Andrea Longo

1174957

QBAR

PROGRAMMAZIONE AD OGGETTI

NOTE DI COMPILAZIONE

Poiché nel codice relativo al progetto sono state utilizzate keyword introdotte da c++11, per la corretta compilazione è necessario utilizzare un diverso file progetto, QBar.pro, che è disponibile nella cartella della consegna.
Per la compilazione si eseguono i comandi:

```
qmake -o Makefile QBar.pro

make

./QBar
```

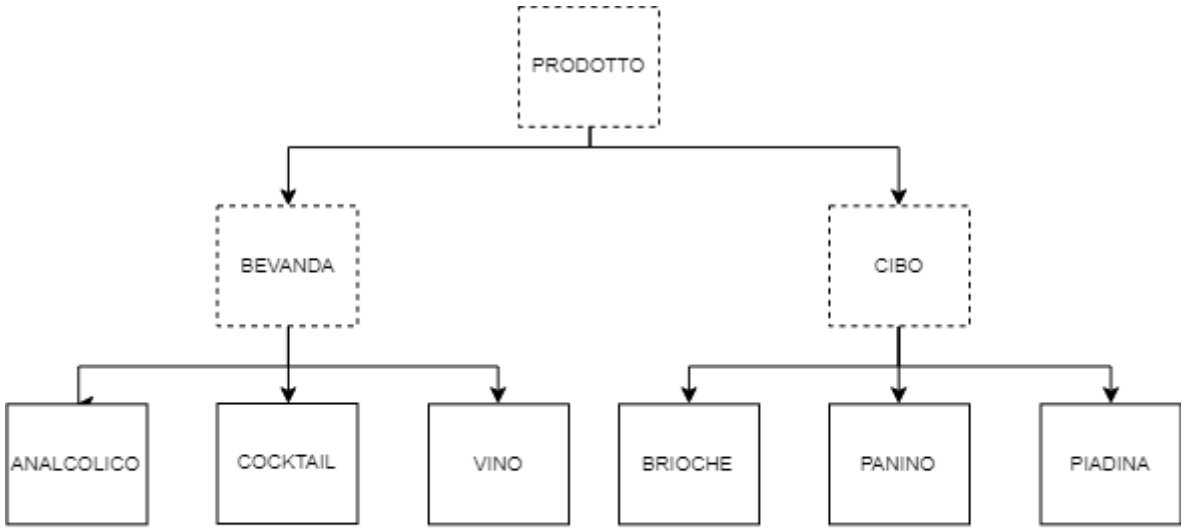
DESCRIZIONE

Lo scopo del progetto è quello di modellare un software utile alla gestione del listino prodotti di un bar, quindi con funzionalità di inserimento e rimozione, possibilità di visionare il listino dei prodotti, e di effettuare ricerche secondo diversi parametri.

Lo scopo non è quindi quello di fornire un servizio per il monitoraggio delle disponibilità dell'inventario, ma solo quello di offrire una lista di prodotti acquistabili.

GERARCHIA

La gerarchia si sviluppa su tre livelli:



La classe base astratta:

- **Prodotto**: è una classe base astratta, che porta informazioni generiche come il nome del prodotto, l'età minima e la scadenza.

Abbiamo un ulteriore livello di derivazione con due classi astratte:

- **Bevanda**: è una classe astratta derivata da Prodotto, le informazioni aggiunte sono i carboidrati, le proteine e i grassi, oltre alla possibilità di essere o meno alcolica.
- **Cibo**: è una classe astratta derivata da Prodotto, le informazioni aggiunte sono i carboidrati, le proteine e i grassi, oltre alla possibilità di essere o meno vegano.

(Il motivo per cui sia Cibo che Bevanda possiedono attributi riguardo carboidrati, proteine e grassi, anziché essere descritti direttamente su Prodotto, è che un eventuale estensione della gerarchia potrebbe aggiungere articoli che non richiedono questi dettagli, come ad esempio tabacchi, etc..).

Le classi astratte di secondo livello, vengono poi derivate da diverse classi concrete:

- **Analcolico**: è una classe concreta derivata da Bevanda, che oltre ai parametri ereditati aggiunge informazioni riguardo il prezzo netto, il codice a barre, la variante, il contenitore e la quantità, infine contiene un campo statico che rappresenta la tassa da applicare per ottenere il prezzo di vendita.
- **Cocktail**: è una classe concreta derivata da Bevanda, contiene parametri aggiuntivi rigurado il prezzo netto, il codice a barre, la taglia del cocktail, la famiglia e la gradazione alcolica, oltre ad un campo statico che rappresenta la tassa.
- **Vino**: è una classe concreta derivata da Bevanda, porta informazioni riguardo il prezzo netto, il codice a barre, la regione, l'anno di produzione, la quantità e la gradazione alcolica, anche questa classe ha un suo campo statico contenente la tassa.
- **Brioche**: è una classe concreta derivata da Cibo, i parametri aggiuntivi sono il prezzo di preparazione, il codice a barre, l'impasto utilizzato, la forma e il ripieno, anch'essa contiene un campo dati statico con il valore della tassa.
- **Panino**: è una classe concreta derivata da Cibo, che oltre al prezzo di preparazione e al codice a barre aggiunge il tipo di pane utilizzato, come nelle precedenti classi, anche Panino contiene un campo dati con la tassa.
- **Piadina**: è una classe concreta derivata da Cibo, che fornisce parametri supplementari come il prezzo di preparazione, il codice a barre e il tipo di impasto utilizzato, oltre al campo dati statico relativo alla tassa.

A livello di codice non viene sfruttata direttamente la gerarchia, ma uno SmartPtr<Prodotto*>.

E' infatti stato definito un template SmartPtr<T> che ha come unico scopo la gestione della memoria profonda, quindi non introduce ulteriori parametri alla gerarchia.

POLIMORFISMO

I metodi virtuali puri introdotti dalla classe Prodotto sono:

- **virtual ~Prodotto()**, il distruttore virtuale utile per invocare sempre il distruttore corretto.
- **virtual Prodotto* create(Json::Value&) const**, un metodo che ritorna un puntatore polimorfo a Prodotto. Il sottotipo concreto che lo implementa riceve come parametro un Json::Value contenente le informazioni riguardo l'oggetto serializzato, quello che deve fare è ritornare un puntatore ad un oggetto creato con le informazioni prelevate da Json::Value.
- **virtual Prodotto* clone() const**, che ha lo scopo di ritornare un puntatore polimorfo a Prodotto. L'implementazione da parte di una classe derivata concreta, richiede di ritornare un puntatore ad una copia dell'oggetto di invocazione.
- **virtual std::string getBarCode() const**, questo metodo virtuale puro viene implementato dalle sottoclassi concrete e deve ritornare una stringa con il bar code del prodotto rappresentato dall'oggetto di invocazione.
- **virtual double getPrezzo() const**, questo metodo polimorfo, richiede alla sottoclasse concreta di ritornare un double che rappresenta il prezzo lordo del prodotto.
- **virtual std::string getTipo() const**, questo metodo virtuale puro ritorna una stringa che rappresenta il tipo del prodotto.
- **virtual void serialize(Json::Value&) const**, questo metodo virtuale puro viene implementato dalle classi derivate concrete, viene utilizzato per la serializzazione. La sottoclasse che lo implementa riceve un Json::Value e deve popolarlo con i propri dati.

Nella classe derivata Bevanda viene introdotto un altro metodo virtuale puro:

- **virtual double getPrezzoNetto() const**, richiede alla sottoclasse concreta che lo implementa, di ritornare il prezzo netto della bevanda.

Nella classe derivata Cibo viene introdotto un altro metodo virtuale puro:

- **virtual double getPrezzoPreparazione() const**, richiede alla sottoclasse concreta che lo implementa, di ritornare il prezzo della preparazione del cibo.

Inoltre sia Bevanda che Cibo, possiedono un metodo virtuale non puro:

- **virtual double calcolaCalorie() const**, che calcola le calorie della Bevanda/Cibo, il metodo è virtuale poiché può eventualmente essere sovrascritto/arricchito dalle sottoclassi, per un calcolo più dettagliato delle calorie.

SERIALIZZAZIONE

Per consolidare la consistenza dell'applicazione vengono rese disponibili le funzionalità di lettura/scrittura da/su file, che consentono quindi di memorizzare gli oggetti del listino e di ricaricarli in un secondo momento.

Il formato di file adottato è JSON, poiché è estremamente "human redable", e risulta essere molto meno prolisso di altri formati. Nonostante sia relativamente giovane, è attualmente molto diffuso, e fa della semplicità la sua principale caratteristica, che in progetti non troppo complessi, come in questo caso, è un notevole punto di forza.

E' stata utilizzata una libreria esterna **JSONCPP** per operare su file json. La libreria è estremamente semplice, ed è stata ritenuta più che sufficiente per gli scopi di questo progetto. I file sono già presenti dentro la cartella della consegna. Per maggiori dettagli riguardo il funzionamento, il codice, insieme ad una semplice documentazione, sono reperibili qui:

https://github.com/open-source-parsers/jsoncpp

Per gestire la lettura/scrittura da/su file degli oggetti, è stato adottato il "second level of sophistication" descritto da isocpp.org.

https://isocpp.org/wiki/faq/serialization#serialize-inherit-no-ptrs

Struttura Gerarchia

La classe base Prodotto contine una static std::map<std::string, Prodotto*>, posizionata nella sezione protetta (protected) della classe, cosi da renderla accessibili alle sottoclassi.

