

Louis TEMPE

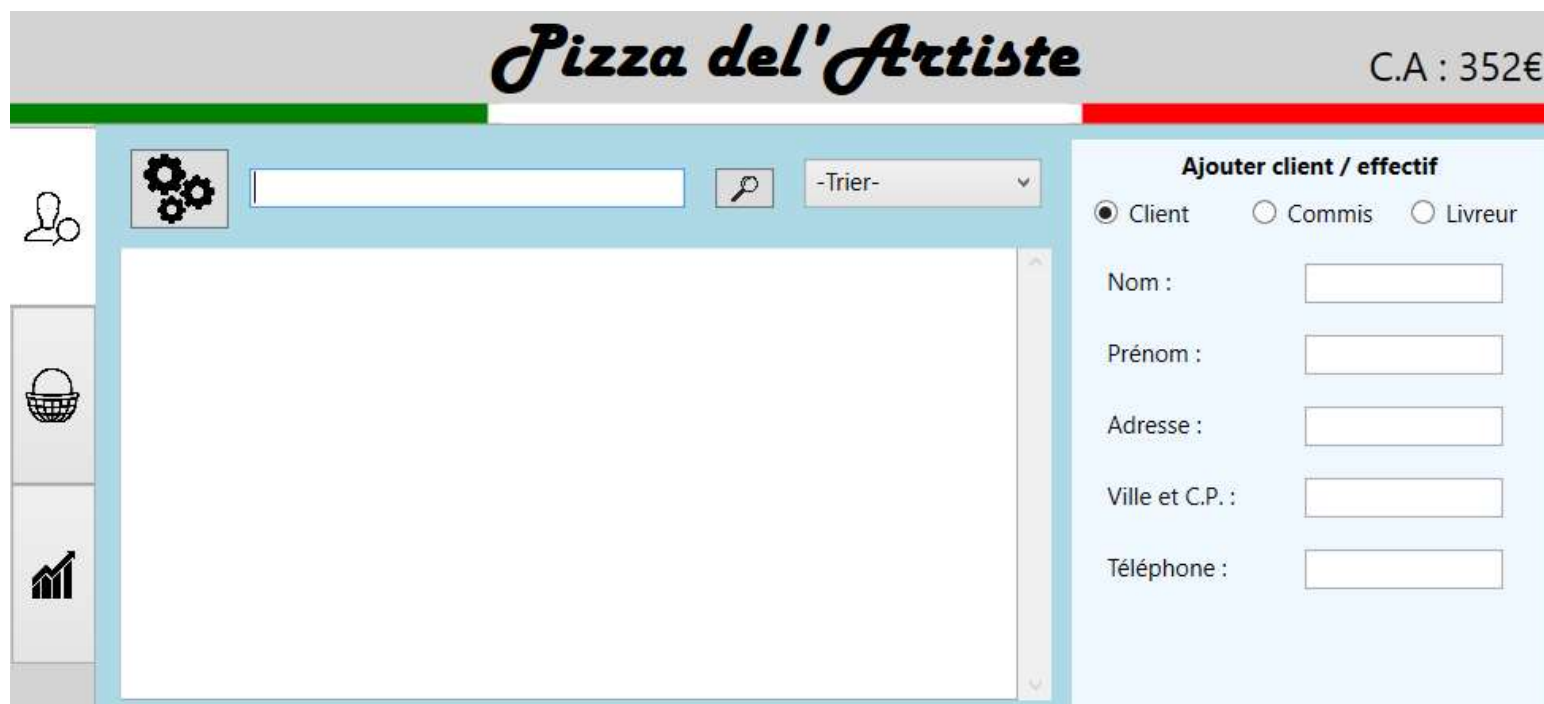
Programmation Orientée Objet - Avancée

A3 - Groupe C

Problème - Pizzeria




Rapport – Problème



Pizza del'Artiste



The screenshot shows the user interface of a web application for 'Pizza del'Artiste'. The header features the application name in a stylized font and the current total 'C.A : 352€'. The interface is divided into a left sidebar with icons for users, settings, a shopping basket, and a bar chart, and a main content area. The main area includes a search bar, a sorting dropdown menu, and a large empty container for a list of items. On the right, there is a form titled 'Ajouter client / effectif' with radio buttons for 'Client', 'Commis', and 'Livreur', and input fields for 'Nom', 'Prénom', 'Adresse', 'Ville et C.P.', and 'Téléphone'.

