Tommaso Azzalin Matricola: 1169740

QCafe

Progetto del corso “Programmazione ad oggetti” – A.A. 2018/2019

Sommario

Introduzione 1
Pattern progettuale 1
Gerarchie di tipi 2
Gerarchia di tipi (modello) 2
Descrizione della gerarchia 2
Gerarchia di tipi (vista) 3
Chiamate polimorfe 3
Formato dei file di caricamento/salvataggio dei dati del contenitore 4
Istruzioni di compilazione 4
Tempo richiesto per lo sviluppo 5

Introduzione

QCafe è un applicativo realizzato in **C++11** con la parte grafica realizzata con il framework **Qt**. L’applicativo permette la gestione di una coda (first-in-first-out) di ordini effettuati dai clienti di una caffetteria. Dal punto di vista dell’utente, esso è composta da due finestre: “**QCafe – Cashier**” e “**QCafe – Barista**”.

La prima, fornisce un’interfaccia per inserire all’interno della coda gli ordini, con la possibilità di personalizzarli. La seconda fornisce l’interfaccia che permette di visualizzare la coda per poter espletare gli ordini, rimuovendoli una volta completati, annullando ordini non più validi e permettendo piccole modifiche dell’ultimo minuto richieste dal cliente dell’ordine.

In entrambe le interfacce è presente la possibilità di salvare lo stato degli ordini (ovvero gli ordini ancora da espletare) in un file in formato XML, con la possibilità di recuperare lo stato attuale della coda degli ordini in un altro momento, aprendo i file .xml generati dall’applicativo.

Nota: il codice sorgente e l’interfaccia grafica sono principalmente in lingua inglese solo per pura comodità e per nessun altro motivo specifico.

Pattern progettuale

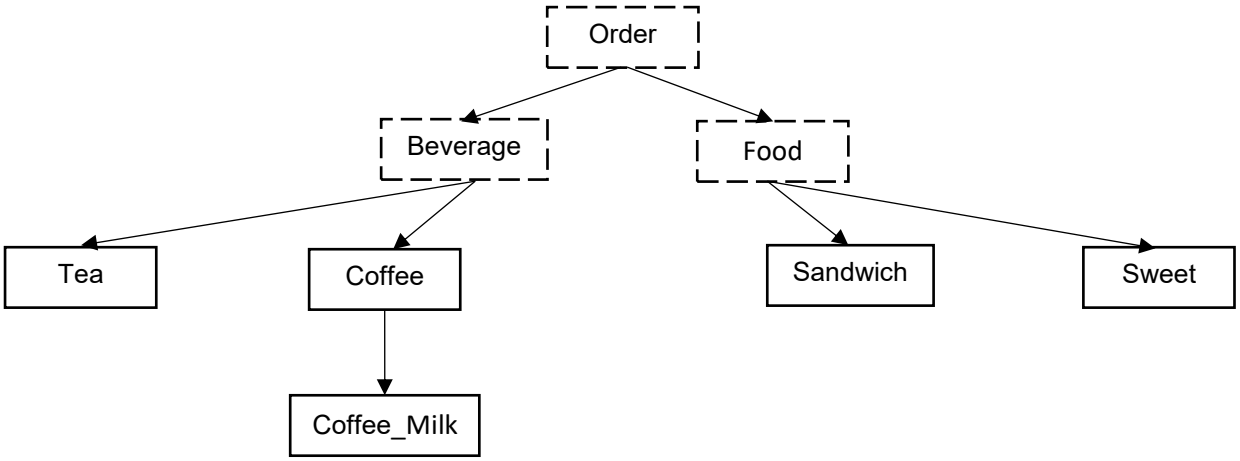
Per lo svolgimento del progetto è stato scelto di utilizzare il pattern di progettazione **MVC** (**Model-View-Controller**), implementato con le seguenti classi:

- **QCafe_Model**: la classe seguente offre un’interfaccia per l’utilizzo del contenitore (classe **Qontainer**) e la gerarchia di tipi illustrata in questo paragrafo. Qontainer implementa una struttura dati di tipo FIFO tramite una lista concatenata (singolarmente): in particolare, l’inserimento (in coda) e la rimozione (in testa) sono in tempo costante, la rimozione (in posizione casuale) è in tempo lineare, così come la ricerca degli elementi al suo interno;
- **QCafe_Controller**: la classe seguente permette di fare da tramite fra il modello e la vista, in modo da renderli indipendenti l’uno dall’altro permettendo, ad esempio, di poter modificare o cambiare completamente la vista senza la necessità di effettuare modifiche al modello;
- **QCafe_View**: la classe seguente permette di interagire con il modello (tramite la connessione al controller) attraverso un’interfaccia grafica. Questa classe non viene direttamente istanziata (è astratta), ma è utilizzata come base comune per le due classi **QCafe_Barista** e **QCafe_Cashier** che realizzano le due finestre descritte nel paragrafo di introduzione.

