Création d'une application à l'aide de Visual Studio et de WPF

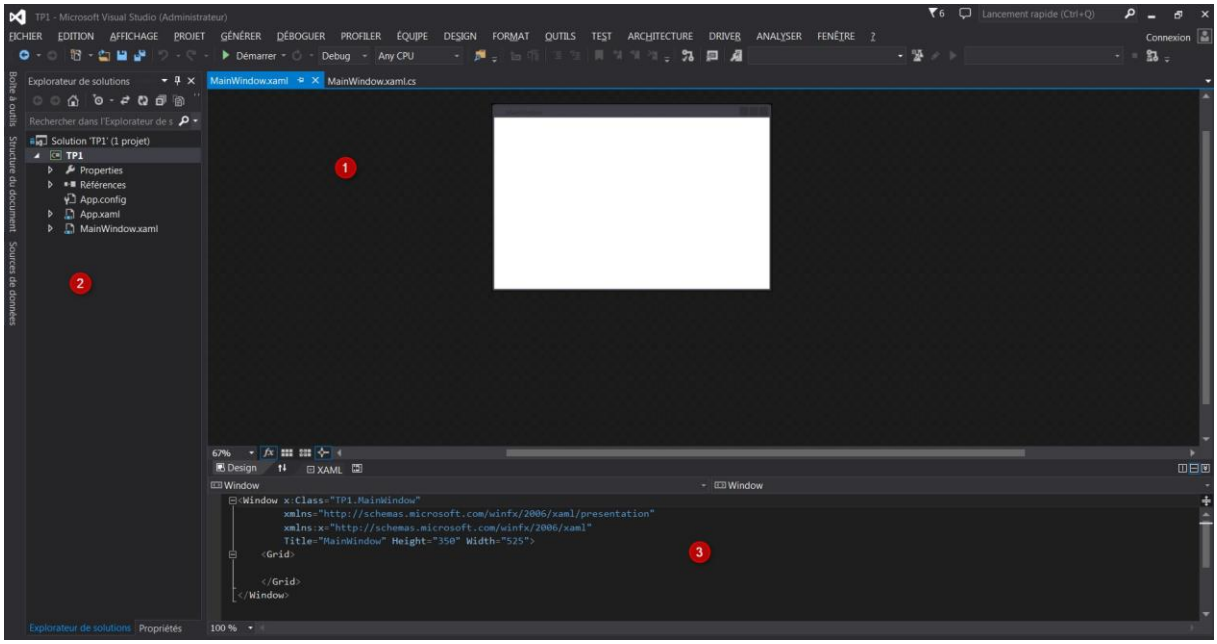
1. Création d'un nouveau projet

Pour créer un nouveau projet, lancez Visual Studio, puis sélectionnez **Fichier / Nouveau / Projet**.

Sélectionnez ensuite **Application WPF** dans **Visual C#**.

Donnez un nom à votre projet, et notez l'emplacement qui contiendra les sources du projet ainsi que les binaires générés.

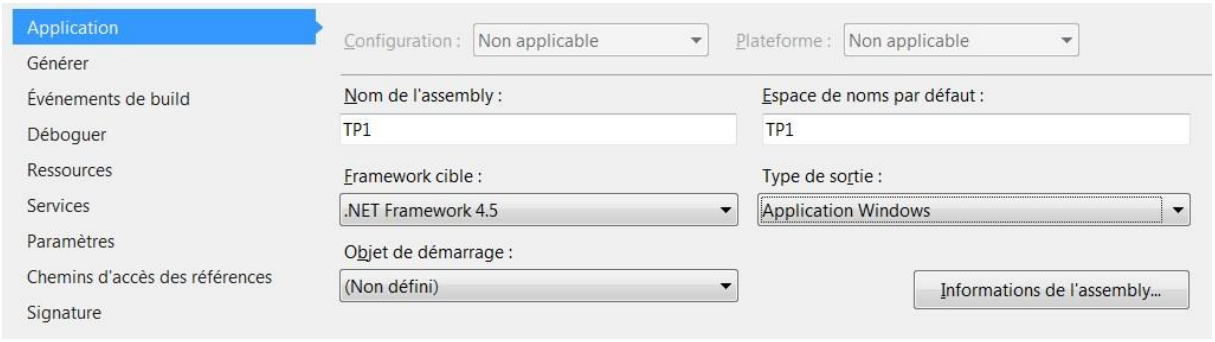
Après avoir validé, observez l'écran et identifiez les différentes zones de votre projet : le Designer, le xaml associé au designer, l'arborescence des différents éléments de votre projet.



Parcourez également le menu pour voir tout ce que vous avez à disposition. Notamment :

- **Affichage**, pour voir toutes les fenêtres à votre disposition. **Liste d'erreurs** contient les erreurs générées par la compilation. **Sortie** affiche des messages d'état relatifs à diverses fonctionnalités de l'environnement de développement intégré (IDE)
- **Projet**, pour ajouter des éléments à votre projet (nouveau fichier de code, nouvelle fenêtre…). Pour voir les propriétés de celui-ci également, nous y reviendrons plus tard.
- **Générer**, avec construire vos binaires. Quelle est la différence entre **Générer**, **Regénérer** ? Quelle est l'utilité de **Nettoyer** ?

Revenez aux **propriétés** du projet :



Dans **Application**, nous pouvons définir le nom de l'**assembly** et du **namespace** par défaut.

L'assembly est caractérisée par :