La std::string contiene il tipo del Prodotto sotto forma di stringa, mentre Prodotto* è un puntatore polimorfo, lo scopo della std::map è quindi quello di mappare ogni stringa con la classe della gerarchia che rappresenta.

La std::map viene popolata durante un'inizializzazione statica dalle varie sottoclassi concrete. Per farlo in ogni sottoclasse concreta è stato descritto un oggetto che viene inizializzato staticamente e popola la std::map con una coppia chiave/valore, dove la chiave è il nome del tipo sotto forma di stringa, mentre il valore è un puntatore di quel tipo.

Metodi Gerarchia

Dopo aver trattato la struttura della gerarchia, analizziamo i metodi utili alla serializzazione/deserializzazione.

La classe base astratta richiede alle sottoclassi concrete l'implementazione di un metodo **serialize**, che come descritto in precedenza, ha lo scopo di smembrare l'oggetto e memorizzarlo su un Json::Value.

L'operazione inversa viene svolta dal metodo **Prodotto* unserialize(Json::Value&)**, il quale ricevuto un Json::Value ha lo scopo di invocare il metodo **create** della sottoclasse corretta, e ritorna infine un puntatore polimorfo a Prodotto*. Il metodo create, già descritto, ha lo scopo di ricostruire l'oggetto a partire da un Json::Value e ritornare un puntatore polimorfo a Prodotto*.

Utilizzo

E' stata sviluppata una classe JsonIO, per rendere più facile le operazioni di lettura/scrittura da/su file.

Il modello fa utilizzo dei metodi di serilaizzazione della gerarchia in due suoi metodi:

- **void save(const std::string&) const**, questo metodo riceve una std::string che rappresenta il file su cui salvare il contenuto del modello, viene poi passato il controllo a JsonIO, che grazie al metodo **void write(const Qontainer<SmartPtr<Prodotto>>&) const,** si occupa della scrittura su file dei prodotti contenuti nel modello, per farlo si serve del metodo **serialize**.
- **void load(const std::string&)**, riceve come parametro una std::string che rappresenta il file da leggere per popolare il contenuto del modello, anche in questo caso il controllo viene passato alla classe JsonIO, che tramite l'invocazione del metodo **void write(const Qontainer<SmartPtr<Prodotto>>& prodotti) const,** popola il modello con il contenuto del file, per farlo utilizza il metodo **unserialize**.

GUI

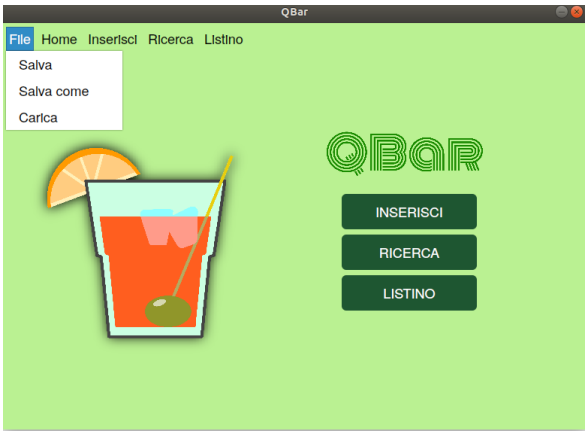
Relativamente alla parte grafica dell'applicazione questa è composta da:

- **Controller**: è il cuore dell'applicativo, si occupa di mostrare i layout corretti in base alla scelta dell'utente, e gestisce le richieste di interazione con il modello da parte della vista.
- **Index Layout**: è la home, ha il semplice scopo di accogliere l'utente, non ha interazioni con il modello.
- **Search Layout**: è il layout di ricerca, è possibile cercare i prodotti secondo diversi parametri, inoltre tramite la selezione di un prodotto è possibile rimuovero o modificarlo.
- **Insert Layout**: è il layout di inserimento, in base al tipo di prodotto scelto genera un form diverso.
- **Listino Layout**: è il layout che mostra il contenuto del listino, oltre a visionare l'intero conenuto è possibile, tramite selezione, la rimozione o la modifica di un prodotto.

Inoltre è stato aggiunto un foglio di stile style.qss, per rendere più gradevoli i colori, le regole non apportano alcun cambiamento drastico ai layout e al posizionamento dei widget in essi. Quindi anche rimuovendo il foglio di stile, l'utilizzo del software rimane invariato.

