PROGETTO “DISTILLERIA VENETA”

RELAZIONE PROGETTO PAO

AA 2020/2021 | BRUGNOLARO FILIPPO | matricola 1217321

Introduzione

Distilleria Veneta è un’applicazione desktop gestionale che si occupa di prendere in carico gli ordini effettuati dai clienti dell’azienda. Le funzionalità sono quindi simili a quelle di un registratore di cassa, cercando però di offrire ai dipendenti un qualcosa di elegante oltre che funzionale.

I dati gestiti sono principalmente formati dalle generalità dei prodotti disponibili nell’azienda che possono essere o meno inseriti nello scontrino finale.

Vincoli Tecnici

- Sistema Operativo Target : Ubuntu 18.04
- Sistema Operativo di sviluppo : Ubuntu 20.04 Focal Fossa
- Versione Qt : 5.9.5
- Versione g++ : 7.4.0

Suddivisione del lavoro

Il progetto è stato sviluppato con Nicholas Pilotto (matricola 1230237). Insieme abbiamo analizzato il dominio del problema, progettato il modello, la gerarchia, il controller e la GUI. Inizialmente avevamo pensato di suddividere tutto nel modo più semplice, ovvero back-end e front-end. Alla fine però abbiamo deciso, per dimostrare le capacità acquisite da entrambi i lati di programmazione, di suddividerci la parte di back-end e di front-end in maniera più o meno equa, in base anche alla difficoltà nel reperimento delle informazioni dei vari compiti. Le modalità di coordinamento sono avvenute tramite incontri Zoom e sincronizzazioni tramite Github.

<table>
<tr><th colspan="2">Brugnolaro Filippo</th><th colspan="2">Nicholas Pilotto</th></tr>
<tr><td colspan="2">deep_ptr.hpp</td><td colspan="2">u_vector.hpp</td></tr>
<tr><td colspan="2">enum.h</td><td colspan="2">style.qss</td></tr>
<tr><td>product.h**</td><td>product.cpp**</td><td colspan="2" rowspan="8">Controllo errori della gerarchia
e
testing generale</td></tr>
<tr><td>non_spirits.h</td><td>non_spirits.cpp</td></tr>
<tr><td>spirits.h</td><td>spirits.cpp</td></tr>
<tr><td>cream.h</td><td>cream.cpp</td></tr>
<tr><td>liquor.h**</td><td>liquor.cpp**</td></tr>
<tr><td>grappa.h</td><td>grappa.cpp</td></tr>
<tr><td>young.h**</td><td>young.cpp**</td></tr>
<tr><td>old.h</td><td>old.cpp</td></tr>
<tr><td>view.h*</td><td>view.cpp*</td><td colspan="2">resources.qrc</td></tr>
<tr><td>gridshow.h*</td><td>gridshow.cpp*</td><td colspan="2">data.json</td></tr>
<tr><td>receiptshow.h*</td><td>receiptshow.cpp*</td><td>receipt.h</td><td>receipt.cpp</td></tr>
<tr><td>qstackedwidgethover.h</td><td>qstackedwidgethover.cpp</td><td>model.h</td><td>model.cpp</td></tr>
<tr><td>qproduct.h*</td><td>qproduct.cpp*</td><td>controller.h</td><td>controller.cpp</td></tr>
<tr><td>overlay..h</td><td>overlay.cpp</td><td>filter.h</td><td>filter.cpp</td></tr>
<tr><td>tablerow.h</td><td>tablerow.cpp</td><td>io_json.h</td><td>io_json.cpp</td></tr>
</table>

* Le classi contrassegnate contengono SIGNAL & SLOT sviluppati da Nicholas Pilotto