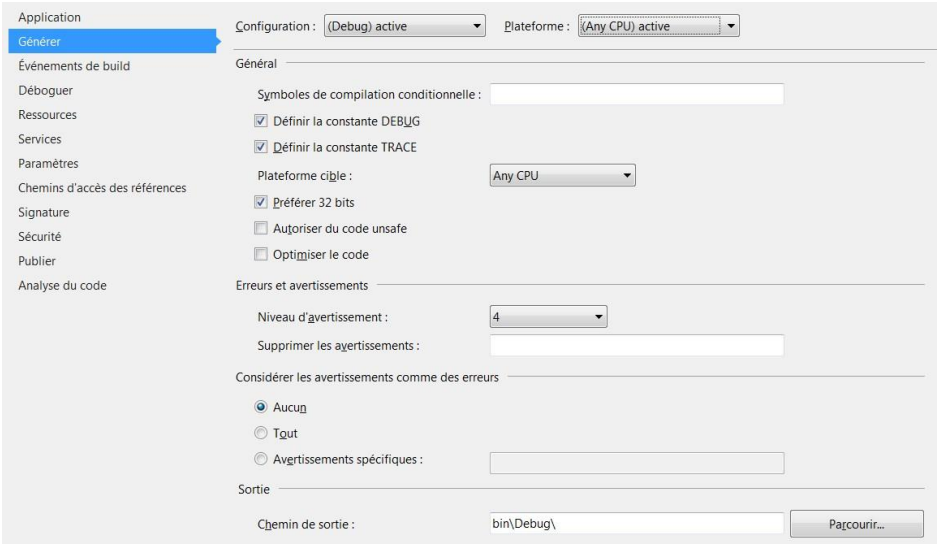
- Son nom
- Un numéro de version
- Un identificateur de culture, qui permet par exemple de définir le séparateur décimal pour les nombres
- Une clé publique, qui permet aux programmes utilisant cette assembly de vérifier qu'il s'agit de celle qu'ils ont référencée

Le namespace permet de regrouper un ensemble de classes dans un « groupement ».

Nous pouvons également choisir le Framework cible. Une application construite avec la Framework 4.5 ne pourra tourner que sur une machine où est installé ce framework. Ce n'est pas comme en C++ ou un éxécutable peut inclure les dlls dont il a besoin. En .Net, un programme ira chercher les assemblys nécessaires dans le GAC, qui est noatmment peuplés par les installations du Framwork .Net

Et le type de sortie permet de choisir si notre application sera pourvue au non d'une IHM, s'il s'agit d'un .exe ou d'une .dll

Dans **Générer**, nous pouvons choisir la configuration et la plateforme cible :



Quelles sont les différences entre Debug et Release ? Quel mode choisir lorsque l'on code ? Et quand on doit livrer une application à un client ?

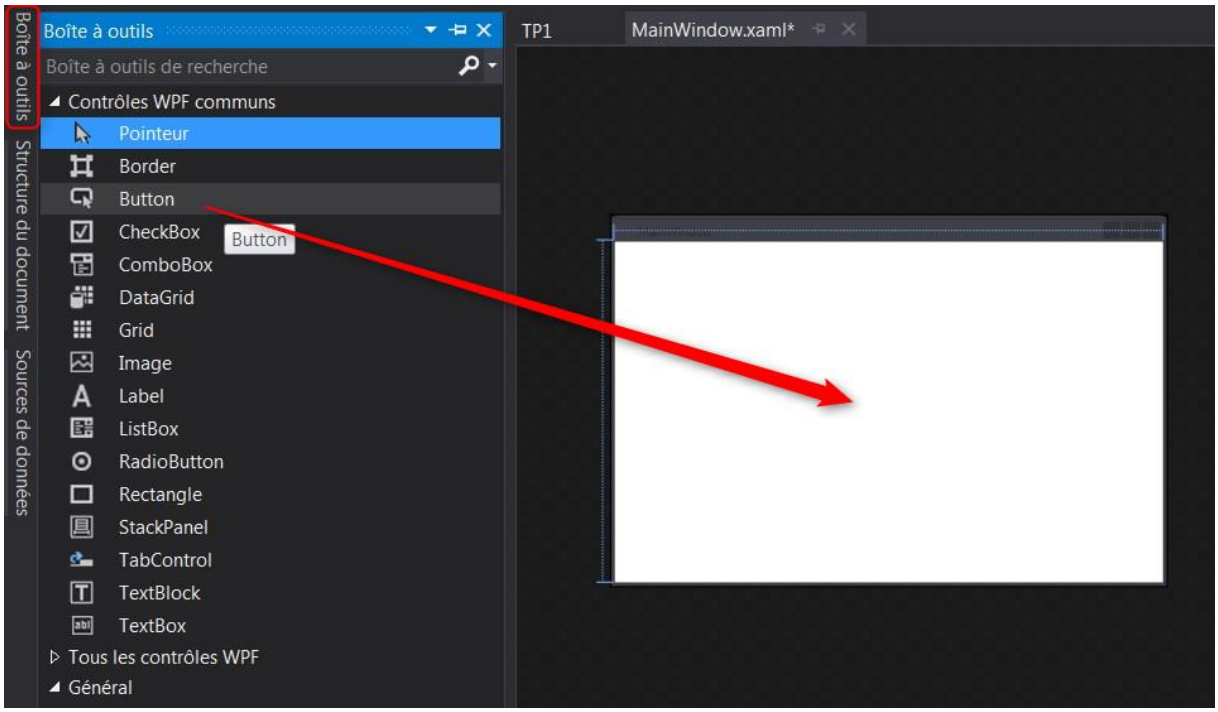
Qu'est-il préférable de choisir comme platforme cible ? x86, x64 ou anyCPU ?

Une fois vérifiées les propriétés du projet, revenez au code. **Générez** votre projet, vérifiez **la fenêtre de sortie**, puis démarrez-le en **debug**.

Stoppez l'exécution. Vous revenez à votre code.

2. Ajout d'éléments graphiques

A l'aide du **designer**, ajoutez un élément **Button** sur votre fenêtre :



Vérifiez la fenêtre du code xaml juste en dessous, que voyez-vous ?

Allez maintenant dans les **propriétés** du bouton (raccourci : F4). Naviguez parmi les propriétés pour voir ce qui est disponible. Vous pouvez tester des modifications (couleur, taille, marges, alignement, etc…) et observer l'effet sur le xaml.

Mettez la propriété **Width** du bouton sur **auto**. Qu'observez-vous ? Changer le texte qui apparait sur le bouton. Qu'observez-vous ?