Pizza del'Artiste C.A : 352€

   -Trier- ▼

Ajouter client / effectif

☒ Client ☐ Commis ☐ Livreur

Nom :

Prénom :

Adresse :

Ville et C.P. :

Téléphone :

1 – Introduction

L'objectif de ce projet était d'informatiser le système de commande-production-livraison d'une pizzeria afin d'améliorer son rendement, le tout dans une solution C# sous forme d'interface graphique (WPF), en suivant un cahier des charges précis. C'est donc 30 bonnes heures (sans compter le temps passé à la création du diagramme UML et des TP effectués pour me familiariser avec WPF) que j'ai passé, seul, sur ce problème afin de créer une interface WPF dynamique qui réponde aux attentes de la pizzeria. Ce rapport de 4 pages vient expliquer la répartition des classes dans mon diagramme UML, le fonctionnement du programme, les difficultés rencontrées avec une petite auto-critique et une conclusion. En annexe se trouve un guide de l'utilisateur.

2 – Diagramme UML

Le diagramme est composé de 10 classes, dont 3 classes mères abstraites et une classe partielle, ainsi que 2 interfaces.

IComparable<in T> : Interface de Visual Studio nécessaire pour effectuer un tri sur une liste générique, avec la méthode *Sort* sans paramètres, d'une liste d'objets de cette interface. Son implémentation nécessite la méthode *int CompareTo(T other)* qui permet de comparer l'instance de cette classe avec l'objet de type T selon le critère défini. Pour les classes de mon projet, le type T est le même que celui de l'instance.

IIdentifiable : Interface que j'ai créé pour permettre, dans un premier temps, de donner une identité à chaque instance des classes qui l'implémente : elle est définie par une propriété *string Numero{get ;}* qui donne le numéro d'identité, le numéro de téléphone pour les *Personne*, et le numéro de commande pour les *Commande*. Elle est également définie par une méthode *string ToFile()*, qui retourne une chaîne de caractère destinée à être écrite sur une ligne de fichier texte. En effet, tous mes *Identifiables* doivent pouvoir voir leurs données être enregistrées dans un fichier au format .csv. Cette interface est très utile lorsque je veux exécuter plusieurs fois la même action sur des listes d'objets d'une classe différente.

IList : Elle n'est pas dans le diagramme et je ne l'ai reliée à aucune classe, mais je l'ajoute ici car je l'ai beaucoup utilisée. Interface de Visual Studio dont hérite toute liste générique. Très utile pour créer des tableaux de listes dans certains cas, et pour mettre une liste d'objets en paramètre d'une méthode, lorsque l'on souhaite que le cast puisse être modifié.

Par exemple, je ne peux pas faire :

```
void Methode(List<Personne> personnes) { personne.ForEach(p => Afficher(p)); }
```

puis appeler *Methode(clients)*; alors que si je définis de cette manière *void Methode(IList liste){...}*, je peux appeler *Methode(clients)* ou *Methode(Commande)* sans problème, donc ça m'a permis d'éviter beaucoup de redondance dans les lignes de code.

Personne (abstract) : Une personne est caractérisée par un nom, un prénom, une adresse, une ville, un numéro de téléphone et un nombre de commandes. Le numéro de téléphone est unique à chaque personne, c'est ce qui lui sert d'identifiant par implémentation de *IIdentifiable*. Concernant le nombre de commandes, je l'avais d'abord mis uniquement aux clients, mais je me suis dit que toutes les classes héritant de *Personne* allaient posséder un nombre de commandes (effectuées par le client, gérées par le commis, et livrées par le livreur).

On retrouve également 2 champs *static* : *viles* qui retourne une liste de toutes les villes de chaque instance créée, et *maxCommandes*, qui donne le nombre de commandes le plus grand associé à une personne parmi toutes les instances de la classe.

On retrouve une méthode d'affichage sous forme de chaîne *string ToString()*, ainsi qu'une méthode virtuelle d'écriture vers un fichier, *string ToFile()*, implémentée par l'interface *IIdentifiable*.

Client : Classe héritée de *Personne*, dont toute instance est caractérisée par une date d'adhésion à la pizzeria, une quantité d'achats cumulés, qui représente la quantité dépensée par le client dans la pizzeria. On retrouve également un champ *static* : *maxAchat*, qui renvoie la plus grande quantité d'achats cumulés effectué par un *Client* parmi toutes les instances de cette classe créées. Méthodes *string ToString()*, *ToFile()* de *IIdentifiable*, et *int CompareTo(Client c)*, qui compare avec un autre client sur le critère des achats cumulés.

Salarie (abstract) : Classe dérivant de *Personne*, dont chaque instance est caractérisée par un état (0 pour « Sur place », 1 pour « En congés », et 2 pour « Sur la route » dans le cas d'un livreur), et un salaire, calculé à partir d'une méthode abstraite dans cette classe. Méthodes *string ToString()*, *string ToFile()*, et *abstract double CalculerSalaire()*.