** Le classi contrassegnate contengono alcune metodi utili per il filtraggio e I/O sviluppati da Nicholas Pilotto

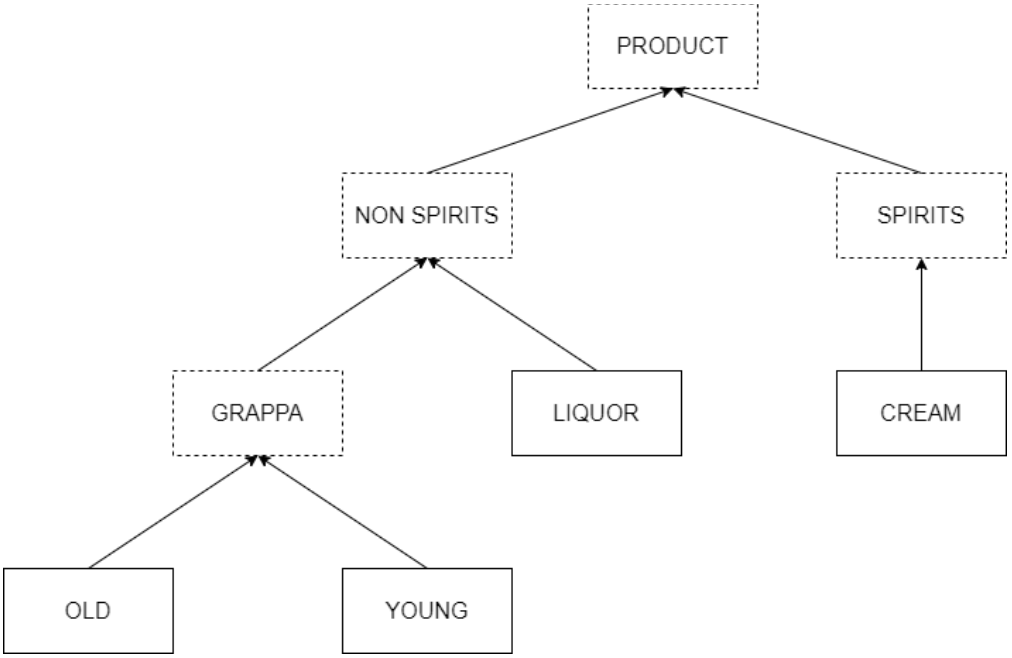
La stima riguardo le ore impiegate in questo progetto è la seguente:

- Analisi del problema : 2 ore
- Progettazione modello e GUI : 3 ore
- Apprendimento Qt : 16 ore
- Programmazione : 30 ore
- Debugging : 3 ore
- Testing : 1 ore

Il totale si attesta sulle 55 ore circa, superando non di molto la soglia di ore assegnata.
La giustificazione è data dal fatto che sono servite molte ore per apprendere Qt, libreria per me nuova, e mi è servito molto tempo per cercare di decidere quali classi particolari utilizzare.

Gerarchia principale

La gerarchia rappresenta tutte le varie tipologie di prodotti disponibili all'interno del dominio dell'azienda.
Gli oggetti che potranno essere istanziati saranno di tipo cream, liquor, old e young.
Quest'ultime 2 derivano da una classe astratta grappa che non istanzia tutti i metodi della classe base.
Analogamente grappa e liquor deriveranno da "non spirits" e cream da "spirits".
Tutte le classi descritte derivano dalla classe base astratta product.



PRODUCT

Classe astratta e polimorfa posta alla base della gerarchia che descrive in maniera generale tutte le tipologie di prodotti all'interno del dominio.

Per i campi dati si hanno bottle_size : kind, std::string : name, double : alcohol_content, che descrivono un prodotto allo stato grezzo.

La classe è dotata di un costruttore a tre parametri tutti con valori di default:
product(bottle_size = medium, const std::string& = "", double = min_ac)
Vengono dichiarate due variabili statiche min_ac e max_ac rappresentanti rispettivamente il valore minimo e massimo di gradazione degli alcolici disponibili.

Per i metodi se ne hanno alcuni non virtuali che elencherò velocemente :

- double : get_default_price() : protetto in quanto si vuole far conoscere alle sottoclassi il costo fisso di un prodotto, ma non si vuole che sia noto al pubblico.