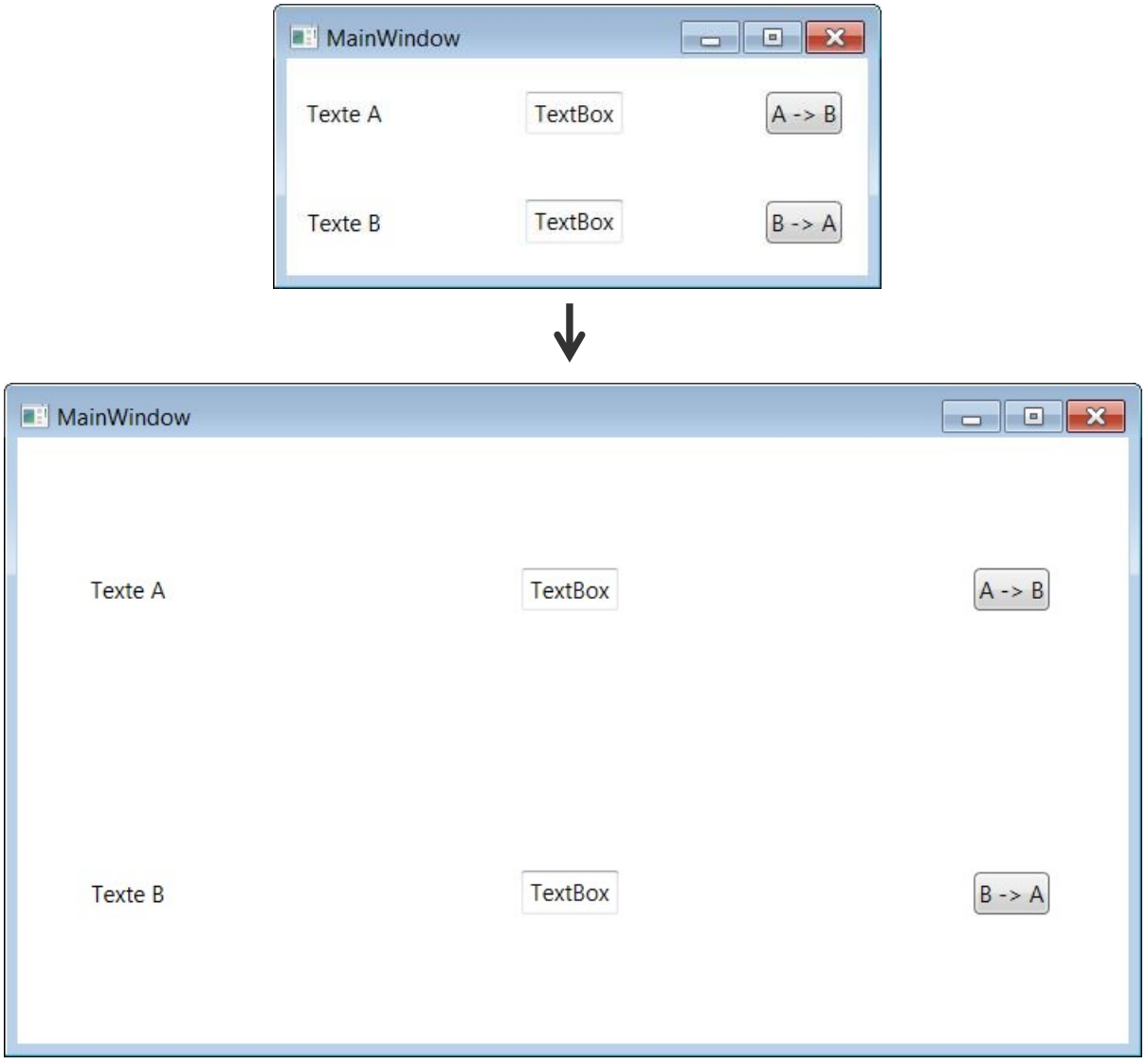
Tentez maintenant d'ajouter un nouvel bouton sur votre fenêtre. Que se passe-t-il ?

Pour découper proprement notre fenêtre, supprimez le bouton, puis ajoutez un élément **Grid.** Faites en sorte qu'elle prenne tout l'espace disponible dans la fenêtre. Partagez là maintenant en 2 lignes et 3 colonnes.

Ajoutez maintenant dans cette grille les éléments suivants :

- 2 **TextBlock** (qui seront sur la 1ère colonne)
- 2 **TextBox** (qui seront sur la 2ème colonne)
- 2 **Button** (qui seront sur la 3ème colonne)

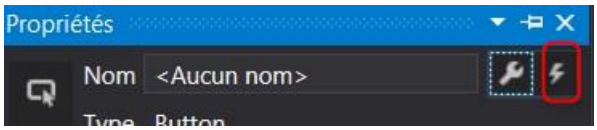
En jouant sur les propriétés d'alignement et de Width, les éléments doivent rester disposés de la même manière les uns par rapport aux autres si vous redimensionnez la fenêtre :



Compilez et testez.

3. Gestion des évènements

Vous allez maintenant intercepter les clics effectués sur les boutons. Pour ceci, retournez dans les propriétés du bouton A -> B, vous verrez un petit éclair en haut à droite de la fenêtre :



Cliquez dessus, vous avez accès à tous les évènements disponibles pour cet élément.

Donnez un nom de fonction pour l'évènement **Click** : ButtonABClick. Visual créé alors la fonction, qu'il va falloir remplir.

Remplissez la fonction avec la ligne de code suivante :

```
MessageBox.Show("You just clicked Button A -> B");
```

Compilez, lancez, et cliquez sur le bouton A -> B pour voir ce qu'il se passe. Stoppez le programme.

Retournez dans le code de la fonction **ButtonABClick**, passez votre souris sur **MessageBox** et sur **Show**. Vous verrez des *tooltips* apparaitre donnant des informations sur la classe et sur la fonction.

Faites maintenant *F12* (Go to Definition) sur **MessageBox**. Visual vous ouvre un fichier sur lequel vous pouvez voir :

- Le nom de l'*assembly* qui contient cette classe. Comment se nomme cette *assembly* ? Vous pouvez également voir son chemin sur le disque. Vous pourrez vérifier que cette *assembly* se trouve bien dans les **Références** de votre projet (dans l'explorateur de solutions). Sans cela, vous ne pourriez pas utiliser cette classe.
- Le nom du *namespace* qui contient cette classe. Quel est-il ? Si vous retournez dans votre fichier de code **MainWindow.xaml.cs**, vous pouvez voir ce nom de *namespace* utilisé avec un **using** en haut du fichier. Cela évite d'avoir à écrire :
  ```
  System.Windows.MessageBox.Show
  ```
  Le *using <namespace>* dispense d'avoir à écrire le nom du *namespace* devant la classe.
- Les méthodes disponibles pour cette classe. En vous aidant des *tooltips* et du F12, réécrivez le code **MessageBox.Show** en choisissant une des autres surcharges de Show pour que la fenêtre ressemble à ceci (*caption* Attention, icône de *warning*, bouton *OK* en bas à droite)

Compilez, testez.

