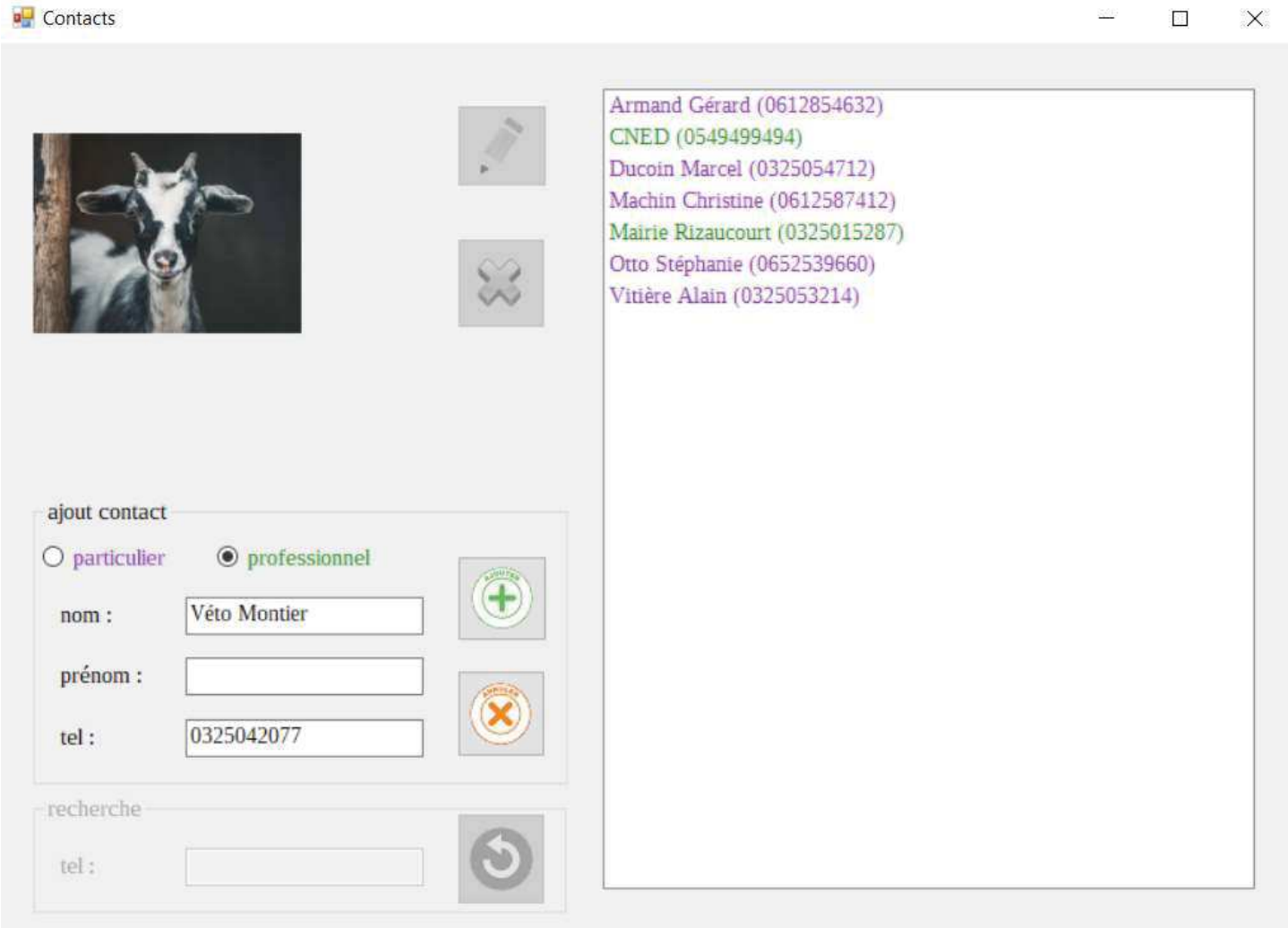
Fiche de Synthèse : Gestion des Contacts *Décembre 2019*

Développement :
Il s'agit d'une application de gestion des contacts, avec leur qualité (particulier ou professionnel), une photo et leurs coordonnées téléphoniques.



