Relazione del Progetto relativo l'insegnamento di Programmazione e Modellazione a Oggetti per la sessione Invernale 2020/2021.

Relatore:
Giacomo Di Fabrizio

Docente:
Prof. Saverio Delpriori

1) Specifica del software:

Il progetto consiste nella realizzazione di un software grafico per l'ordinazione di pasti e bevande.

Suddetto software dovrà consentire:

- La selezione delle pietanze (con relativa quantità) desiderate.
- La rimozione di pietanze che precedentemente erano state selezionate.
- La visualizzazione a schermo del numero di pietanze ordinate.
- La visualizzazione a schermo del prezzo totale che l'utente dovrà pagare.
- La simulazione del pagamento tramite carta di credito o tramite pagamento diretto alla cassa.

Il software salverà i dati selezionati in un file di testo su disco in formato JSON e, quando l'utente sceglierà di effettuare il pagamento, il software terminerà e il contenuto all'interno del file di testo verrà cancellato in modo tale da poter salvare le pietanze di una successiva ordinazione.

2) Studio del problema:

Dalla specifica insorgono diverse problematiche che necessitano di essere affrontate:

1. Creazione dinamica e a tempo di esecuzione degli oggetti
2. Rappresentazione sempre aggiornata degli elementi ordinati dall'utente e del prezzo totale
3. Utilizzare un unico database in cui memorizzare gli ordini
4. Rappresentare correttamente gli oggetti creati tramite una ListViewItem

Delle possibili soluzioni potrebbero essere l'utilizzo del Factory Method (conosciuto anche come Virtual Constructor) per realizzare a tempo di esecuzione i vari oggetti. Inoltre, una validissima opzione risulta quella di utilizzare insieme al Factory Method anche il Builder in quanto gli oggetti che vengono creati a tempo di esecuzione risultano abbastanza complessi da realizzare.

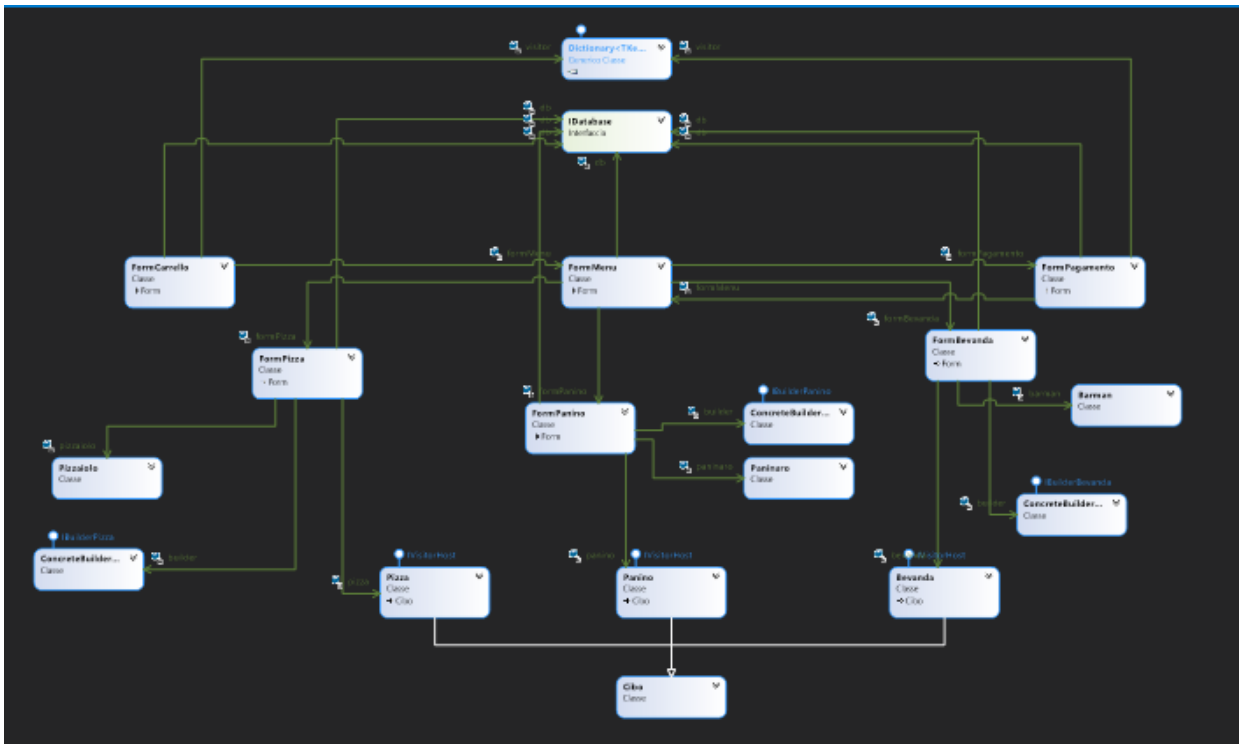
Per il secondo punto una valida opzione consiste nell'utilizzare il pattern Visitor e, in modo opportuno gli eventi messi a disposizione dal linguaggio C# e dal framework .NET.