Commis : Classe héritée de *Salarie* dont chaque instance est caractérisée par une date d'embauche. On y retrouve les méthodes *string ToString()*, *string ToFile()*, *int CompareTo(Commis c)* de *IComparable* pour comparer avec un autre *Commis* sur le critère de l'ancienneté, *int Anciennete()* qui retourne l'ancienneté du *Commis* et *double CalculerSalaire()*. Le salaire est déterminé à partir de l'ancienneté du *Commis*. Selon le cahier des charges, c'est le *Commis* qui enregistre le *Client* lors de la création de la *Commande*.

Livreur : Classe héritée de *Salarie* dont chaque instance est caractérisée par un moyen de transport. On y retrouve également un champ *static* qui renvoie une liste de tous les moyens de transports utilisés par chaque instance de cette classe. Méthodes *double CalculerSalaire()*, *string ToString()*, *string ToFile()*, et *int CompareTo(Livreur l)*, pour comparer les livreurs sur le critère du moyen de transport. Le salaire est déterminé à partir du moyen de Transport du *Livreur*, le critère est un peu nul mais je n'ai rien trouvé de mieux pour faire intervenir le moyen de transport. Selon le cahier des charges, c'est le livreur qui récupère la *Commande*, et le paiement de la *Commande* par le *Client*.

Produit (abstract) : Un *Produit* est caractérisé par un nom, une taille, et un prix, déterminé depuis une liste triée de la *Pizzeria*. Méthodes *string ToString()* et *double DeterminerPrix()*.

Pizza et Boisson : 2 classes héritées de *Produit*, qui se différencient par le fait que la taille d'une *Boisson* est en fait un volume (selon l'affichage), et par le fait qu'une *Pizza* doit obligatoirement composer une commande.

Commande : Classe assez importante, d'une *Commande* caractérisée par un numéro de commande, une heure, une date, un prix, un état (0 pour « En préparation », 1 pour « En livraison » et 2 pour « Fermée »), et un booléen précisant que la commande a été payée ou non. A chaque commande est associée un *Client*, celui qui effectue la *Commande*, un *Commis*, celui qui la gère, et un *Livreur*, celui qui la livre, ainsi qu'une liste des *Produit* qui la compose. On a une relation d'agrégation avec ces 4 éléments. La commande possède au moins un *Produit*, plus précisément une *Pizza*, et un seul *Client*, *Commis*, et *Livreur*. Cependant, ces derniers peuvent exister sans être associés à une commande, l'inscription se faisant préalablement dans la *Pizzeria*. On retrouve également un champ *static*, *plusChere*, qui renvoie le prix de la *Commande* la plus chère parmi toutes les *Commande* déjà créées. Méthodes *string ToString()*, *string ToFile()*, *double CalculerPrix()* qui calcule le prix à partir des prix des *Produit*, et *int CompareTo(Commande c)*, qui compare avec une autre *Commande* sur le critère de l'état de la commande.

Pizzeria (partial) : *Main*, classe principale héritée de *Window* dans laquelle s'exécute la plupart du code, et qui relie l'ensemble des classes du diagramme à l'interface graphique WPF. La *Pizzeria* est composée d'une liste de *Client*, de *Commis*, de *Livreur*, de *Commande*, d'un chiffre d'affaires, et de 2 listes triées comprenant l'ensemble des *Pizza* et des *Boisson* avec leur prix selon les tailles. Elle comprend une méthode *string ToString()* que je n'utilise pas, mais qui peut toujours être utile pour connaître l'intégralité des infos de la *Pizzeria*. Il y a également des attributs correspondant aux noms des fichiers dans lesquels sont enregistrées les données de chaque élément de chaque liste, que je n'ai pas ajouté au diagramme. Enfin, elle comprend des dizaines de méthodes pour gérer toute l'interface graphique et les éléments de la *Pizzeria*, que je n'inclurai pas au diagramme, tant elles sont nombreuses.

Le diagramme a évidemment été créé en amont, avant la création de la solution C#, et beaucoup de modifications ont vu le jour au fur et à mesure que le projet a avancé, celui qui est présenté est la version finale.