MANUALE DI UTILIZZO

Index Layout



La home page è molto semplice, presenta il logo e il nome del bar.

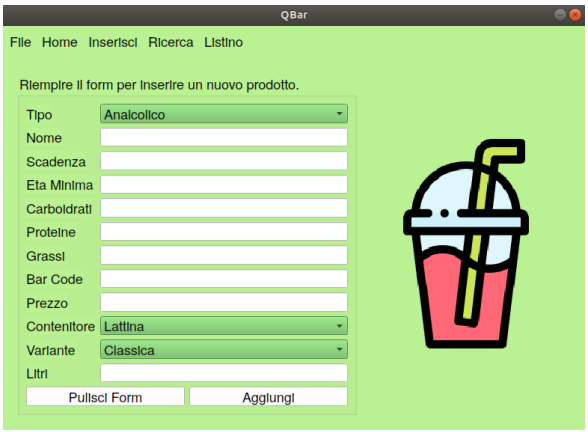
Sotto si ha una barra di navigazione, che si può utilizzare per accedere alle diverse schermate, come alternativa alla barra del menu.

La barra del menu, che è presente in tutte le schermate, contine un menu a tendina "File" da cui è possibile salvare il file corrente, salvare su un altro documento, o caricare un file.

Le restanti voci del menu fungono da barra di navigazione.

Insert Layout

La finestra di inserimento dei prodotti è piuttosto intuitiva.



verrà Form".

Dopo aver scelto il tipo di prodotto che si desidera inserire, viene mostrato il form corretto, il quale va completato per intero e con valori non nulli, altrimenti mostrato un messaggio di errore.

Sul fondo troviamo i due pulsanti, uno di inseriemento, "Aggiungi" e un secondo per la pulizia del form, "Pulisci

Anche il pulsante "Aggiungi", in caso di inserimento dei dati, provvederà a pulire il form.

Search Layout

Il layout di ricerca contiene una semplice barra di ricerca, e un tabella contenente gli elementi trovati.

Per effetturare una ricerca è sufficienete scegliere il parametro su cui si vuole effettuare la ricerca, digitare il valore nella barra di ricerca e premere il pulsante “Ricerca”.

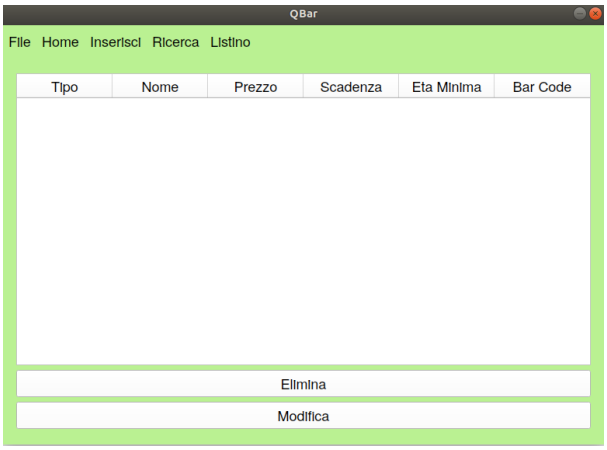
Nella tabella verranno mostrati gli eventuali risultati. E’ possibile selezionare un risultato e agendo sui pulsanti sotto rimuoverlo o modificarlo.

Listino Layout

Anche quest scheda è molto intuitiva.

E’ possibile accedervi in qualunque momento per visulizzare l’intero contenuto del listino, verrà infatti aggiornato automaticamente ad ogni inserimento/modifica.

E’ anche possibile selezioanare un elemento ed eliminarlo oppure effetuare una modifica.



TEMPISTICHE

Il progetto ha richiesto 58 ore circa.

Le ore in eccesso sono dovute principamente allo sviluppo delle parti grafiche (loghi, icone, foglio di stile, etc.), e alla sistemazioni di piccoli errori emersi nella fase di testing finale.

Una lista più dettagliata delle ore impiegate nello sviluppo.

Gerarchia

- Ricerca e Analisi della gerarchia da modellare: 2 ore
- Codifica del Qontainer: 4 ore
- Codifica della Gerarchia: 8 ore
- Codifica Modello e SmartPtr: 3 ore
- Debugging Vista: 2 ore

Vista

- Tutorato e Studio di Qt: 15 ore
- Analisi e Wireframing GUI: 2 ore
- Codifica GUI: 12 ore
- Debugging Vista: 2 ore

Stile

- Sviluppo loghi e icone: 3 ore
- Stesura regole CSS: 1 ora

Conclusione

- Testing finale: 2 ore
- Relazione finale: 2 ore