Gerarchie di tipi

Gerarchia di tipi (modello)

La gerarchia realizzata è composta di otto classi, con una base astratta comune, Order, da cui derivano altre due classi astratte, Beverage e Food, e da quest'ultime vengono derivate tutte le altre. Le classi sono definite all'interno di un namespace chiamato **QCafe**. È possibile schematizzare la gerarchia di tipi con lo schema seguente:



Descrizione della gerarchia

- **Order**: rappresenta un generico ordine di un prodotto effettuato nella caffetteria. Contiene informazioni sul nome del prodotto a cui si riferisce l'ordine, sul nome del cliente, sulla volontà di averlo d'asporto e sull'eventuale presenza di uno sconto sul prezzo totale dell'ordine. Il prezzo è calcolato sulla base di informazioni non presenti in questa classe.
- **Beverage** (derivato da **Order**): rappresenta un ordine di una bevanda. Contiene informazioni sulla quantità di prodotto in tre fasce (Small, Medium, Large) e sulla volontà di avere un dolcificante o meno all'interno.
- **Tea** (derivato da **Beverage**): rappresenta un ordine di un tè. Non contiene informazioni in più della classe da cui deriva ma dà in più solamente informazioni sul tipo di tè scelto (ovvero se è nero, verde o bianco) e il prezzo del prodotto. In un'ipotetica estensione della gerarchia si potrebbe derivare una classe Tea_Milk (per tè con latte) oppure una classe che rappresenta un ordine di un prodotto a base di tè.
- **Coffee** (derivato da **Beverage**): rappresenta un ordine di un caffè. Contiene informazioni sulla varietà di caffè desiderata dal cliente e sul prezzo del prodotto, calcolato in base alla varietà del caffè scelto e alla preparazione necessaria per realizzare il caffè da parte del barista (la scelta della macinatura e dello strumento, ad esempio).
- **Coffee_Milk** (derivato da **Coffee**): rappresenta un ordine di un caffè a cui è aggiunta la panna montata su richiesta del cliente, o di una bevanda al caffè cui nella sua realizzazione sia necessario l'utilizzo di latte (come un cappuccino), o entrambe. Contiene informazioni sugli aspetti appena citati e sul prezzo del prodotto.
- **Food** (derivato da **Order**): rappresenta un ordine di un cibo. Contiene informazioni sulla volontà del cliente di riscaldare il prodotto e sul fatto di essere vegano, vegetariano e/o senza glutine, anche se queste ultime dipendono dagli ingredienti del prodotto e quindi non sono disponibili in questa classe.
- **Sandwich** (derivato da **Food**): rappresenta un ordine di un panino. Contiene informazioni sull'ingrediente principale del panino, sul tipo di pane, sulla volontà di mettere formaggio, lattuga e una salsa, selezionando fra tre disponibili. Permette di conoscere sulla base degli ingredienti scelti sia il prezzo che se l'ordine è senza glutine o vegetariano o vegano.
- **Sweet** (derivato da **Food**): rappresenta un ordine di un dolce. Contiene informazioni sul gusto del dolce (cioccolato, mela e cannella oppure classico, nella classe indicato con *PLAIN*) e sulla volontà di avere della panna montata assieme all'ordine. Come per Sandwich, permette di conoscere sia il prezzo, sia se l'ordine è senza glutine o vegetariano o vegano.

Le classi **Order**, **Beverage** e **Food** sono astratte perché contengono campi dati comuni a tutti i loro rispettivi sottotipi, ma troppo generici per considerarli un tipo istanziabile. **Sandwich**, **Sweet**, **Tea**, **Coffee** e **Coffee_Milk** sono direttamente istanziabili.

Al fine di controllare i valori dei dati per la creazione degli oggetti si è fatto largo uso di campi dati enumerativi e booleani. Per i campi dati enumerativi sono stati inoltre creati dei metodi statici per la conversione da stringa a enum e viceversa. Ad esempio, per la classe Sweet si è voluto controllare il valore del gusto di un dolce: si è definito un tipo enumerativo **flavour** con valori possibili *PLAIN, CHOCOLATE* e *APPLE_AND_CINNAMON*. Si sono definiti poi due metodi statici: **string to_string(flavour)** e **flavour to_flavour(const string&)**.