Réalisez la même opération (événements, méthode à coder) pour le bouton B-> A.

Nous allons maintenant avoir besoin d'identifier chaque **TextBox** pour y avoir accès dans le *code behind* (fichier .xaml.cs)

Attribuez l'identifiant *TextBoxA* à la TextBox de la 1ère ligne, *TextBoxB* à celle de la seconde lligne, en passant par le xaml. Quelle est l'attribut que vous avez utilisé ?

Remplacez le code **ButtonABClick** par celui-ci :

```
        private void ButtonABClick(object sender, RoutedEventArgs e)
        {
            TextBoxB.Text = TextBoxA.Text;
        }
```

Faites de même pour **ButtonBAClick**, en adaptant bien évidemment.

Compilez, et testez. Expliquez pourquoi on a accès aux attributs **TextBox** dans notre code *C#*, alors qu'ils sont définis dans le xaml. Expliquez ce que fait le code des méthodes appelées par les clics.

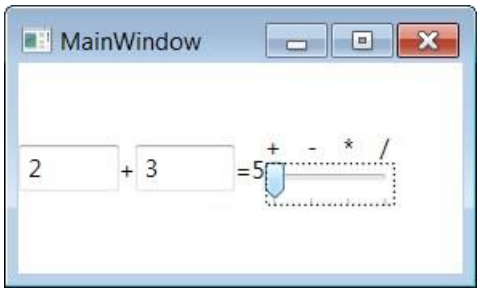
4. DataBinding, Converter, DataTemplate, Style, DataTrigger

Plutôt que de passer par les clics de boutons pour synchroniser le texte entre les 2 **TextBox**, on veut le faire automatiquement, en temps réel. Ecrivez le code nécessaire, en passant uniquement par le *xaml*.

Compilez, et testez.

Ajouter maintenant un nouveau projet dans votre solution, que vous nommerez **Calculator**.

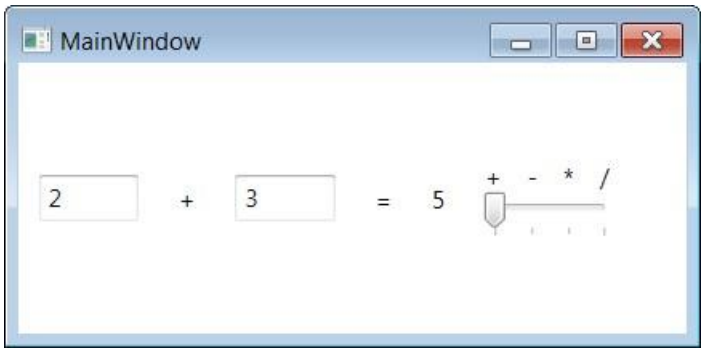
L'écran aura l'apparence suivante :



On pourra réaliser les 4 opérations arithmétiques de base sur 2 entiers, l'application nous donnera le résultat.

C'est un enchainement horizontal de contrôles graphiques. Quel élément allez-vous utiliser pour les contenir ?

Les contrôles sont trop rapprochés les uns des autres. Vous définirez un **Style** pour mettre des marges qui seront utilisées par tous les contrôles. Le **Style** doit être centralisé. On souhaite que l'application ressemble à ceci :

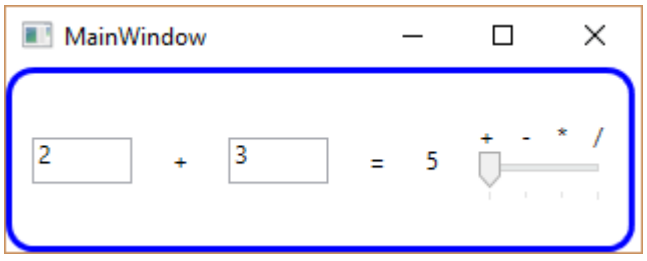


Le calcul devra se faire en temps réel, aucune validation de la part de l'utilisateur ne sera nécessaire. On veillera à utiliser le **DataBinding** et un/des **Converter** si nécessaire

Pour cela vous aurez besoin de :

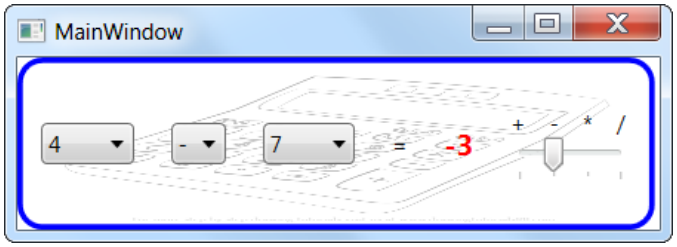
- Coder une classe **Operation** qui *correspondra* à notre modèle de données. Concrètement, elle devra donc contenir 2 valeurs de type **int**, un opérateur de type **string**, et le résultat de type **int**.
- Instancier un objet typé **Operation**. Vous pouvez le faire directement dans le *xaml*, dans les ressources du contrôle principal (l'instance sera accessible dans tous les nœuds fils)
- *Binder* les 2 entrées (correspondant aux 2 **TextBox** sur l'IHM) et l'opération choisie (**Slider** sur l'IHM) avec les attributs correspondant dans notre instance d'**Operation**. Pour le *slider*, vous aurez besoin de *convertir* sa valeur qui est de type int en un string qui correspondra au type de l'opération
- Déclencher le calcul lorsque l'utilisateur entre un nouveau nombre ou choisit une nouvelle opération.
- *Notifier* l'IHM lorsque le résultat est mis à jour dans notre instance.

Si cela n'est pas déjà fait, passez par un **DataTemplate** pour afficher votre instance d'**Operation**, avec un résultat qui doit ressembler à ça :



Créer un autre **DataTemplate** pour changer l'apparence de votre calculette. Par exemple, remplacer les **TextBox** par des **ComboBox** avec valeurs prédéfinies, etc... Ajoutez également une image de fond à votre application. Vous êtes libres, mais il faut au maximum éviter de changer autre chose que le **DataTemplate** pour changer votre écran.

Faites maintenant en sorte que le résultat s'affiche en rouge s'il est égal à 0. Puis s'il est inférieur à 0. On utilisera pour cela un/des **DataTrigger**.