Per avere un unico database in cui memorizzare ogni singolo dato la soluzione migliore è utilizzare il Singleton.

Per rappresentare gli oggetti in una ListView una possibile soluzione è implementare il pattern Adapter.

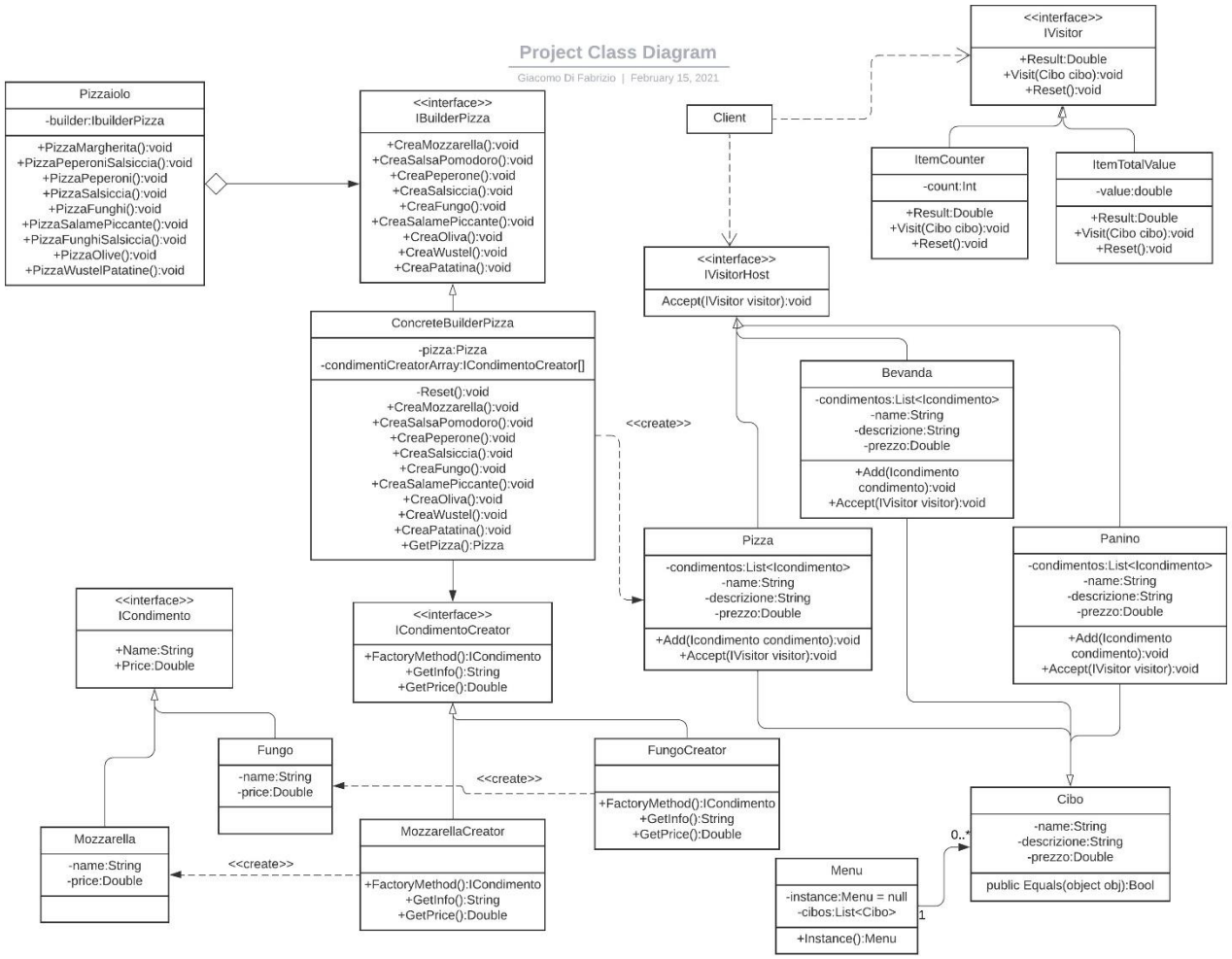
3) Scelte Architetturali:

Il seguente diagramma delle classi in UML è la più semplice rappresentazione delle varie relazioni che ogni classe ha l'una con l'altra (seguirà poi un successivo diagramma nel quale verranno esplicati maggiormente le relazioni e l'utilizzo dei pattern utilizzati)



Come è possibile notare dall'immagine, in questo diagramma delle classi si comprende soprattutto il come le Form comunicano tra di loro.
Ogni Form è collegata alla FormMenu, che fa da coordinatore delle restanti Form. Inoltre, si può ben notare che ogni Form fa riferimento allo stesso database. FormPagamento e FormCarrello implementano il Visitor, mentre FormBevanda, FormPanino e FormPizza implementano il Factory Method e il Builder andando a creare oggetti di tipo Bevanda, Panino e Pizza, che specializzano il tipo Cibo.

Nel diagramma successivo, invece, si andranno meglio a comprendere le soluzioni adottate per la realizzazione del progetto che abbiamo analizzato precedentemente.



Creazione dinamica degli oggetti

Al fine di creare oggetti complessi in maniera dinamica e a tempo di esecuzione si è optato per un utilizzo coordinato di Factory Method e Builder.
Questi ultimi, infatti, sono due pattern di tipo creazionale.
Il primo è stato utile per realizzare in maniera dinamica i vari elementi che vanno a costituire l'oggetto finale, mentre il secondo è stato utilizzato per separare la costruzione di un oggetto complesso dalla sua rappresentazione, in modo che lo stesso processo di costruzione possa creare rappresentazioni diverse.
Per motivi pratici, nel diagramma è possibile vedere l'implementazione di questi due pattern solo per quanto riguarda la creazione di oggetti di tipo Pizza, ma il processo è analogo sia per oggetti di tipo Panino e di tipo Bevanda.

Come è possibile notare, i condimenti vengono creati a tempo d'esecuzione dal metodo FactoryMethod() e poi vengono assemblati insieme l'uno con l'altro dall'oggetto di tipo Pizzaiolo, che utilizza un builder.
Più nello specifico l'oggetto di tipo Pizzaiolo utilizza un builder di tipo ConcreteBuilderPizza il quale all'interno ha un array di tipo ICondimentoCreateor[] con il quale va a creare i vari condimenti e li lega per formare l'effettiva pizza, che verrà restituita tramite il metodo GetPizza().

Rappresentazione sempre aggiornata degli elementi ordinati dall'utente e del prezzo totale