Order è la base astratta della gerarchia e in quanto tale viene usata come tipo per istanziare la classe template Qontainer, indirettamente, in quanto viene usata la classe **DeepPtr** richiesta nel file delle specifiche del progetto, anch'essa template.

Infatti, invece che avere un'istanza del container del tipo:

```
Qontainer<Order*> q;
```

se ne ha una del tipo:

```
Qontainer<DeepPtr<Order>> q;
```

Gerarchia di tipi (vista)

Come già affermato nel paragrafo **Pattern progettuale**, per realizzare la vista si è creata una piccola gerarchia di tipi per poter implementare la vista dell'applicativo. Inoltre, sono stati implementati dei tipi derivati dai widget del framework Qt per un maggior controllo degli stessi e una comodità maggiore nel loro utilizzo. Di seguito sono illustrate le gerarchie di tipi implementate (sotto forma di elenco, non graficamente come per la gerarchia di tipi del modello):

- **QCafe_View** (derivato da **QWidget**): implementa un'interfaccia grafica comune ai due sottotipi (in particolare utilizza la classe MainMenu).
  - **QCafe_Barista**: implementa l'interfaccia grafica per la finestra "QCafe – Barista".
  - **QCafe_Cashier**: implementa l'interfaccia grafica per la finestra "QCafe – Cashier".
- **MainMenu** (derivato da **QMenuBar**): istanziato in QCafe_View, visualizza un menu *File* e un menu *About* per compiere alcune operazioni con i file di salvataggio del programma e per ottenere informazioni su quest'ultimo.
- **EditOrder** (derivato da **QWidget**): istanziato in QCafe_Cashier, visualizza un insieme di widget per personalizzare l'ordine base (selezionato nel widget QCafe_Cashier) assieme alle personalizzazioni fornite da EditOrderTabPanel.
- **EditOrderTabPanel** (derivato da **QTabWidget**): istanziato in QCafe_Cashier, implementa un'interfaccia grafica a tab, in cui ogni tab è un sottotipo di EditOrderSingleTab.
- **EditOrderSingleTab** (derivato da **QWidget**): istanziato in EditOrderTabPanel, questo widget ridefinito (incluso fra i widget che vengono visualizzati da EditOrderTabPanel), assieme ai suoi "sotto-widget", permette di personalizzare l'ordine base selezionato.
  - **EditOrderBeverageTab**.
    - **EditOrderCoffeeTab**.
    - **EditOrderTeaTab**.
  - **EditOrderFoodTab**.
    - **EditOrderSandwichTab**.
    - **EditOrderSweetTab**.
- **OrdersTabPanel** (derivato da **QTabWidget**): implementa un'interfaccia grafica a tab in cui ogni tab è un widget di tipo OrderList.
- **OrderList** (derivato da **QListWidget**): istanziato in OrdersTabPanel, visualizza gli ordini aggiunti al contenitore.
- **OrderListItem** (derivato da **QListWidgetItem**): istanziato in OrderList, visualizza un singolo ordine aggiunto al contenitore, in particolare i suoi dettagli e il suo numero in quell'istante.
- **EditQueuedOrderDialog** (derivato da **QDialog**): istanziato in QCafe_Barista, visualizza una finestra di dialogo temporanea per effettuare piccole modifiche ad un ordine indicato in un OrderListItem, previa la generazione di un evento "doppio clic" su di esso.

Chiamate polimorfe

Nella gerarchia di tipi del modello sono presenti metodi virtuali che permettono di effettuare chiamate polimorfe.

Innanzitutto, la classe base Order contiene il distruttore virtuale **~Order()** (per eliminare correttamente dalla memoria oggetti puntati da puntatori polimorfi, come quelli nel contenitore). Ma contiene altri metodi virtuali,

che sono: **double price() const**, **Order* clone() const** e **string orderDetails() const** virtuali puri (il primo deve restituire il prezzo del prodotto, ma lo si può sapere solo quando l'oggetto è istanziato; il secondo deve clonare l'oggetto di invocazione restituendo un puntatore polimorfo; il terzo deve restituire in formato testuale i valori dei campi dati dell'oggetto di invocazione). È presente infine un ulteriore metodo virtuale **QCafe_type className() const** il cui compito è restituire il tipo dinamico dell'oggetto d'invocazione senza dover effettuare un controllo mediante dynamic_cast o typeid, troppo onerosi dal punto di vista computazionale (**QCafe_type** è un tipo enumerativo definito all'interno namespace **QCafe**).

