Relazione progetto Programmazione a Oggetti
A.A. 2019/20

Gabriel Bizzo - 1170734 Marco Rosin - 1120673

Andrea Moscon - 1121217

Relazione di Rosin Marco

UNIVERSITÀ DEGLI STUDI DI PADOVA | DIPARTIMENTO MATEMATICA

Abstract e Funzionalità

Lo scopo del progetto è fornire un applicativo per informatizzare una pizzeria da asporto, implementando una gestione efficiente di inventario, menu e comande.

L'applicativo fornisce all'utente le seguenti funzionalità:

- Inserimento, rimozione e modifica di un articolo dal menù

- Inserimento, rimozione e modifica (in modo *smart*) di un ingrediente dall'inventario.
- Inserimento, rimozione e modifica (in modo *smart*) di una comanda da una bacheca apposita.
- Una funzionalità di *contabilizzazione* per calcolare il guadagno/perdita della pizzeria in un determinato periodo di tempo.
- Una funzionalità di salvataggio e caricamento di tutti i dati del programma.

Progettezione e descrizione delle gerarchie utilizzate

Chiamate Polimorfe

- `clone()`: Metodo virtuale puro della classe `Risorsa`; effettua una copia dell'oggetto di invocazione e ritorna un puntatore al nuovo oggetto creato.
- `salva()`: Metodo virtuale puro della classe `Risorsa`; effettua la *serializzazione* dell'oggetto di invocazione in un oggetto JSON, ricevuto come parametro, che verrà salvato su file.
- `carica()`: Metodo virtuale puro della classe `Risorsa`; effettua la *deserializzazione* dell'oggetto di invocazione, assegnando ai campi dati propri dell'oggetto i corrispondenti valori dell'oggetto JSON, ricevuto come parametro, proveniente da file.
- `modifica()`: Metodo virtuale puro della classe `Risorsa`; effettua la modifica dei campi dati dell'oggetto di invocazione in modo polimorfo, estraendo i nuovi valori dall'oggetto ricevuto come parametro.
- `getComposizione()`: Metodo virtuale puro della classe `Articolo`; restituisce una lista di oggetti di tipo `Consumabile` che rappresentano gli ingredienti presenti nell'articolo di invocazione.
- `getPrezzo()`: Metodo virtuale puro della classe `Articolo`; restituisce il prezzo di vendita di un articolo (calcolato in modo differente per ogni sottotipo di `Articolo`).
- `getSpesa()`: Metodo virtuale puro della classe `Consumabile`; restituisce la spesa sostenuta dalla pizzeria per l'acquisto dell'oggetto di invocazione (calcolato in modo differente per ogni sottotipo di `Consumabile`).
- `rendiEditabile()`: Metodo virtuale puro appartenente alla classe `tabellacomposita`

I/O

Il programma permette caricamento e salvataggio dei dati su file in formato JSON. È stato scelto il formato JSON in quanto:

- Ha una sintassi semplice e leggibile
- È notevolmente meno verboso rispetto al formato XML.

- Supporta nativamente il caricamento e salvataggio delle mappe non ordinate, che sono state usate in modo estensivo nel modello dell'applicazione

Per implementare tali funzionalità sono state usate le apposite classi della libreria Qt (`QJsonDocument`, `QJsonObject` ecc).

Al fine di aumentare l'estensibilità del codice invece di implementare un'unica funzione di salvataggio/caricamento nell'interfaccia pubblica del modello si è deciso di fornire a ogni classe i metodi `carica()` e `salva()`, che hanno rispettivamente il compito di inizializzare i campi dati dell'oggetto di invocazione con i valori presenti nel file (*deserializzazione*) e di salvare i valori dei campi dati dell'oggetto di invocazione su file (serializzazione). Con questa implementazione il progettista che vorrà estendere la gerarchia dovrà solamente fornire l'implementazione di questi metodi per fornire la funzionalità di I/O alle classi da lui aggiunte.