3 – Fonctionnement du programme

Le programme est décomposé en plusieurs dizaines de méthodes que je ne vais évidemment pas toutes détailler, chaque méthode étant utile soit pour l'Initialisation, soit pour un des 3 modules créés (Client et Effectif, Commandes, et Statistiques). Concernant l'initialisation, la 1ere méthode qui est créée, appelée dans le constructeur au lancement du programme, initialise toutes les listes d'identifiables (clients, commis, livreurs, commandes), ou des listes triées (pizzas et boissons), et affecte aux attributs correspondant aux noms des fichiers, le nom des fichiers csv, pour ne pas avoir à les réécrire dans toutes les fonctions

qui les utilisent, surtout si je devais modifier le nom ou l'emplacement du fichier par la suite. Elle récupère ensuite les données de toutes les listes à partir de ces fichiers justement. Pour chaque personne, une ligne du fichier correspondant désigne une personne, avec ses attributs selon son type. Pour les commandes, chaque commande est désignée sur 4 lignes, la première avec son numéro, la 2^e ses informations (date, client/commis/livreur associés, état, etc.), la 3^e les pizzas qui la compose et leur prix, et la 4^e éventuellement les boissons qui la compose. Pour les produits, chaque ligne correspond à un produit avec son nom, et ses 3 prix en fonction de ses 3 tailles, qui sont rangés dans une liste triée qui leur est attribuée.

L'initialisation se poursuit en implémentant certains événements par défaut grâce à la lambda-expression, pour éviter d'avoir une dizaine de méthodes supplémentaires, l'implémentation se fait tout simplement sur une ligne en précisant `control.Event += (s, e) => { ... }`; pour ces quelques événements, avec `s` pour *object sender* et `e` pour l'argument. Evidemment, les événements avec un algorithme plus complexe sont implémentés par des méthodes plus bas dans le programme. Certaines méthodes sont initialisées aussi, on retrouve *CalculerCA()*, qui calcule et actualise l'affichage du chiffre d'affaires de la pizzeria chaque fois qu'une nouvelle commande est payée et fermée. Une méthode d'actualisation de l'affichage qui fait appel à plusieurs méthodes (tris, filtres, statistiques) est aussi appelée pour la première fois, et sera réappelée à chaque nouveau changement dans le programme (nouveau client, nouvelle commande, etc.)

Concernant l'enregistrement des données dans les fichiers csv, un bouton « *Enregistrer les données* » est visible en bas de la fenêtre avec un petit carré qui est soit vert, soit rouge, si les données ont été enregistrées ou non. Pour enregistrer les données, l'évènement click du bouton fait appel plusieurs fois à la méthode *EnregistrerIdentifiable* qui prend en paramètre un objet de l'interface *IList*, soit n'importe quelle liste générique, et le nom du fichier csv. Cette méthode utilise tout simplement la méthode *ToFile()* de chaque *IIdentifiable*.

Pour l'affichage des objets, chaque objet de chaque liste se trouve dans une *ListBoxItem*, avec la méthode *ToString()* qui leur est attribuée qui retourne toutes les données importantes de la personne ou de la commande sous forme de chaîne. Je n'ai pas utilisé de *DataBinding* pour ce faire, pour 2 raisons : la première est que j'ai appris trop tard son existence, et rendu compte de son efficacité, et la 2^{de} que les *ObservableCollection<>*, bien qu'ils ont de nombreux avantages, n'ont pas certaines méthodes intéressantes des listes génériques telles que le *Sort()*, même si ce problème aurait facilement pu être contourné, j'en ai conscience. A la place, la méthode *ActualiserListBox()* est appelée chaque fois que j'ai besoin d'actualiser l'affichage, et ça marche très bien comme ça. Comme les *ListBoxItem* sont créées dans des boucles, un moyen que j'ai trouvé pour récupérer les informations des Identifiables affichés, est tout simplement d'affecter à la propriété *Tag* de chaque *ListBoxItem* l'Identifiable correspondant, qui sera récupéré dès lors que l'utilisateur cliquera dessus, par exemple pour pouvoir modifier les informations d'un client. Pour les 2 premiers modules, il est possible de filtrer les identifiables selon certains critères en fonction de leur type, ou de les trier, ou bien de chercher un identifiable grâce à *string.Contains*.

Les méthodes pour filtrer les clients/commis/livreurs m'ont demandé beaucoup de temps, car je voulais pouvoir afficher les 3 listes en même temps ou séparément, et ajouter un critère de filtre spécifique à chaque classe lorsque seule une classe est cochée (par exemple, pouvoir filtrer les livreurs selon leur moyen de transport lorsque Livreurs est coché et que Commis et Client sont décochés, etc. J'ai d'abord fait une méthode longue et redondante avec plein de if/else, et j'ai par la suite utilisé des délégations pour diviser le nombre de lignes de code et rendre ça moins redondant et plus lisible, bien qu'on retrouve toujours quelques if/else. Plus d'infos dans le code source, tout y est détaillé.