Dans cet exemple, nous avons vu qu'il est aisé de changer l'apparence de notre application en ne modifiant que la partie graphique (et a fortiori le DataTemplate).

Nous avons bien séparé la couche *View* (IHM) du *Model* (classes représentant les objets métier que l'on manipule, ici l'opération.

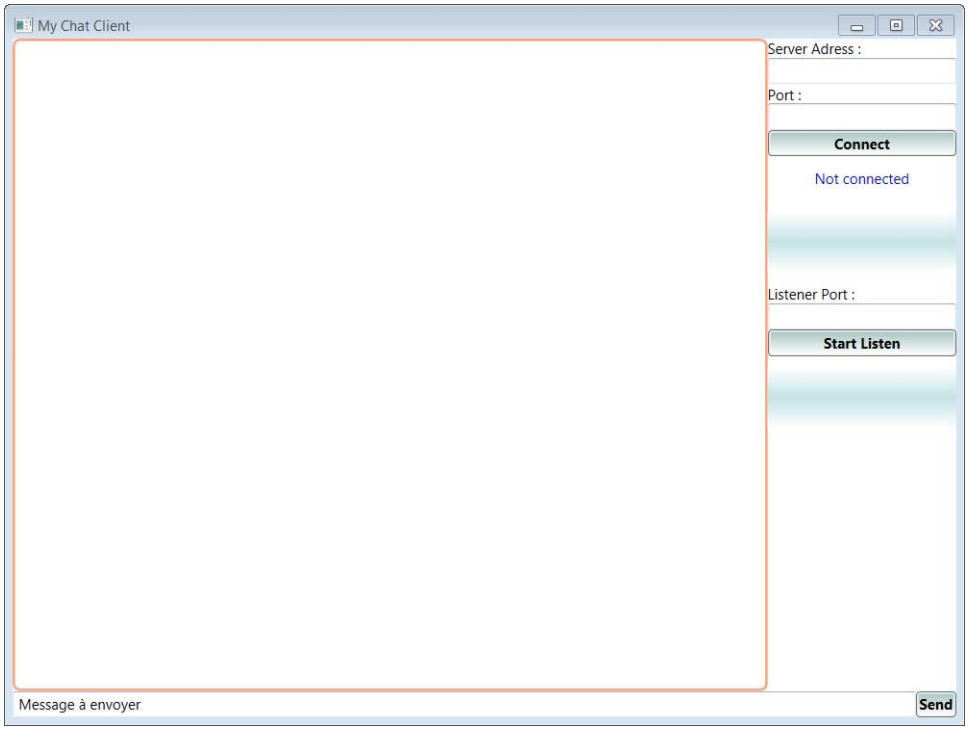
Pour les applications complexes, il faut aller encore plus loin, et adopter l'architecture **MVVM** (Model View View-Model). Cela signifie que nous aurons toujours notre *Model* et notre *View* bien séparés, mais qu'en plus, le Model ne doit plus être instancié directement dans la View comme nous venons de le faire, mais dans une classe *View-Model* servant à faire la liaison entre la partie affichage et la partie modèle de données. Nous allons mettre ce concept en application dans le projet suivant.

5. Programmation d'un Chat type IRC

1. Partie Client

Nous allons maintenant coder un Chat assez basique : nous allons commencer par coder la partie client, qui va se connecter à un serveur (adresse et port, définis par l'utilisateur) et lui envoyer des messages avec le protocole UDP. Le client pourra se mettre en écoute sur un port lui aussi défini par l'utilisateur.

Apparence du chat :



Première étape :

Créer le GUI. Il sera défini dans deux fichiers nommé ChatClientView.xaml et ChatClientView.xaml.cs

Passez par une **Grid** pour placer vos éléments, par un **StackPanel** pour les éléments situés dans le bandeau à droite.

Vous donnerez un style à vos boutons, comme sur le screen: dégradé de couleur (**LinearGradientBrush**), texte en gras, etc... Le style étant partagé par les boutons, vous créerez un dossier **Resources** dans votre projet, et vous y ajouterez le fichier définissant le style des boutons : ChatClientButtonStyle.xaml. L'élément racine de ce fichier doit être un **ResourceDictionary** car on pourra y ajouter plusieurs éléments si besoin.

Dans les ressources de votre fenêtre principale, vous importerez ce ou ces styles en fusionnant le **ResourceDictionary** de celle-ci avec celui importé, on utilisera donc la balise **ResourceDictionary.MergedDictionaries**.

Seconde étape :

Créer les modèles. Dans une architecture MVVM, nous voulons séparer la vue du modèle, avec qu'un changement de code sur l'un n'impacte pas le code de l'autre.

Nous allons donc créer deux classes : une **Connection** qui va représenter toute la partie réseau, et une **Message** qui va représenter les messages envoyés et reçus. Chaque classe sera définie dans son propre fichier.

La classe **Message** sera pour le moment très simple. Elle va posséder 3 attributs :

- *Sender*, qui sera l'IP de celui qui a envoyé le message reçu
- *TimeStamp*, qui sera l'horodatage du message reçu
- *Content*, le contenu du message proprement dit.

La classe **Connection** va elle contenir :

- *ServerAdress*
- *ServerPort*
- *ListenerPort*
- *Server*, de type **UdpClient**
- *Listener*, de type **UdpClient**
- Des méthodes permettant de : se connecter au serveur, de déclencher une boucle d'écoute, d'envoyer un message au serveur, de recevoir un message.

Troisième étape :

Nous allons ensuite créer une classe **ChatClientViewModel**. Cette classe va assurer la liaison entre le GUI et nos modèles. Elle sera donc instanciée une fois au démarrage de l'application, sera définie comme *DataContext* de notre fenêtre principale, et c'est elle qui instanciera les modèles et les manipulera.

Cela implique que lors d'un événement déclenché par l'utilisateur, la fonction appelée dans la fenêtre appellera elle-même une fonction du *View-Model* qui gèrera cet évènement en manipulant le ou les *modèles* nécessaires.