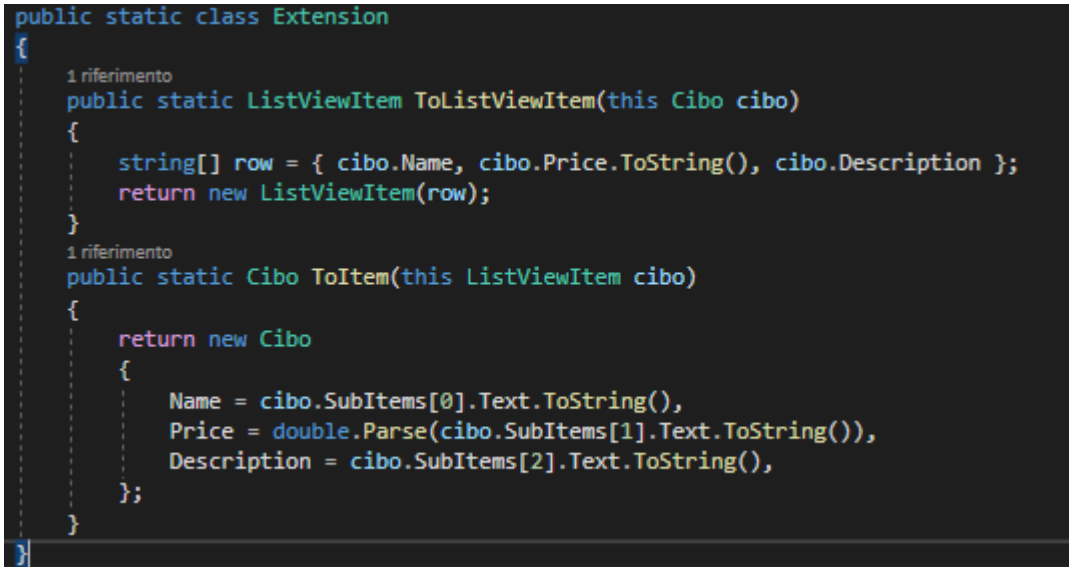
Per fare ciò, è stato implementato il pattern Visitor.
Questo perché l'intento era di definire le due nuove operazioni senza modificare le classi degli elementi su cui opera.
In particolare, le classi in questioni sono le classi Pizza, Panino e Bevanda.
Queste ultime, infatti implementano l'interfaccia IVisitorHost che possiede un metodo che, in gergo tecnico, accetta il visitor.
Le classi ItemValue e ItemTotalValue implementano l'interfaccia IVisitor che possiede il metodo Visit(Cibo cibo) con il quale appunto è possibile calcolare la quantità degli oggetti ordinati e il prezzo del conto finale.
Per quanto riguarda invece il continuo aggiornamento di questi ultimi, al posto di utilizzare il design pattern Observer si è preferito sfruttare il meccanismo degli eventi proprio del linguaggio C# in quanto più pratico e funzionale ai fini della progettazione del software.

Utilizzare un unico database in cui memorizzare gli ordini

Per fare ciò non esiste nulla di più pratico e funzionale del design pattern Singleton.
Esso infatti è stato utilizzato sia per realizzare un' unica istanza della classe Database in cui salvare e leggere i dati sia per realizzare un' unica istanza della classe Menu(vedi associazione uno a molti nel diagramma UML).
Queste due classi infatti possiedono il metodo GetInstance() che ci garantisce che di oggetti creati ce ne siano uno solo ed unico per classe.

Rappresentare correttamente gli oggetti creati tramite una ListViewItem

Per fare ciò sono stati utilizzati due metodi estensionali riferiti uno alla classe ListViewItem e uno alla classe Cibo.
Il primo ToListViewItem, che prende in ingresso un oggetto di tipo Cibo e restituisce un oggetto di tipo ListViewItem, è utilizzato per realizzare una riga di una tabella che compare a schermo in cui vengono descritti nome, descrizione e prezzo del cibo.
Il secondo ToItem, che prede in ingresso un oggetto di tipo ListViewItem e restituisce un oggetto di tipo Cibo, fa il viceversa, e viene utilizzato per rimuovere la riga selezionata dalla tabella che compare a schermo.
Per la rimozione di un oggetto dalla ListViewItem è stato inoltre necessario overridare i metodi Equals e GetHashCode della classe Cibo in modo tale da rendere comparabili le istanze della classe stessa.