Il caricamento delle risorse avviene in automatico ad ogni apertura del programma attraverso il metodo `caricaRisorse()`. Il salvataggio è manuale e a discrezione dell'utente; tuttavia per prevenire accidentali perdite di dati non salvati alla chiusura del programma viene visualizzata una finestra di dialogo in cui è possibile scegliere se salvare i dati modificati prima di uscire o meno. È inoltre possibile salvare manualmente i dati del programma selezionando l'opzione "Inventario e Menu" all'interno della sezione "Salva" nella barra dei menu.

È inoltre possibile modificare manualmente il contenuto dei file JSON: tuttavia questo metodo è fortemente sconsigliato in quanto un errore di sintassi nel file provocherebbe un errato caricamento dei dati.

Istruzioni di compilazione

Il progetto è stato sviluppato utilizzando alcune funzionalità presenti in C++11 (`auto`, `nullptr` e `to_string`). Per questo motivo è stato necessario modificare il file .pro aggiungendo la direttiva "`CONFIG += c+11`".

Per compilare ed eseguire il programma sono quindi necessari i seguenti comandi (si suppone che il terminale sia aperto nella cartella del progetto):

1. `qmake bellaPadova.pro`
2. `make`
3. `./bellaPadova`

Suddivisione dei compiti - Ore di sviluppo

Mi sono occupato dell'implementazione dell'intera funzionalità di I/O e di alcune parti di modello, controller e *backend* tra i widget (sia principali che secondari) e il controller. Gabriel si è occupato dell'implementazione di tutti i Wizard e di alcune parti di modello, controller e dei widget principali della vista. Andrea si è occupato della progettazione e realizzazione dell'aspetto grafico della vista

(incluso il foglio CSS) e di alcune parti di modello, controller e dei widget secondari della vista.

È doveroso precisare che la suddivisione dei compiti appena esposta è approssimativa, in quanto la fase di sviluppo è stata integrata da *meeting* Zoom giornalieri in cui si è discusso l'andamento della codifica di ogni componente e la risoluzione di eventuali criticità riscontrate. Sebbene questa metodologia di sviluppo sia risultata più lenta rispetto a una strategia *divide-et-impera* ci ha permesso di completare più velocemente le singole componenti, riducendo così la necessita di effettuare test di integrazione e focalizzando la ricerca in itinere di eventuali bug al solo componente in sviluppo.

Lo sviluppo del progetto ha richiesto approssimativamente 60 ore di lavoro individuale così suddivise:

- Analisi preliminare dei requisiti: 2 ore
- Progettazione modello: 2 ore
- Progettazione GUI: 3 ore
- Apprendimento libreria Qt: 4 ore + 7 ore tutorato
- Codifica modello: 10 ore
- Codifica GUI: 20 ore
- Debugging e testing: 10 ore
- Stesura relazione: 2 ore

Il superamento del monte ore individuale è stato causato dall'apprendimento della libreria Qt e dalla risoluzione di alcuni bug difficili da identificare all'interno del *qontainer*.

Ambiente di sviluppo

Il progetto è stato sviluppato in un'ambiente così configurato:

- **Sistema Operativo:** Microsoft Windows 10 Pro N 64bit (ver 1903)
- **Compilatore:** MinGW 5.3.0 32bit
- **Qt** ver 5.9.5

Modello e container sono stati sviluppati utilizzando l'IDE Visual Studio Code, poiché la maggior familiarità con esso ha permesso di eliminare il tempo di adattamento a un nuovo IDE. Successivamente il progetto è stato migrato a QtCreator in quanto la continua configurazione manuale (scrittura del *makefile*, *linking* delle librerie Qt) è stata ritenuta ingestibile.

La fase di testing/debugging è stata svolta simultaneamente in ambienti Windows e Linux (quest'ultimo in particolare per la ricerca di *memory leak* tramite *Valgrind*).