- double : kind_price() const : variazione di prezzo in base alla dimensione
- double : price_increment() const : variazione di prezzo totale
- double : taxes() const : calcola le tasse in base al grado alcolico e alla dimensione
- void : set_kind_bottle(bottle_size = medium) : setter per cambiare la dimensione di un prodotto
- std::string : fromKindToStdString(bottle_size) const : converte l'enum bottle_size in stringa
- double : operator+(const product&) const : ritorna la somma dei prezzi dei prodotti (risp. -)
- bool : operator<(const product&) const : ritorna il valore di verità del confronto fra gradazioni (risp. >)

SPIRITS

Classe astratta in quanto non implementa tutti i metodi virtuali puri, la quale ha il principale compito di suddividere la classe PRODUCT in macrocategorie in quanto ci sono norme differenti da rispettare.
Vengono reimplementati i metodi kind_price() const e price_increment() const che serviranno a ridefinire il nuovo prezzo base dei superalcolici.
Si noti come vengano anche reimplementate le variabili statiche max_ac e min_ac e come in particolare assumano un ruolo importante anche nel costruttore.
La classe è dotata di un costruttore a tre parametri tutti con valori di default:
spirits(bottle_size = medium, const std::string& = "", double = min_ac)

NON SPIRITS

Classe astratta in quanto non implementa tutti i metodi virtuali puri, la quale ha il principale compito di suddividere la classe PRODUCT in macrocategorie in quanto ci sono norme differenti da rispettare. Come in spirits, vengono reimplementati i metodi kind_price() const e price_increment() const che serviranno a ridefinire il nuovo prezzo base degli alcolici.
Si noti come vengano anche reimplementate le variabili statiche max_ac e min_ac e come in particolare assumano un ruolo importante anche nel costruttore.
La classe è dotata di un costruttore a tre parametri tutti con valori di default:
non_spirits(bottle_size = medium, const std::string& = "", double = min_ac)

Nota : Si è voluto guardare oltre al progetto in sé, in quanto SPIRITS e NON SPIRITS potrebbero essere utilizzate in futuro per far derivare altre tipologie di prodotti che rispettino le caratteristiche e le normative della specifica categoria a cui si fa riferimento.

GRAPPA

Classe astratta in quanto non implementa tutti i metodi virtuali puri, la quale ha il principale compito di rappresentare l'insieme delle grappe in generale.
Vengono reimplementati i metodi kind_price() const e price_increment() const che serviranno a ridefinire il nuovo prezzo base delle grappe.
Si noti come vengano anche remiplementate le variabili statiche max_ac e min_ac e come in particolare assumano un ruolo importante anche nel costruttore.
La classe è dotata di un costruttore a tre parametri tutti con valori di default:
grappa(bottle_size = medium, const std::string& = "", double = min_ac)

LIQUOR

Classe concreta in quanto implementa tutti i metodi virtuali puri e che rappresenta la categoria dei liquori che hanno una gradazione alcolica X, con min_ac < X < max_ac.
La classe contiene una sottoclasse utile per l' I/O (di cui parleremo nella sezione dedicata all' I/O).
Per i campi dati, oltre a quelli ereditati dalle superclassi, si hanno color : col, u_vector<taste> : tastes, che vanno a descrivere meglio le caratteristiche specifiche del prodotto.
Vengono reimplementati i metodi kind_price() const e price_increment() const che serviranno a ridefinire il nuovo prezzo base dei liquori.
Si noti come vengano anche remiplementate le variabili statiche max_ac e min_ac e come in particolare assumano un ruolo importante anche nel costruttore.

La classe è dotata di un costruttore a cinque parametri tutti con valori di default :
liquor(const color = white, const u_vector<taste>& = {}, bottle_size = medium, const std::string& = "", double = min_ac)

CREAM

Classe concreta in quanto implementa tutti i metodi virtuali puri e che rappresenta la categoria delle creme che hanno una gradazione alcolica X, con min_ac < X < max_ac.
La classe contiene una sottoclasse utile per l' I/O (di cui parleremo nella sezione dedicata all' I/O).
Per i campi dati, oltre a quelli ereditati dalle superclassi, si hanno color : col, u_vector<taste> : tastes, che vanno a descrivere meglio le caratteristiche specifiche del prodotto.
Vengono reimplementati i metodi kind_price() const e price_increment() const che serviranno per ridefinire il nuovo prezzo base delle creme.
Si noti come vengano anche remiplementate le variabili statiche max_ac e min_ac e come in particolare assumano un ruolo importante anche nel costruttore.
La classe è dotata di un costruttore a cinque parametri tutti con valori di default :
cream(const color = white, const u_vector<taste>& = {}, bottle_size = medium, const std::string& = "", double = min_ac)