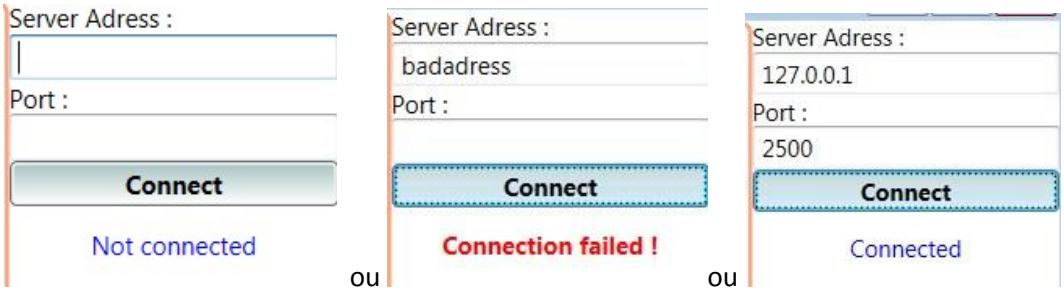
Ce *View-Model* devra donc instancier la **Connection** nécessaire et les **Message** à chaque réception pour les afficher dans le GUI.

Quatrième étape :

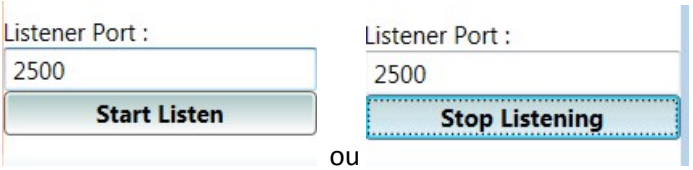
Il faut maintenant coller tous les éléments ensemble pour que ça fonctionne !

On commencera par ajouter des attributs à notre View-Model :

- **ConnectionStatus**, qui nous renseignera sur le statut de la connexion, à l'aide d'un enum : Pas connecté, connecté ou erreur de connexion. Avec un DataBinding, un Converter et un DataTrigger, on le reliera à notre GUI à un TextBlock qui nous indiquera le statut de la connexion comme ceci :



- **ListeningStatus**, qui sera relié au bouton idoine dans le GUI avec DataBinding et Converter :



- **TextDisplayed** qui servira à afficher les messages entrants dans la fenêtre de chat.
- `MessageTyped` qui sera bindé au message entré par l'utilisateur
- `connection`, de type Connection

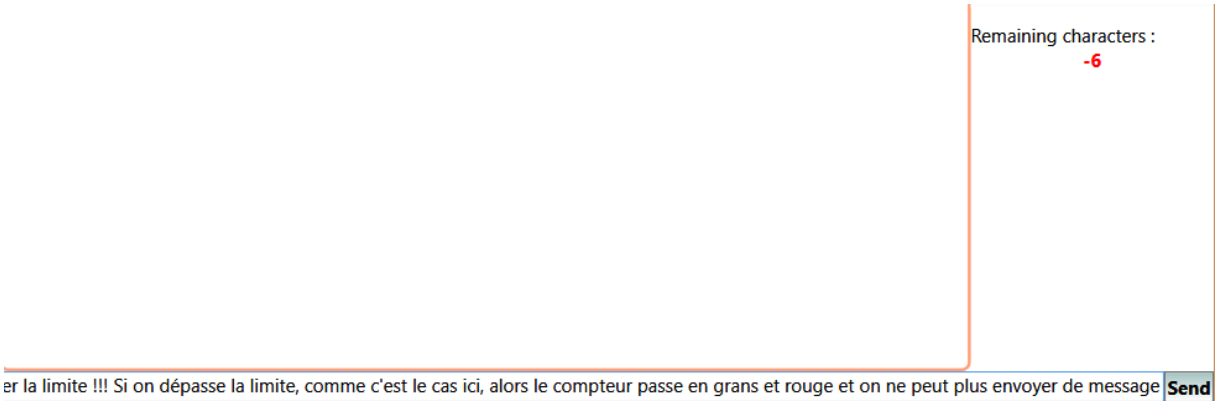
De quelle interface doit hériter votre *View-Model* pour avertir l'IHM en cas de changement d'un de ses attributs ? Et que faut-il implémenter pour rendre cela fonctionnel ?

Lorsque l'utilisateur clique sur Listen, il faut que cela appelle une boucle d'écoute de messages. Implémentez-là de la manière la plus simple, sans thread ou autre. Lancez votre application, mettez-vous en écoute. Que remarquez-vous ?

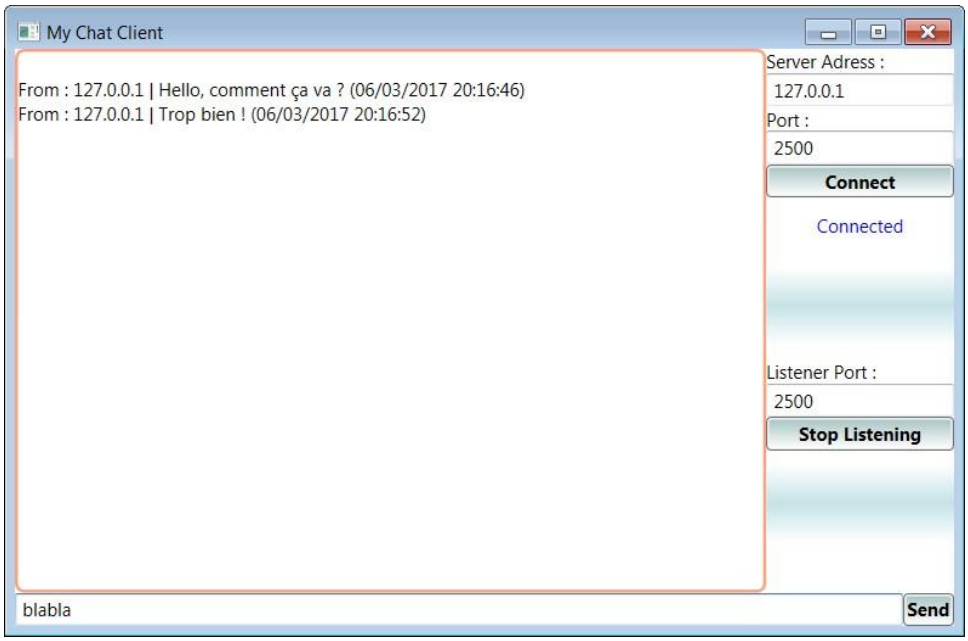
Pour pallier le problème observé, on pourrait lancer le processus d'écoute dans un nouveau thread. Mais le FrameWork .Net 4.5 permet d'utiliser les mots clés **async** / **await** qui vont nous éviter ce travail parfois fastidieux.