Dans le module Commandes, on retrouve un affichage et des fonctionnalités assez similaire, qui m'ont demandé plus ou moins de temps. Quelques difficultés supplémentaires à faire en sorte qu'un salarié ne puisse pas être associé à une commande s'il est en congé, ou sur la route pour un livreur, et que l'état des salariés ne peut pas être modifié s'il est associé à une commande non fermée, mais je m'en suis sorti, et également pour faire en sorte que le client puisse ajouter ou enlever autant de produit qu'il veut à condition qu'il y ait au moins une pizza dans la commande. Cela a donné lieu à de nombreuses méthodes redondantes et des cast de cast assez moche, (que j'ai laissé en commentaire dans le code source), mais que j'ai réussi à bien alléger, notamment grâce à l'opérateur « `? :` » que j'aime beaucoup et que j'utilise pour tout depuis que je l'ai découvert.

Concernant le module statistique, il s'agit d'un simple affichage du nombre de commandes effectuées par client, gérées par commis, et livrées par livreurs, de la quantité d'achats effectués par clients dans la pizzeria, et du prix de chaque commande, avec une moyenne qui s'affiche plus bas, la même méthode est appelée chaque fois, pour éviter toute redondance de code, merci les délégations.

Par rapport aux délégations, j'en ai utilisé à plusieurs reprises, en créant 3 *delegate* différents notamment, que j'utilise pour filtrer selon les critères, pour enregistrer les données selon le type de l'identifiable, ou pour afficher une propriété selon la statistique désirée dans le module statistiques, mais j'ai également utilisé les délégations des méthodes associées aux listes génériques telles

que *Sort*, *ForEach*, en utilisant la lambda-expression, ou encore pour associer des événements aux contrôles comme dit plus haut. Une autre astuce que j'ai beaucoup utilisé est d'affecter des objets ou des conditions à la propriété *Tag* de certains contrôles (*sender.Tag* par exemple), pour mieux m'en sortir pour récupérer certaines données dans les événements.

Ajouts par rapport au cahier des charges : Un filtre pour les clients, commis, et livreurs, et pour les commandes. Possibilité de trier l'affichage de ces 4 listes. Le choix des critères de filtre et de tri varie en fonction des éléments sélectionnés (si seul un commis est sélectionné, possibilité de filtrer selon sa date d'embauche, et de trier selon son ancienneté, par exemple). Barre de recherche (ok ça prend 2 secondes à coder). Couleurs, drapeau de l'Italie. Menu des produits avec leurs prix selon les tailles déjà définies par la pizzeria dans le fichier *produits.csv*, avec affichage en temps réel du prix du produit lors de la sélection dans la création de la commande.

4 – Difficultés rencontrées et Auto-critique

Comme dit précédemment, j'ai très peu utilisé le *DataBinding* dans mon programme (on peut en voir à certains moments notamment pour les *Slider* et les *TextBox* dans les filtres), car j'ai commencé le projet avant leur découverte, et je pense que si on avait eu un peu plus de temps en TD consacré à tout ça, j'aurais peut-être pris le temps, que je n'ai plus à présent, de modifier mon code pour exploiter cette fonctionnalité. Cela n'a pas posé énormément de problème pour autant, mes objets sont bien actualisés en temps réel chaque fois qu'ils ont besoin de l'être, bien que je comprenne tout à fait que ça aurait pu alléger mon code sur certains points.

Beaucoup de lignes de code redondantes se sont retrouvées dans mon code, j'ai réussi à en alléger pas mal depuis, notamment grâce aux délégations et aux interfaces. J'ai laissé quelques dizaines de lignes de codes moche en commentaire pour avoir un aperçu de la différence avant/après, bien qu'on retrouve quelques lignes répétitives que je n'ai pas réussi à minimiser (pour créer une liste de nouveaux *Client*, de *Commis* et de *Livreur*, les constructeurs étant différents, je ne voyais pas comment alléger le code même si j'en avais très envie).

J'ai essayé de minimiser l'utilisation des *switch* dans mon code, et des *if* de partout, afin de ne pas rendre un code sale. Tous les noms des variables sont cohérents, il peut arriver que certaines lignes de code soient longues, mais j'ai essayé de faire un code le plus facilement lisible et compréhensible, bien commenté. Dans les classes, les attributs qui ne sont pas retournés en propriété sont en *readonly*, j'ai fait en sorte d'éviter le plus possible de retourner des objets en propriété, par exemple, une commande n'a pas de propriété pour client, commis ou livreur, mais elle retourne les numéros de téléphone qui leurs sont associés. Les noms des contrôles sont généralement assez logiques (spX pour StackPanel, cbxX pour une ComboBox, btnX pour un Button, chkX pour une CheckBox, rbX pour un RadioButton, etc.).

