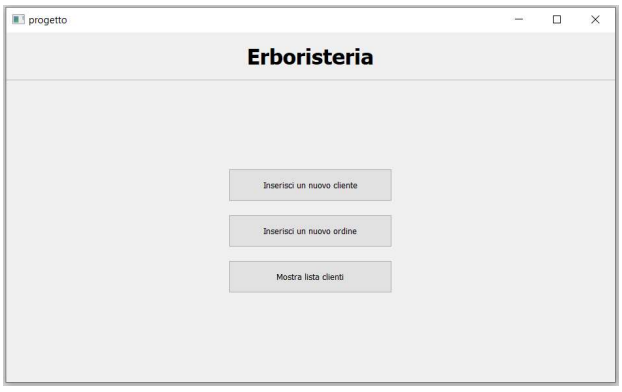
All'interno del progetto, il polimorfismo viene sfruttato per definire i vari prodotti, grazie alla presenza delle classi astratte e quindi per la virtualizzazione dei metodi. I metodi virtuali sono:

1. std::string stampa(), definito puro in Prodotto, che, tramite l'overloading dell'operatore <<, stampa nello spazio apposito il prodotto selezionato, mostrandone prezzo e quantità (nella classe InserisciOrdine stampa nel QPlainTextEdit fattura).
2. double calcolaPrezzo(), definito puro in Prodotto, che viene richiamato quando si deve calcolare il prezzo di ogni singolo prodotto. Per esempio, all'interno del metodo double calcolaTotale() nella classe Ordine, è richiamato per fare la somma del prezzo di ogni prodotto salvato nell'ordine.
3. calcolaLavorazione(), definito puro in Apistica, che sulla base del parametro attuale tempoLavorazione delle classi figlie, calcola il prezzo definitivo del prodotto selezionato.

DESCRIZIONE GUI

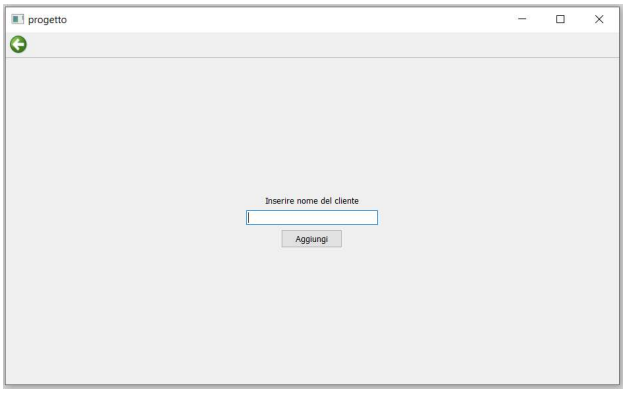
Per la gestione delle quattro schermate sulla quale si basa l'applicazione, abbiamo creato una classe handler: essa infatti consente di visualizzare una schermata e di spostarsi da una schermata all'altra, tramite segnali e slot generati da i bottoni appositi. Inoltre, permette alle classi InserisciOrdine, InserisciCliente e MostraClienti interagire tra di loro sempre con segnali e slot e quindi permette, quando è necessario, l'aggiornamento dei dati presenti in una schermata diversa da quella visualizzata in quel preciso momento.

Nello spazio sottostante, vengono riportate le quattro schermate dell'applicazione con alcuni dettagli tecnici per un uso più pensato.



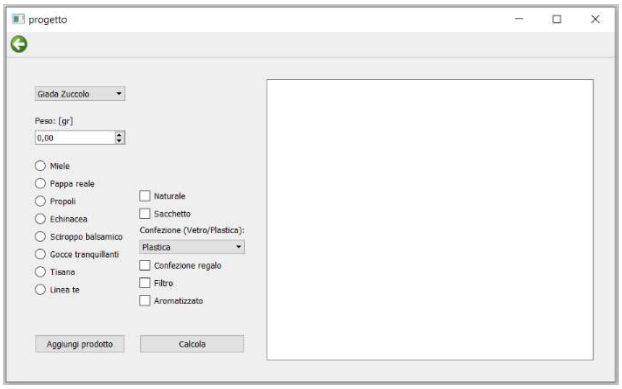
SCHERMATA "HOME"

Schermata principale dell'applicazione. Da qui è possibile scegliere cosa fare, in base ai bottoni disposti verticalmente in centro alla schermata:
- inserire un nuovo cliente
- inserire un nuovo ordine ad un cliente
- mostrare la lista dei clienti inseriti



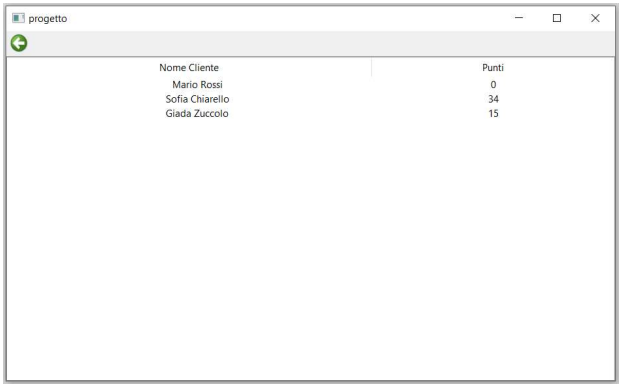
SCHERMATA "INSERISCI CLIENTE"

Schermata nella quale è possibile inserire il nome del cliente da aggiungere. Da modello è presente una sola LineEdit dove si può inserire o "*nome cognome*" o un "*nome utente*".
La freccia nella toolbar permette di tornare alla schermata *home* nel caso si è entrati qui per errore. Altrimenti, dopo aver cliccato sul pulsante *Aggiungi*, appare una message box di conferma e si ritorna automaticamente alla schermata *home*



SCHERMATA **"INSERISCI ORDINE"**

Schermata di inserimento di un ordine. Sono stati inseriti controlli gestionali per gli errori. In particolare, non si può aggiungere un prodotto se non si ha selezionato il peso e/o il prodotto specifico, non si può cliccare sul pulsante "*Calcola*" se non si è prima aggiunto un prodotto, non si può cambiare cliente dopo aver inserito già un prodotto con un determinato cliente. La freccia nella toolbar ha lo scopo di far tornare nella schermata home e di confermare (o no) il successo dell'inserimento dell'ordine. Le CheckBox poste a fianco ai RadioButton sono progettate per essere considerate solamente se il prodotto selezionato le chiama in causa. In caso contrario, non verranno mostrati messaggi di errore nella schermata, ma non verrà considerato nessun cambiamento di prezzo



SCHERMATA **"MOSTRA CLIENTI"**

Semplice schermata che elenca in una tabella i clienti presenti e mostra i rispettivi punti.
La freccia nella toolbar riporta semplicemente alla schermata *home*.