On veillera à gérer correctement les *exceptions*, notamment les problèmes de connexion, qui doivent être notifiés à l'utilisateur par l'affichage d'une **MessageBox** explicitant le problème.

Les messages devront faire 160 caractères maximum, sinon l'envoi sera bloqué. On implémentera un compteur de caractères « à la twitter » : il nous indiquera en temps réel combien de caractères on peut encore écrire, et sera mis en rouge gras si on a dépassé la limite :



On utilisera pour cela, DataBinding, Converter et DataTrigger. Rien d'autre.

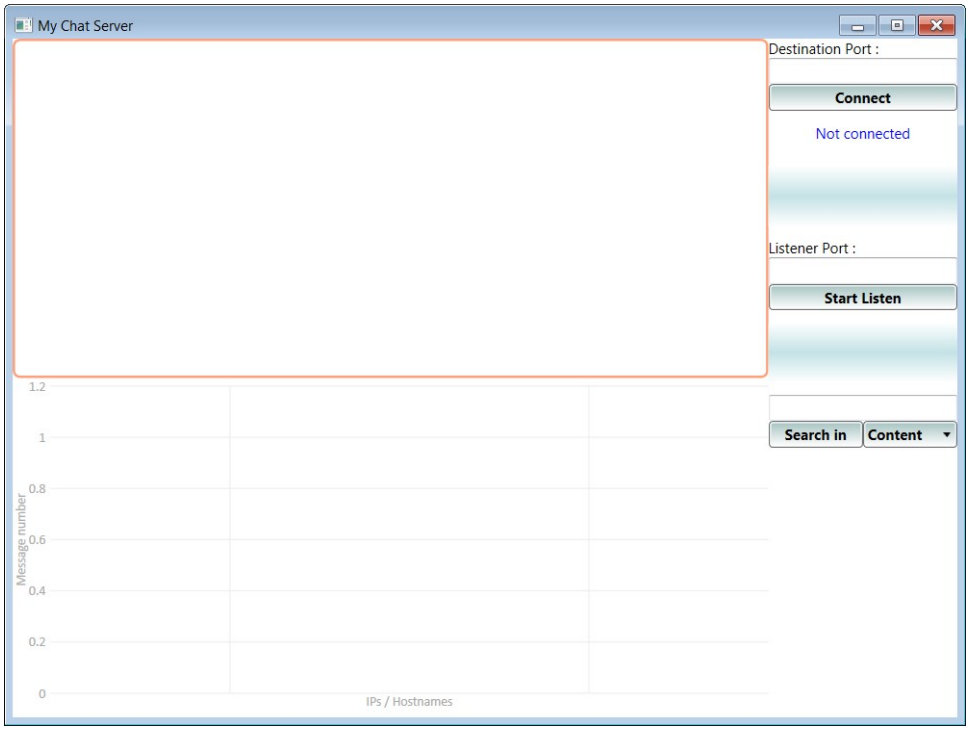


2. Partie Serveur

On va maintenant coder la partie Serveur, sur laquelle les clients pourront envoyer leurs messages et duquel ils pourront recevoir les messages des autres clients.

Le Serveur va donc avoir une architecture similaire au client : il va écouter sur le port d'envoi des clients, et renvoyer les messages en *Broadcast* sur le port d'écoute des clients.

Voilà son apparence :



Il reprend des éléments du client. En revanche, on n'a plus la zone de saisie, ni d'adresse de destination puisque l'on envoie en broadcast. On conserve tout le reste.

Le Serveur va donc afficher dans la zone principale les messages reçus, et les renvoyer vers les clients.

- Première modification à effectuer sur la classe `Connection` par rapport à celle du client : on passera à la méthode `Connect` de notre sender (de type `UdpClient)` comme premier argument l'adresse de diffusion broadcast. On utilisera pour cela le champ adéquat de la classe `IPAddress` .

- Ensuite, notre serveur va recevoir une trame UDP depuis un client, celle-ci étant modélisée par un objet de type `UdpReceiveResult` . Le champ `Buffer` va permettre de récupérer le contenu du message, le champ `RemoteEndPoint` des informations sur celui qui a envoyé le message, notamment son adresse IP. On va construire un objet de type `Message` à partir de ces informations pour les afficher sur la fenêtre principale, par l'intermédiaire d'un `ItemsControl` dont on settera l'`ItemsSource` à une `ObservableCollection<Message>` qui sera donc un attribut du ViewModel. L'`ObservableCollection` est une liste qui permet de notifier le GUI en cas d'ajout ou de suppression d'éléments. Mais attention, aucune notification n'est envoyée en cas de modification d'un élement existant.

- En revanche, côté client, l'`UdpReceiveResult` aura comme `RemoteEndPoint` les informations envoyés par le serveur. Hors nous ne voulons pas afficher l'IP du serveur, mais l'IP du client qui a envoyé le message. Pour cela, le serveur va sérialiser un objet Message contenant notamment l'IP du client, l'envoyer comme contenu de la trame UDP, et le client devra donc désérialiser le champ `Buffer` de l'objet de type `UdpReceiveResult` pour l'afficher dans la fenêtre de chat.

- Pour cela, on va ajouter à la classe `Message` un constructeur sans paramètre, condition nécessaire à une classe sérialisable. On va ensuite coder une méthode `string Serialize()`, qui va permettre de sérialiser l'objet courant. On utilisera pour cela un `XmlSerializer` et un `StringWriter.` Et on codera une méthode `public static Message DeSerialize(UdpReceiveResult udpFrame)` qui permettra de générer un objet de type Message à partir de la trame UDP reçue. On utilisera ici le même `XmlSerializer` et un `StringReader.` Le `XmlSerializer` pourra donc être déclaré comme attribut static de la classe comme ceci :
  ```
  private static XmlSerializer ser = new XmlSerializer(typeof(Message))
  ```
  En effet, il permet de sérialiser / désérialiser des objets de type `Message`, et donc ne dépend pas de l'instance mais de la classe.
  Vous mettrez un *breakpoint* dans votre code (raccourci F9) après l'appel à la méthode `Serialize` de votre `XmlSerializer` pour voir le contenu de votre `StringWriter.` Vous pouvez voir le contenu en faisant un clic droit sur votre objet, et en affichant « *espion express* »

Nous allons ensuite coder un petit module de recherche dans l'historique. Comme vous pouvez le voir sur le screenshot du serveur, on aura une `TextBox` pour le champ de recherche, une `ComboBox` permettant à l'utilisateur de chercher soit dans les IP/hostname , soit dans le contenu des messages, et un bouton permettant de déclencher la recherche.