Cadre :
Il s'agit d'un TP proposé dans le cadre du cours "Bases de la Programmation ", par Elisabeth Martins Da Silva.

Support :
Développé en C# avec Visual Studio Community 2019.

Contraintes :
Une sauvegarde par sérialisation doit être faite automatiquement.

Difficultés rencontrées :
A la création d'un nouveau contact, une photo par défaut était proposée, différente selon si le contact à ajouter est un particulier ou un professionnel.
Or, dans mon cas, lorsque je modifiais un contact existant, la photo par défaut remplaçait la photo que j'avais choisie pour ce contact.
Pour corriger ça, il a " suffit " de rajouter une condition à l'évènement de la modification de la sélection du bouton radio correspondant :

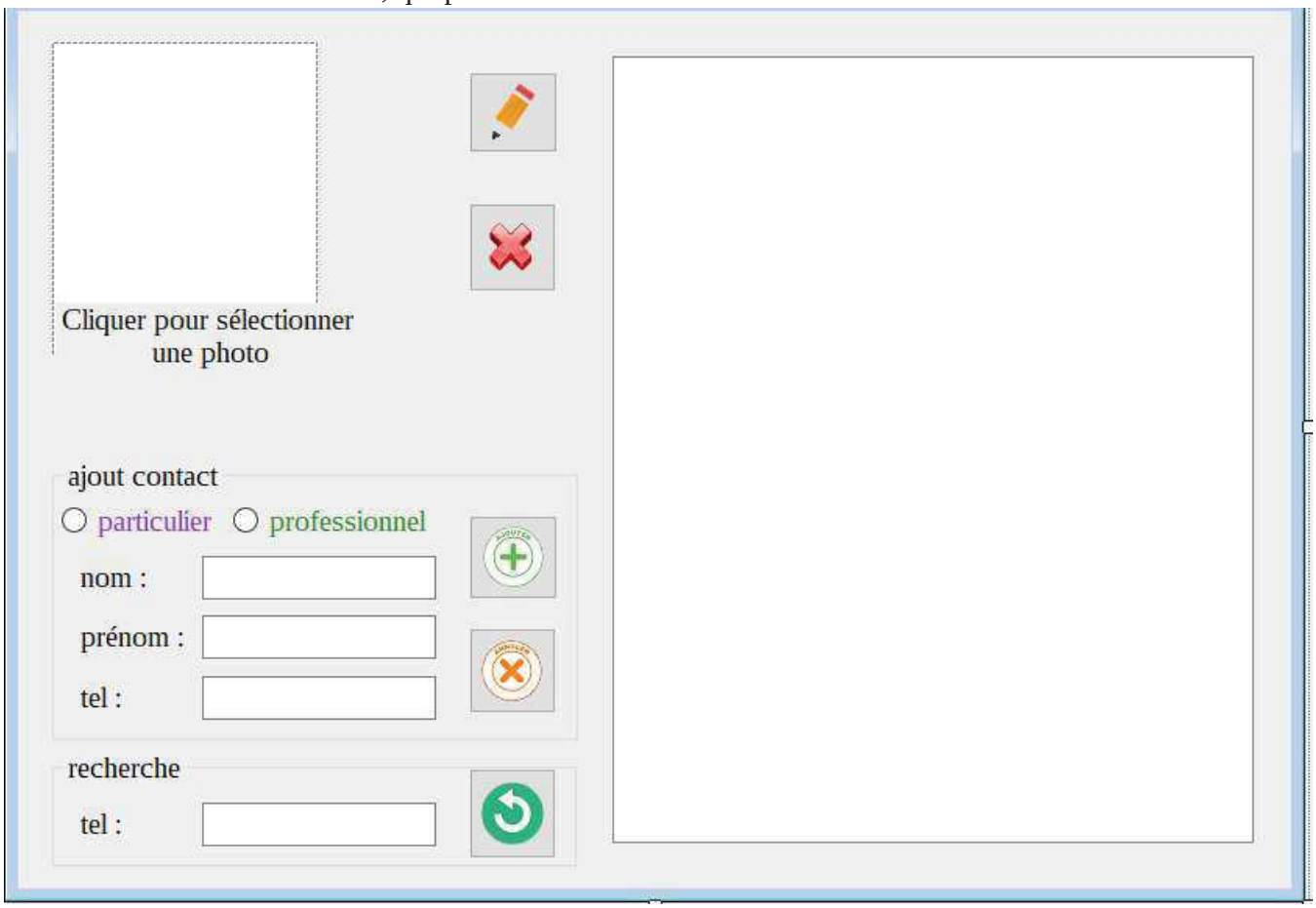
```
if (rdbProfessionnel.Checked && btnModifier.Enabled)
{
```