YOUNG

Classe concreta in quanto implementa tutti i metodi virtuali puri e che rappresenta la categoria delle grappe basiche che hanno una gradazione alcolica X, con min_ac < X < max_ac.
La classe contiene una sottoclasse utile per l' I/O (di cui parleremo nella sezione dedicata all' I/O).
Per i campi dati, oltre a quelli ereditati dalle superclassi, si hanno color : col, u_vector<taste> : tastes, che vanno a descrivere meglio le caratteristiche specifiche del prodotto.
Vengono reimplementati i metodi kind_price() const e price_increment() const che serviranno per ridefinire il nuovo prezzo base delle grappe basiche.
Si noti come NON vengano remiplementate le variabili statiche max_ac e min_ac e vengano solamente ereditate.
Si noti come ci sia un incremento di prezzo differente (nullo) rispetto alla classe GRAPPA in quanto viene considerata basica.
La classe è dotata di un costruttore a cinque parametri tutti con valori di default :
cream(const color = white_trasparent, const u_vector<taste>& = {}, bottle_size = medium, const std::string& = "", double = min_ac)

OLD

Classe concreta in quanto implementa tutti i metodi virtuali puri. Classe che rappresenta la categoria delle creme che hanno una gradazione alcolica X, con min_ac < X < max_ac.
La classe contiene una sottoclasse utile per l' I/O (di cui parleremo nella sezione dedicata all' I/O).
Per i campi dati, oltre a quelli ereditati dalle superclassi, si hanno color : col, u_vector<taste> : tastes, bool : barrique, unsigned int : month, che vanno a descrivere meglio le caratteristiche specifiche della grappa invecchiata specificando anche se è o meno barricata e quanti mesi di invecchiamento sono serviti.
Vengono reimplementati i metodi kind_price() const e price_increment() const che serviranno per ridefinire il nuovo prezzo base delle grappe invecchiate. (si noti come le variabili barrique e month siano importanti!)
Si noti come NON vengano remiplementate le variabili statiche max_ac e min_ac e vengano solamente ereditate.
La classe è dotata di un costruttore a sette parametri tutti con valori di default :
cream(const color = yellow_trasparent, const u_vector<taste>& = {}, bool = false, unsigned int = 18, bottle_size = medium, const std::string& = "", double = min_ac)

Tengo particolarmente a sottolineare come i costruttori siano complicati e siano stati usati valori di default per semplificare la scrittura dell' I/O.

Ogni classe contiene gli specifici metodi getter relativi ai propri campi dati

Metodi polimorfi

La gerarchia contiene i seguenti metodi virtuali:

- virtual product* clone() const = 0 : è il metodo di clonazione polimorfa e restituisce un puntatore alla copia dell'oggetto di invocazione.
  Si noti come esso sia stato implementato nelle classi concrete in quanto non ha senso clonare una base astratta dato che non sarebbe istanziabile.
- virtual std::string type_product() const = 0 : è un metodo che ad ogni sua implementazione deve ritornare una stringa che identifica la classe derivata; può essere utilizzato sia per la serializzazione, di cui parleremo successivamente, sia per avere informazioni sul tipo di oggetto polimorfo senza ricorrere al dynamic_cast (cha avviene attraverso RTTI), ma utilizzando il dynamic binding che ha un overhead minore in quanto si sfruttano le vtables.
- virtual u_vector<taste> get_tastes() const = 0 : è un metodo che ad ogni sua implementazione deve ritornare il vettore contenente i gusti del prodotto.
- virtual color get_color() const = 0 : è un metodo che ad ogni sua implementazione deve ritornare il colore.
- virtual std::string code() const = 0 : è un metodo che ad ogni sua implementazione aggiornerà il codice aggiungendo lettere o numeri identificativi per la creazione del codice.
- virtual double promotion() const = 0 : è un metodo che ritorna il prezzo totale di un prodotto scontato di una percentuale decisa da una variabile statica all'interno della classe instanziabile.
  Si noti come questa funzionalità sia stata implementata nelle classi concrete in quanto non ha senso scontare un prodotto non istanziabile.
- virtual std::string get_image_path() const : è un metodo che ad ogni sua implementazione aggiorna il percorso dove verranno posizionate le immagini. Si note come nelle classi istanziabili si sia immesso anche il nome dell'immagine oltre a quello della cartella.
- virtual ~product() = default : distruttore virtuale puro che permette l'invocazione del giusto distruttore da parte di qualsiasi oggetto polimorfo.
- virtual product* create(QMap<QString, QVariant>&) const = 0 : vedi SERIALIZZAZIONE
- virtual std::string write() const = 0 : vedi SERIALIZZAZIONE