- On commencera par coder une classe SearchAttribute très simple. Elle contiendra cet enum :

  ```
  public enum eSearchAttribute
  {
      eContent = 0,
      eSender
  }
  ```

  Et deux champs, `Index`, du type de cet enum, et `Display` de type `string`.
  Elle sera utilisée pour gérer la `ComboBox`

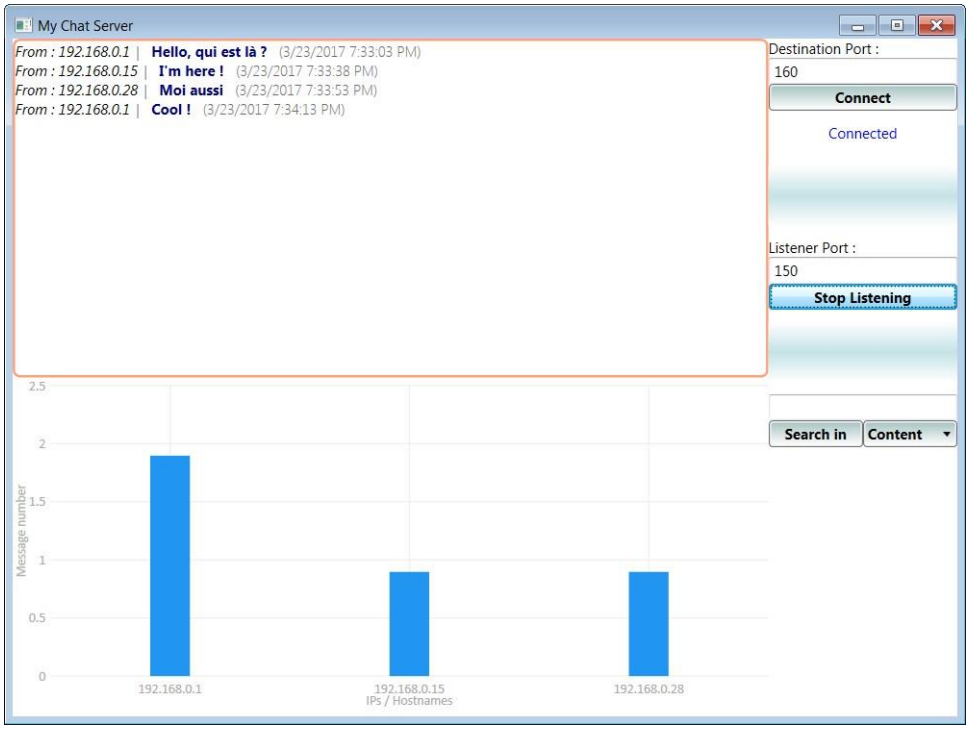
- On ajoutera un nouvel élément de type `Window` à notre projet, que l'on nommera SearchResult. Cette fenêtre sera appelée au clic sur le bouton *Search In* grâce à la méthode `ShowDialog()` et affichera les résultats de la recherche. Elle contiendra un `ItemsControl` dont on définira l' `ItemsSource` sur la liste qui contiendra les résultats de la recherche.

- On ajoutera ces 3 attributs à notre ViewModel :
  ```
  public ObservableCollection<Message> HistorySearchResult { get; private set; }
  public ICollectionView SearchAttributes { get; private set; }
  public SearchAttribute.eSearchAttribute SearchAttributeIndex { get; set; }
  ```
  Le premier contiendra une liste de `Message` vérifiant les conditions de la recherche, le second sera la liste des méthodes de recherche disponibles, à savoir par *Sender* / par *Contenu*, et servira donc à remplir la `ComboBox`, le dernier sera l'index de l'élément sélectionné dans la `ComboBox`. Sur la `ComboBox`, on utilisera à bon escient les attributs `ItemsSource`, `SelectedValue`, `SelectedValuePath` et `DisplayMemberPath`.

- Pour la recherche proprement dite, on pourra utiliser un sous ensemble du C# : le langage **LINQ**. Il permet de requêter « à la SQL » sur des collections, comme notre liste de `Message` par exemple. Par exemple, si `MessageHistory` contient la liste des Message arrivés sur le serveur, et `textToSearch` le texte à chercher entré par l'utilisateur, on peut écrire :
  `MessageHistory.Where(x => x.Sender == textToSearch)`
  X représente un élément de la liste, donc un `Message`, et on applique un filtre *Where* dessus permettant de ne garder que les `Message` dont le Sender correspond au texte entré par l'utilisateur. On peut ensuite convertir le résultat en une nouvelle liste.

Enfin, on finira par afficher un graphique temps réel permettant de voir combien de message sont arrivés par Sender. On utilisera pour cela le package *LiveCharts*.
Dans Visual, allez dans *Outils*, puis *dans Gestionnaire de package NuGet*. Sélectionnez *Gérer les packages NuGet pour la solution*. Dans la catégorie *En Ligne*, chercher « **LiveCharts** », sélectionner *LiveCharts.Wpf*, installez le pour le projet *Server*. Vous pouvez voir que deux *assemblies* ont été ajoutées dans les références. Elles contiennent les classes *LiveCharts* que l'on va pouvoir manipuler pour créer notre graphique.

Partagez l'écran principal en deux. Le haut contiendra les messages qui arrivent, le bas le graphique :



Le graph sera un élément `CartesianChart` que l'on ajoutera au xaml. Un bon exemple de ce qu'il faut faire est disponible ici :https://lvcharts.net/App/examples/v1/wpf/Basic%20Column
Il faut fortement s'en inspirer pour notre besoin. Par contre, il ne faudra pas utiliser des `int` comme valeurs du graph comme sur cet exemple, mais des `ObservableValue` car cette classe permet de notifier le GUI en cas de changement de valeur, ce qui permettra de rafraichir le graph au fur et à mesure de l'arrivée des messages.

Pour connaitre la posiiton d'un sender (représenté par son IP) sur l'axe des abscisses, on utilisera un attribut `private Dictionary<string, int> ipIndexGraph`, qui permettra pour une ip donnée (`string`) de connaitre son index (`int`) sur l'axe.