Description détaillée du développement

J’ai suivi les étapes suivantes pour le développement:
- la création de l’interface de l’application
- étude des classes métier fournies
- chargement de la fenêtre
- ajout d’un contact
- modification d’un contact
- suppression d’un contact
- recherche par tél
- mise en couleurs

1. La création de l’interface

J’ai utilisé l’outil de Visual Studio, qui permet de “ dessiner ” ce dont on a besoin.



Le code correspondant est généré automatiquement.

2. Les classes métier

3 classes métier nous étaient fournies : la classe abstraite Contact, et les classes filles Professionnel et Particulier

```
public abstract class Contact
{
    private string nom;
    private string tel;
    private Image photo;
    2 références
    public Contact(string nom, string tel, Image photo)
    {
        this.nom = nom;
        this.tel = tel;
        this.photo = photo;
    }
    5 références
    public string getNom()
    {
        return this.nom;
    }
    8 références
    public string getTel()
    {
        return this.tel;
    }
    4 références
    public Image getPhoto()
    {
        return this.photo;
```

3. Chargement de la fenêtre

Une classe abstraite Serialise nous a aussi été fournie, qui comporte les méthodes statiques Sauve() et Recup().

```
abstract class Serialise
{

    //--- S♦rialisation ---
    1 référence
    public static void Sauve(string fichier, Object objet)
    {

        //--- si le fichier existe, il faut le supprimer ---
        if (File.Exists(fichier))
        {
            File.Delete(fichier);
        }

        //--- cr♦ation du flux pour l'♦criture dans le fichier ---
        FileStream flux = new FileStream(fichier, FileMode.Create);

        //--- cr♦ation d'un objet pour le formatage en binaire des informations --
        BinaryFormatter fbinaire = new BinaryFormatter();

        //--- s♦rialisation des objets de la collection
        fbinaire.Serialize(flux, objet);
```

Au chargement, le fichier dont le chemin a été préalablement défini par l'utilisateur se charge s'il a été créé (sinon il se créé à ce moment là) et la mise à jour de la liste des contacts se fait, afin que la liste se remplisse.

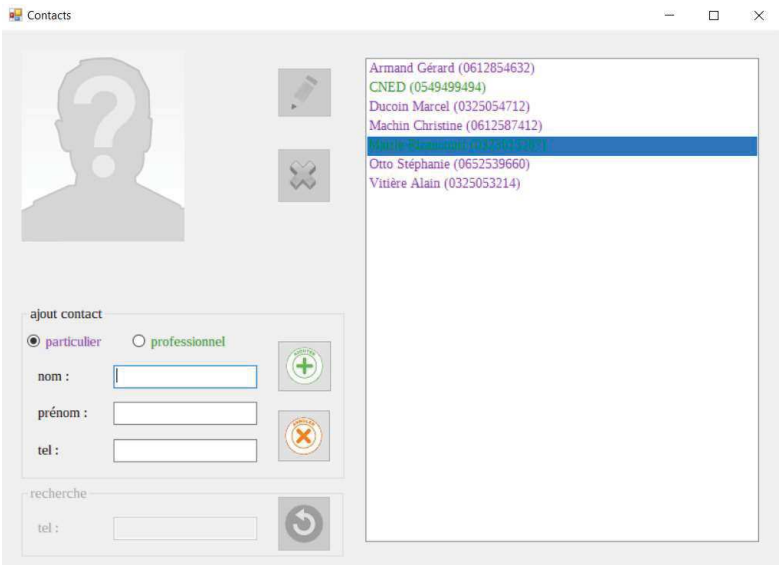
```
private void Form1_Load(object sender, EventArgs e)
{
    // récupération de la liste de contacts :
    Object sauve = Serialise.Recup(fichier);

    if (sauve != null)
    {
        lesContacts = (List<Contact>)sauve;
        majLstContact(0);
    }
}
```

```
// Mise à jour de la listBox lstContact avec choix de la ligne sélectionnée:
6 références
private void majLstContact(int indLesContacts)
{
    lstContact.Items.Clear();

    for (int i = 0; i < lesContacts.Count; i++)
    {
        lstContact.Items.Add(lesContacts[i].infosContact());
    }
    if (indLesContacts < 0)
    {
        lstContact.SelectedIndex = -1;
    }
    else
    {
        lstContact.SelectedIndex = indLesContactsToListbox(indLesContacts);
    }

    // Sauvegarde de la liste lesContacts :
    Serialise.Sauve(fichier, lesContacts);
}
```