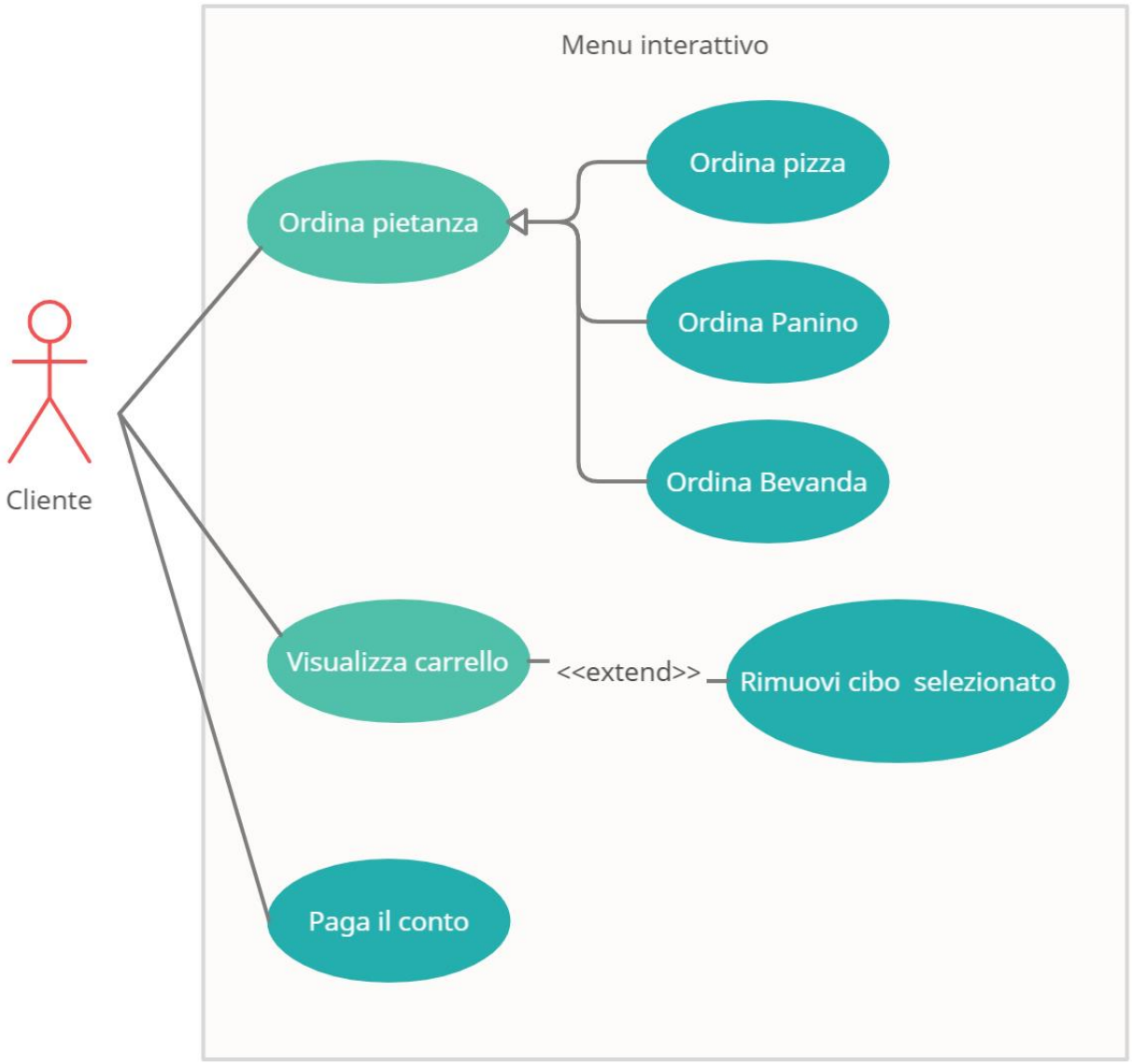


```
public static class Extension
{
    1 riferimento
    public static ListViewItem ToListViewItem(this Cibo cibo)
    {
        string[] row = { cibo.Name, cibo.Price.ToString(), cibo.Description };
        return new ListViewItem(row);
    }
    1 riferimento
    public static Cibo ToItem(this ListViewItem cibo)
    {
        return new Cibo
        {
            Name = cibo.SubItems[0].Text.ToString(),
            Price = double.Parse(cibo.SubItems[1].Text.ToString()),
            Description = cibo.SubItems[2].Text.ToString(),
        };
    }
}
```

4) Documentazione sull'utilizzo:

- L'applicativo è stato progettato tramite il framework grafico Windows Forms e Json.NET (inclusione Newtonsoft.Json).
- Il software è stato realizzato per il framework .NET core 3.1.
- Non sono necessari particolari parametri per l'esecuzione o la compilazione della soluzione.

5) Use cases UML:



Come è ben possibile notare dal diagramma dei casi d'uso, il cliente può:
ordinare le pietanze, visualizzare il carrello (rimuovendo se è sua intenzione ciò che non vuole più ordinare) e pagare il conto.
(Ricordo che il pagamento non è altro che una simulazione grafica e che i dati inseriti non verranno memorizzati in alcun luogo)