Autre difficulté rencontrée : Respecter 4 pages minimum pour faire un rapport d'un projet qui m'a pris plusieurs dizaines d'heures tant j'ai de choses à dire. Toutes les informations complémentaires sont dans le code source pour en savoir plus, ou dans le guide utilisateur en Annexe.

5 – Conclusion et perspectives

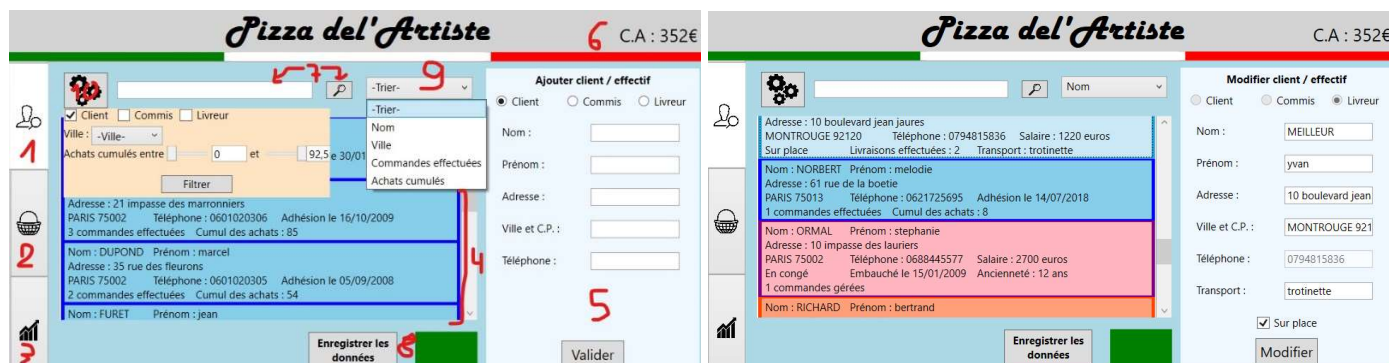
Pour conclure, je pense que l'objectif a été atteint, l'interface WPF fonctionne correctement, je ne vois aucune incohérence dans ce que j'ai fait, deux personnes ne peuvent pas avoir le même numéro, impossible de modifier l'état d'un salarié associé à une commande non fermée, on ne revient pas en arrière sur l'état d'une commande, etc. Evidemment, c'est un projet sans fin, avec un peu plus de temps je pourrai ajouter d'autres fonctionnalités comme un programme de fidélité, un module pour modifier les prix des pizzas, ajouter des quantités de produits disponibles, exploiter les fonctionnalités du *DataBinding*.

Je recommande fortement la lecture du cours et l'assimilation par les exercices, de ce MOOC : <https://pimo-wpf.netlify.app/>, fait par des étudiants de l'ESILV, que j'ai pris le temps d'étudier, en faisant intégralement les 3 premiers chapitres ainsi que les exercices/mini-projets avant de me lancer dans le projet de la Pizzeria. Bien sûr, une autre recommandation importante : <https://docs.microsoft.com/fr-fr/dotnet/csharp/> la documentation officielle de C# par Microsoft, que j'ai fini par ajouter à mes favoris, à lire sans modération.

6 – Annexe : Guide utilisateur

L'affichage a été prévu pour une fenêtre de 800 de long et 450 de haut, bien que la modification de sa taille ne pose pas trop de problèmes.

I – Module Effectif / Client



1 – Accès module effectif / client (ici ouvert)

2 – Accès module commandes

3 – Accès module statistiques

4 – Liste des clients, commis, et livreurs de la pizzeria, selon les critères des filtres et du tri. Par défaut, seuls les clients sont affichés (image 1). Clients en bleu, Commis en violet, et Livreurs en Orange-Rouge. En cliquant sur une personne, le formulaire d'inscription à droite s'actualise en un formulaire de modification (image 2). Il est alors possible de modifier les caractéristiques d'une personne en modifiant ses champs. Il n'est pas possible de modifier la fonction de la personne (Client, Commis ou Livreur). Le numéro de téléphone est unique et ne peut pas être changé. Un commis ou un livreur ne peut pas voir son état (sur place, en congés ou en livraison) modifié s'il est déjà associé à une commande en cours (non fermée). Appuyer sur « Modifier » actualise (mais n'enregistre pas) les données. Pour enregistrer, voir 8. Si vous souhaitez revenir au formulaire d'inscription sans modifier la personne, re-cliquez sur elle dans la liste tout simplement.