Altri metodi polimorfi sono presenti nella classe astratta Food: **bool isVegan() const**, **bool isVegetarian() const** e **bool isGlutenFree() const**. Essi sono implementati nelle sottoclassi Sandwich e Sweet e restituiscono true/false in base ai valori assunti dai loro rispettivi campi dati (per esempio, se *s* è un puntatore ad un oggetto di tipo Sandwich*, l'invocazione *s->isGlutenFree()* restituirà true se il valore del campo dati *_bread* è *GLUTEN_FREE*, false altrimenti).

Formato dei file di caricamento/salvataggio dei dati del contenitore

Per memorizzare i dati presenti all'interno del contenitore si è deciso di utilizzare il formato **XML**. La classe che implementa questa funzionalità è **XMLFileIO** e sfrutta le classi del framework Qt denominate **QXmlStreamReader** e **QXmlStreamWriter**, rispettivamente per la lettura e per la scrittura di file in formato XML, definito come segue:

1. un documento XML generato dall'applicativo inizia con un tag di intestazione XML;
2. continua con un tag **<root>** che delimita tutti i successivi, fino a chiudersi a fine file con **</root>**;
3. ogni elemento del contenitore scritto sul file è racchiuso in un tag ***<tipo/>***, self-closing, con ***tipo*** corrispondente al tipo dinamico dell'oggetto a cui si riferisce il tag. All'interno del tag sono elencati sottoforma di attributi tutti i campi dati dell'oggetto che servono alla creazione dell'oggetto nel caso il file venga aperto in lettura dall'applicativo (assieme al campo dati relativo al prezzo, non utile ai fini della creazione di un oggetto), con il loro valore (in particolare, quelli booleani sono presenti solo se il loro valore è true).

Ad esempio, si supponga che sia stato istanziato da qualche parte all'interno del codice sorgente un oggetto *q* di classe Qontainer e un oggetto *x* di classe XMLFileIO.

Questa porzione di codice:

```
q->push(new Coffee_Milk("Cappuccino", Coffee_Milk::SMALL, Coffee_Milk::ARABICA,
Coffee_Milk::SUGAR, "Tommaso", Coffee_Milk::TAKE_AWAY));
q->push(new Sweet("Muffin", Sweet::WHIPPED_CREAM, Sweet::CHOCOLATE, Sweet::HEAT,
"Tommaso", Coffee_Milk::TAKE_HERE, 12.5));
x->writeOnFile(q);
```

genera il seguente file XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--QCafe save file. Do not modify this file's content.-->
<root>
    <coffee_milk product-name="Cappuccino" customer-name="Tommaso" price="1.5" take-
away="1" size="S" sweetener-type="Sugar" variety="Arabica"/>
    <sweet product-name="Muffin" customer-name="Tommaso" price="2.8" discount-
amount="12.5" heat-product="1" flavour="Chocolate" whipped-cream="1"/>
</root>
```

Nota: nei file consegnati è presente un file qcafe_savefile_test.xml contenente alcuni ordini generati per testare l'applicativo.

Istruzioni di compilazione

Il progetto è stato realizzato in varie macchine, di mia proprietà e dell'Università. Le prime hanno SO Windows 10 (Versione 1903), compilatore g++ (versione 5.3.0) e software GNU Make (versione 4.1), framework Qt (versione 5.9.5) e software qmake (versione 3.1). Le seconde hanno SO Ubuntu 16.04 (versione 16.04.06 LTS), compilatore g++ (versione 5.4.0) e software GNU Make (versione 4.1), framework Qt (versione 5.5.1) e software qmake (versione 3.0).

Il file QCafe.pro ottenuto dal comando qmake -project non permette la compilazione del codice sorgente. Per questo motivo, il progetto include anche questo file, con il comando aggiuntivo per permettere di utilizzare le funzionalità di C++11 **QMAKE_CXXFLAGS += -std=c++11**.

Per compilare il progetto è quindi sufficiente posizionarsi nella directory contenente i file e invocare i seguenti comandi:

```
qmake
make
```

Tempo richiesto per lo sviluppo

Attività	**Tempo**
Sviluppo dell'idea progettuale, progettazione della gerarchia di tipi, progettazione del container e dello smart pointer, progettazione dell'interfaccia utente	2h
Apprendimento di Qt: tutorato e documentazione (quest'ultima particolarmente in itinere)	6h
Implementazione del contenitore e dello smart pointer	15h
Implementazione del modello	2h
Implementazione del controller e della scrittura su file	5h, 2h
Implementazione della vista (inclusi i widget ridefiniti)	10h
Test, debug, correzione di errori e rifiniture	10h

Per un totale di tempo impiegato per lo sviluppo di 52 ore. Le due ore in più sono dovute alla sistemazione dell'interfaccia grafica, della ricerca di errori nel codice sorgente e attività di documentazione del codice sorgente.