Cette procédure sera répétée à chaque ouverture, mais aussi à chaque ajout – suppression ou modification de contact.

4. Ajout d'un contact

En cliquant sur un bouton radio dans la zone d'ajout, certaines zones de saisies s'activent pour vous permettre d'ajouter un nouveau contact :



Une fois les information saisies, en cliquant sur le bouton + les informations s'ajoutent dans la liste de contacts. Il est possible d'annuler sa saisie en cliquant sur x

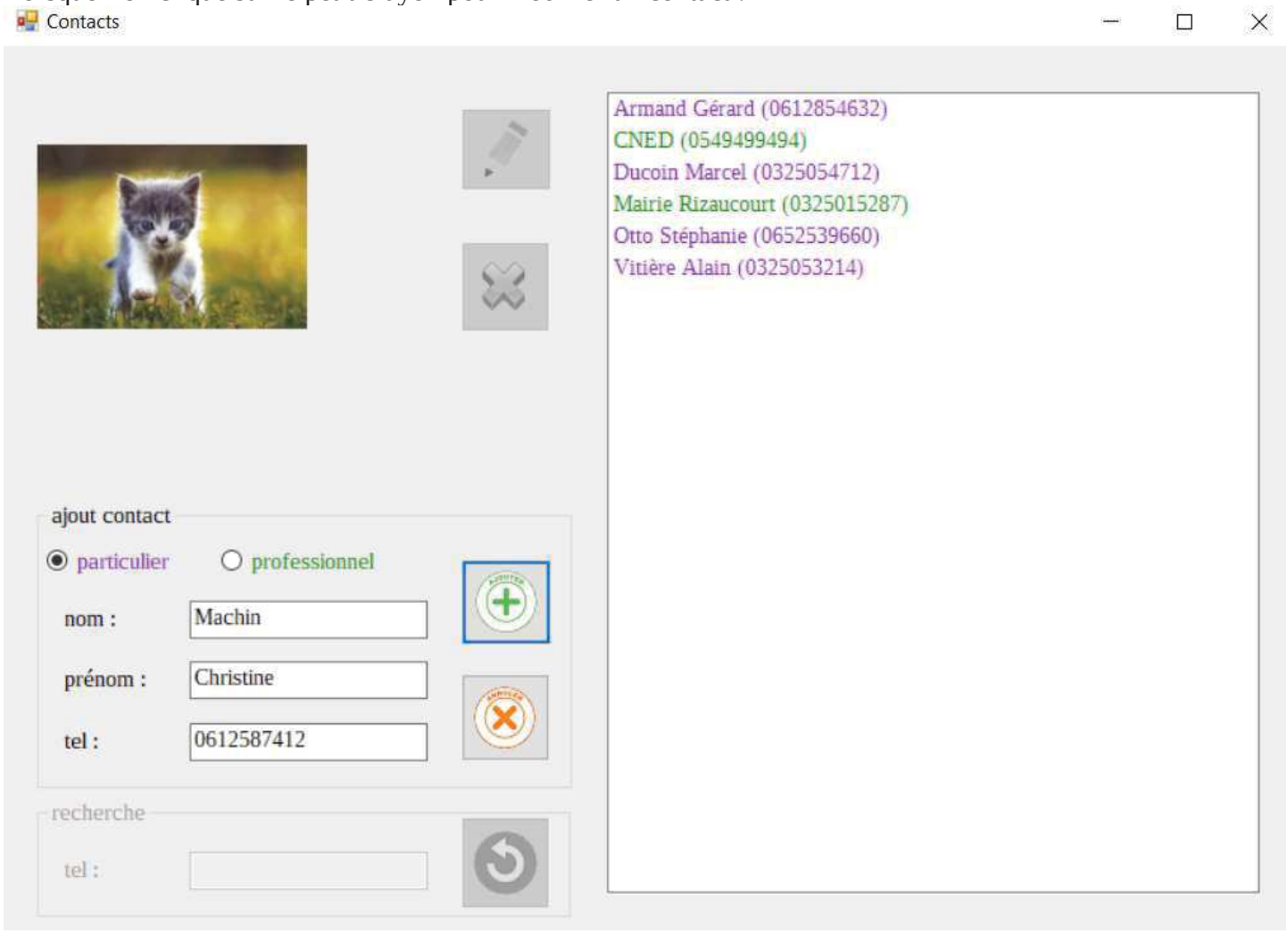
Le développement : si toutes les informations obligatoires sont saisies, alors, quand on clique sur le petit + : un nouveau contact est créée, un object instancié de la classe Particulier ou Professionnel selon le bouton radio qui a été coché, les zones de saisies redeviennent inactives, la liste des contacts est mise à jour en conséquences, les zones de saisies sont vidées, et enfin les boutons radios sont décochés.