Serializzazione

"Distilleria Veneta" consente il salvataggio di file che rappresentano contenitori. Per fare ciò viene utilizzato il file JSON, a discapito dell' XML.

Ci sono varie motivazioni che ci hanno spinto ad utilizzare il JSON rispetto all' XML:

- processamento più facile
- supporto di tantissimi linguaggi di programmazione
- rappresenta un miglior formato di scambio per i dati
- è possibile utilizzare array
- non usa i tag e rende i dati più leggibili

La classe PRODUCT contiene i seguenti metodi virtuali puri:

- virtual product* create(QMap<QString, QVariant>&) const = 0 :
  Ogni sottoclasse concreta D che implementa questo metodo deve leggere gli opportuni campi da una QMap, creare un nuovo oggetto della classe derivata e ritornare un puntatore a tale oggetto.
  Quindi verrà ritornato un D* che va bene per covarianza del tipo di ritorno.

- virtual std::string write() const = 0 :
  Ogni sua implementazione deve scrivere correttamente i propri campi dati e questo viene fatto attraverso l'inserimento con espressioni regolari. Chiaramente oltre ai campi dati verrà scritta anche l'informazione riguardante il tipo dell'oggetto, sfruttando il metodo virtuale puro std::string tipo().

Oltre a ciò, è necessario anche un metodo statico nella classe base PRODUCT che effettui la lettura del file chiamando il giusto metodo create. Per questo motivo allora l'informazione sul tipo dell'oggetto da creare va inserito all'inizio della scrittura del file. Per cui è stato aggiunto un campo dati protetto statico nella classe base PRODUCT:

inline static std::map<std::string, product*> _map

Questa mappa associa una stringa che contiene informazioni sul tipo D di un oggetto ad un puntatore polimorfo che dinamicamente punta a D. Possiamo dunque implementare il tutto con un metodo statico della classe base PRODUCT:

static product* unserialize(QMap<QString, QVariant>&) : è un metodo statico pubblico che legge il tipo dalla QMap passata per riferimento e controlla se c'è il tipo in questione ed in caso positivo chiama il metodo create(). Il late binding ci permette di selezionare il metodo create() corretto a run-time.

Per quanto riguarda l'inizializzazione della mappa, per ogni classe derivata concreta D è inserito un campo dati di tipo classe aux_map_initializer nella parte privata.

La classe aux_map_initializer, ha un campo dati statico di tipo D*. Questo perché grazie alla ridefinizione del distruttore ~aux_map_initializer() è possibile distruggerlo una volta terminato il programma.
Inoltre ha un costruttore aux_map_initializer() che costruisce un oggetto di tipo D (di default) tramite il campo dati e aggiunge il puntatore alla mappa.

Per quanto riguarda la scrittura, verranno creati dei json contenenti varie informazioni riguardanti il pagamento nel percorso ./file/Receipt/ che verrà creato dopo il primo pagamento andato a buon fine.

Istruzioni per la compilazione

Per compilare il programma nella macchina virtuale è necessario:

1. Estrarre la cartella zippata contente il progetto
2. Aprire il terminale in modo tale che il proprio percorso raggiunga la cartella unzippata
   In particolare è necessario essere dentro la cartella ./ProgettoPaO/DistilleriaVeneta
3. Eseguire il comando qmake
4. Eseguire il comando make
5. Eseguire il comando ./DistilleriaVeneta

Nella consegna sono inclusi dei file di esempio per l' I/O