5 – Formulaire d'inscription. Choisissez la fonction de la personne que vous ajoutez (Client, Commis, Livreur). Entrez le nom, prénom, l'adresse, la ville et code postal, le numéro de téléphone de la personne, et éventuellement son moyen de transport si vous avez coché le bouton « Livreur » en haut du formulaire. Le numéro de téléphone est unique, et il doit contenir 10 numéros, pas de lettres. Tous les champs doivent être remplis. Dans le cas contraire, le logiciel ne vous laissera pas ajouter la personne (vous pouvez essayer). Bouton « Valider » pour ajouter la personne à la liste correspondante. Si les filtres correspondent à son identité, il apparaîtra automatiquement sur la liste des personnes affichées (en 4). Les majuscules sont gérées automatiquement.

6 – Chiffre d'affaires de la pizzeria, qui s'actualise chaque fois qu'une commande a été honorée.

7 – Barre de recherche, tapez n'importe quel mot ou bout de mot, et seront affichées toutes les personnes correspondantes (tapez paris pour afficher toute personne habitant paris, un prénom, « mar » pour afficher marcel, quelqu'un qui habite rue des marronniers, ou marie, etc.) Appuyez ensuite sur ENTREE pour effectuer la recherche, ou cliquez sur le bouton de recherche.

8 – Indicateur vert ou rouge et bouton « Enregistrer les données ». Par défaut l'indicateur est vert. Lorsque vous inscrivez une nouvelle personne / commande, ou que vous le mettez à jour, le carré devient rouge. Cela signifie que les nouvelles données n'ont pas été enregistrées dans les fichiers csv. Appuyez sur le bouton « Enregistrer les données » permet d'enregistrer toutes les modifications depuis la dernière fois que l'indicateur était vert, dans les fichiers, et l'indicateur repasse au vert.

9 – Tri. Cliquez pour ouvrir le volet des tris, et choisissez de trier l'ensemble des personnes par nom (alphabétique), par ville (alphabétique), ou par nombre de commandes effectuées/gérées/livrées. Par exemple, sur l'image 2, les clients, commis et livreurs sont triés par nom. Si, dans les filtres (10), Client est coché mais pas Commis ni Livreur, l'option trier par quantité d'achat cumulés vient s'ajouter. De même si vous ne cochez que Commis, vous pourrez trier par Ancienneté, et enfin, en ne cochant que Livreur, vous pourrez trier par moyen de transport (alphabétique).

10 – Filtre. Possibilité de filtrer les personnes à afficher, pour n'afficher que les clients, que les commis, que les livreurs, ou plusieurs catégories en même temps, ou les 3. Possibilité de trier par ville (et par arrondissement). Si seuls les clients sont cochés, possibilité de trier les clients en fonction dans leur quantité d'achat cumulés : défilez le slider de gauche pour obtenir une quantité minimum et celui de droite pour une quantité maximum. Seuls les clients ayant dépensé une quantité comprise entre ces 2 bornes seront affichés lorsque vous appuierez sur ENTREE ou cliquerez sur le bouton Filtrer. Vous pouvez également définir les bornes inf et sup en écrivant directement dans le champ. Si seuls les commis sont cochés, possibilité de filtrer selon une date d'embauche, de la forme jj/mm/aaaa. Effectuez un clic droit ou un double-clic pour effacer le « jj/mm/aaaa » écrit par défaut, et entrez une date avec ce format. Si seuls les livreurs sont cochés, possibilité de les filtrer selon leur moyen de transport. Si vous inscrivez un nouveau livreur avec un nouveau moyen de transport, celui-ci viendra s'ajouter automatiquement à la liste des moyens de transports disponibles au filtre.

II – Module Commande

The image shows two screenshots of the 'Pizza del'Artiste' software interface. The left screenshot is the 'Ajouter Commande' (Add Order) screen. It features a header with the restaurant name and a total amount of 352€. Below the header, there are filters for Client, Commis, and Livreur. A list of items to add is shown, including pizzas and drinks. The right screenshot is the 'Actualiser Commande' (Update Order) screen. It shows a list of items to update, including pizzas and drinks. The interface is designed for managing orders in a pizzeria.

1 – Numéro de commande. Correspond à l'identifiant de la commande à créer. Il se définit automatiquement comme la 1^{re} commande et ne peut pas être modifié.

En dessus se trouvent des panneaux déroulants pour choisir le client qui effectue la commande, le commis qui la gère, et le livreur qui la livre. Les salariés en congé n'apparaîtront pas dans le panneau déroulant. De même, les livreurs déjà en livraison n'apparaîtront pas sur le panneau déroulant. Cependant, tant que les commandes qu'il gère sont en préparation, il peut être affecté à plusieurs commandes.