J'ai écrit des procédures pour toutes ces étapes. Par exemple, pour la création d'un contact :

```
// Création d'un objet Particulier ou Professionnel, et ajout à la liste lesContacts
1 référence
private void creationContact()
{
    if (rdbParticulier.Checked)
    {
        lesContacts.Add(new Particulier(txtNom.Text, txtPrenom.Text, txtTel.Text, pctPhoto.Image));
    }
    else
    {
        lesContacts.Add(new Professionnel(txtNom.Text, txtTel.Text, pctPhoto.Image));
    }
}
```

5. Modification d'un contact

Lorsque l'on clique sur le petit crayon pour modifier un contact :



Toutes les informations, photo incluse, du contact s'affiche dans la zone de saisie.

Pour cette fonctionnalité, j'ai procédé ainsi : création d'un contact provisoire, de type Particulier ou Professionnel selon le type du contact que l'on souhaite modifier, transfert des informations du contact à modifier " dans " le contact provisoire, puis suppression du contact dans la liste, et chargement des informations du contact provisoire dans les zones de saisies correspondantes.

Ensuite, une fois qu'on a modifié le contact on clique sur le bouton +, c'est donc la procédure de l'ajout d'un contact qui se déclenche.

Un exemple : la fonction contactTransitoire qui retourne un nouveau contact tout neuf, en récupérant le type à partir du contact sélectionné :

```
// Création d'un objet transitoire unContact pour la modification
1 référence
private Contact contactTransitoire(int index)
{
    Contact unContact;

    if (lesContacts[index] is Particulier)
    {
        unContact = new Particulier(lesContacts[index].getNom(), ((Particulier)lesContacts[index]).getPrenom(), lesContacts[index].
    }
    else
    {
        unContact = new Professionnel(lesContacts[index].getNom(),lesContacts[index].getTel(),lesContacts[index].getPhoto());
    }
    return unContact;
}
```

6. Suppression d'un contact

Fonctionnalité très simple, et qui s'obtient de manière aussi simple. La procédure supprimerContact()