2 – Nombre de pizzas. Par défaut, le compteur est à 1. Appuyez sur le bouton + pour ajouter une pizza à la commande, et – pour en enlever. Lorsque vous appuyez sur –, le compteur reste bloqué à 1. Une commande doit compter au moins une pizza.

3 – Sélection de la pizza, taille et prix. Sélectionnez une des pizzas proposées par la pizzeria dans le 1^{er} panneau déroulant, et une taille dans le 2nd. Lorsque les 2 sont sélectionnés, le prix apparaît. Vous pouvez le modifier à tout moment tant que la commande n'a pas été validée. Si vous ne sélectionnez pas une pizza ou une taille, la commande ne pourra pas être validée.

4 et 5 – Même principe que 2 et 3, à la différence qu'ici, on choisit une boisson, et le compteur est par défaut à 0 (et non 1). Une commande peut être composée de 0 boissons.

6 – Valider. Appuyez sur ce bouton lorsque tous les champs sont remplis. La commande passera automatiquement « en préparation ». Le chiffre d'affaires de la pizzeria, le nombre de commande de chaque personne qui lui est associé, et la quantité d'achats cumulés du client n'augmentent pas pour l'instant.

7 – Recherche. Même principe que pour le module client effectif (I – 7). A droite se trouve le filtre. Vous pouvez trier les commandes selon 4 critères : Le numéro de commande, la date d'ajout, le prix ou l'état de la commande (En préparation, en livraison, fermée – honorée ?)

8 – Filtres. Possibilité de filtrer les commandes par client, commis ou livreur. Il s'agit d'une union. En sélectionnant un commis et un client, vous verrez apparaître toutes les commandes associées au client, et toutes les commandes associées au commis.

Possibilité de filtrer selon le prix, bougez les slider ou écrivez dans les champs prévus pour définir les bornes inf et sup.

Possibilité de filtrer selon la date de création.

Possibilité de n'afficher que les commandes selon un état particulier, et possibilité d'afficher les commandes honorées, non honorées ou toutes. Cliquez sur « Filtrer » pour lancer le filtre.

9 – Les commandes sont affichées selon le filtre et le tri que vous définissez. Leur couleur varie en fonction de leur état (Rouge pour fermée et non honorée, vert pour fermée et honorée, jaune pour en préparation, et rose pour les commandes en livraison. En cliquant sur une commande non fermée, vous avez la possibilité de modifier son état.

10 - Une commande en préparation peut passer En livraison, ou Fermée. En la passant En livraison, le livreur associé à la commande ne pourra plus être associé à une nouvelle commande tant que toutes les commandes qu'il doit livrer ne seront pas fermées. Si vous cochez Fermée, la commande sera considérée comme non honorée et perdue. En sélectionnant une commande En livraison, il est possible soit de la Fermer sans l'honorer (en cliquant ne cliquant pas sur le bouton « Commande non honorée » (11)), la commande sera considérée comme perdue, soit de l'honorer en cliquant sur le bouton 11. Dans tous les cas, il faudra ensuite appuyer sur le bouton 12 : « Actualiser », pour mettre à jour l'état de la commande. L'indicateur de sauvegarde passera automatiquement au rouge et vous pouvez cliquer à tout moment sur « Enregistrer les données » pour mettre à jour les fichiers.

Une commande en livraison ne peut pas repasser à l'état En préparation.

Une commande fermée ne peut plus être mise à jour.

Une commande honorée permet d'augmenter le nombre de commandes effectuées d'un client, gérées d'un commis et livrées d'un livreur, et le cumul des achats du client, et d'augmenter le chiffre d'affaires de la Pizzeria.

Si vous ne souhaitez plus actualiser la commande, re-cliquez sur la commande dans la liste pour revenir en arrière.

II – Module Statistiques

Pizza del'Artiste			C.A : 352€
Nombre de commandes	Achats cumulés	Prix des commandes	
DURAND alexandre 3	DURAND alexandre 55,5	1 28,5	
FERRAND sylvie 3	FERRAND sylvie 85	2 16	
DUPOND marcel 2	DUPOND marcel 54	3 13,5	
FURET jean 3	FURET jean 92,5	4 33,5	
GABRIAUX cecile 1	GABRIAUX cecile 14	5 27	
DESJARDINS lucille 3	DESJARDINS lucille 43	6 25	
NORBERT 1	NORBERT 8	7 32	
Moyenne : 2,29 commandes	Moyenne : 50,29€	Moyenne : 22,55€	

Vous pouvez ici observer les statistiques de la pizzeria ainsi que la moyenne en bas. Cliquez sur le panneau déroulant à droite de « Nombre de commandes » pour switcher à Commis ou Livreur.