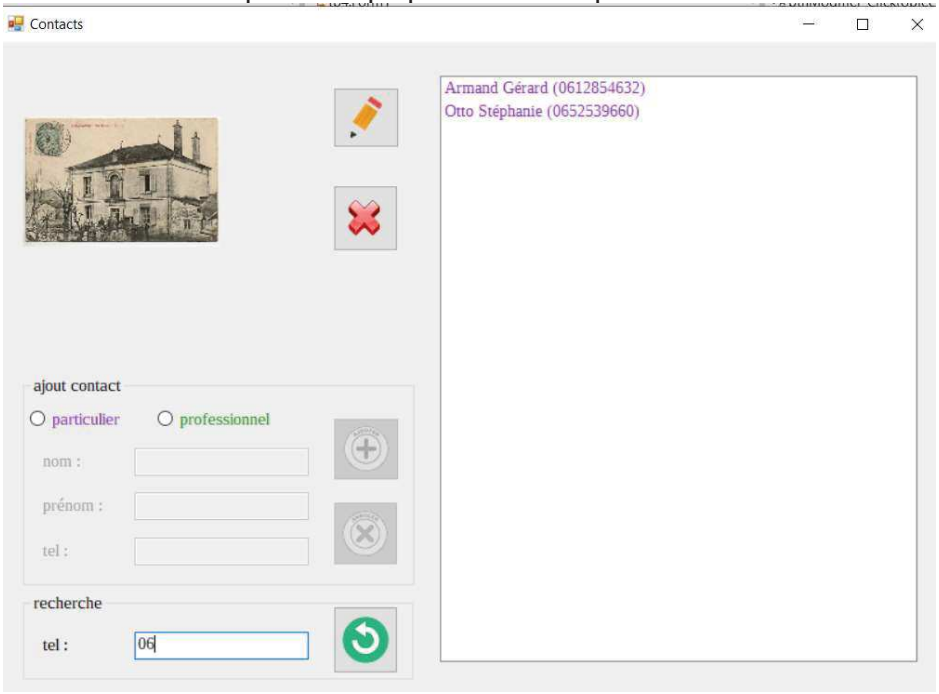
```
// Pour la suppression d'un contact :
2 références
private void supprimerContact()
{
    lesContacts.RemoveAt(indListboxToLesContacts(lstContact.SelectedIndex));
    majLstContact(-1);
}
```

L'évènement qui l'utilise :

```
private void btnSupprimer_Click(object sender, EventArgs e)
{
    int indice = lstContact.SelectedIndex;
    if (indice >= 0)
    {
        supprimerContact();
        majLstContact(indice);
    }
}
```

7. Recherche par n° de téléphone

Quand on saisit dans la zone de recherche, la liste des contacts se “ rétrécit ” au fur et à mesure : à chaque caractère tapé, il ne reste dans la liste que ceux qui peuvent correspondre à la saisie.



La méthode associée vide en réalité la liste des contacts, et la mets à jour à partir de la saisie. Comme c’est instantané, on a l’illusion, à l’utilisation, que seuls ceux qui ne correpondent pas disparaissent. Alors que c’est l’inverse : seuls ceux qui correspondant apparaissent, avec une mise à jour de la liste à chaque caractère saisie.
La méthode que j’ai associé est celle-ci :

```
// Recherche d'un contact à partir de son n° de téléphone
1 référence
private void rechercheContact()
{
    lstContact.Items.Clear();

    for (int i = 0; i < lesContacts.Count; i++)
    {
        if (lesContacts[i].getTel().Contains(txtRecherche.Text))
        {
            lstContact.Items.Add(lesContacts[i].infosContact());
        }
    }
}

/*
```

8. Mise en couleurs

Dans la liste, les contacts ne sont donc pas affichés de la même couleur : les contact Professionnels sont en vert, et les contacts Particulier sont en violet.

J’ai bien documenté mon code, donc inutile de rajouter des informations redondantes :

```
private void lstContact_DrawItem(object sender, DrawItemEventArgs e)
{
    // Dessine le fond de la listBox pour chaque Item
    e.DrawBackground();

    // Défini une brosse avec une couleur par défaut :
    Brush monPinceau = Brushes.Black;

    if (e.Index != -1)
    {
        if (lesContacts[indListboxToLesContacts(e.Index)] is Particulier)
        {
            monPinceau = Brushes.BlueViolet;
        }
        else
        {
            monPinceau = Brushes.ForestGreen;
        }
        // Dessine le texte de l'Item courant sur le fond avec le pinceau paramétré
        e.Graphics.DrawString(lstContact.Items[e.Index].ToString(), e.Font, monPinceau, e.Bounds, StringFormat.GenericDefault);

        // si la listBox a le focus, dessine un focus rectangle sur l'Item sélectionné :
        e.DrawFocusRectangle();
    }
```