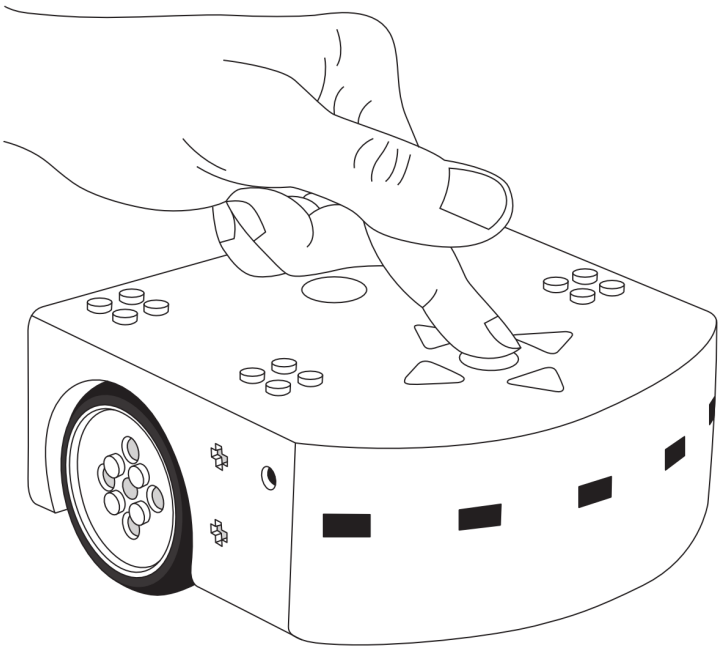
MES NOTES SUR LE PROJET THYMIO-II 2018

GUILLAUMIE Bilal



Projet numéro 15 bis
Semestre 3 : Période 2

I.U.T Informatique Clermont-Ferrand
Novembre 2018

1 Diagrammes

2 Thymio Version 2

3 WPF Application

3.1 Architecture MVVM

Durant la réalisation de notre projet j'ai choisi pour des raisons pratiques et pour ses multiples avantages de continuer notre projet via l'architecture MVVM.
À travers cette architecture assez complète et complexe nous pouvons distinguer trois éléments importants dans les initial M, V et VM

M : La couche Modèle qui contient les données.

V : La couche View qui correspond à ce qui est affiché. La vue contient les différents composants graphiques (boutons, liens, listes) ainsi que le texte.

VM : La couche ViewModèle ce composant fait le lien entre le modèle et la vue. Il s'occupe de gérer les liaisons de données.

Ainsi les valeurs de ma vue sont binder sur des propriétés de mon ViewModel qui ont accès à des données. « Pour rappel, le binding est un mécanisme qui permet de faire des liaisons entre des données de manière dynamique. » Le fait est que si l'on modifie la valeur d'une donnée. Alors celle-ci va notifier cette modification via le mécanisme implémenté par l'interface INotifyPropertyChanged et son gestionnaire d'événement propertyChangedEventHandler .

Possibilité d'implémenter une classe Mère implémentant le mécanisme de binding avec une méthode virtual OnPropertyChanged *Voir les exemples des cours de l'année dernière*

De plus de part cette architecture. Cela nous permet de pouvoir séparer les différentes partie et ainsi leur production sans devoir en permanence être conscient d'avec qui et quelle élément l'on travail.
Voici donc un schéma récapitulatif afin imager l'architecture MVVM.

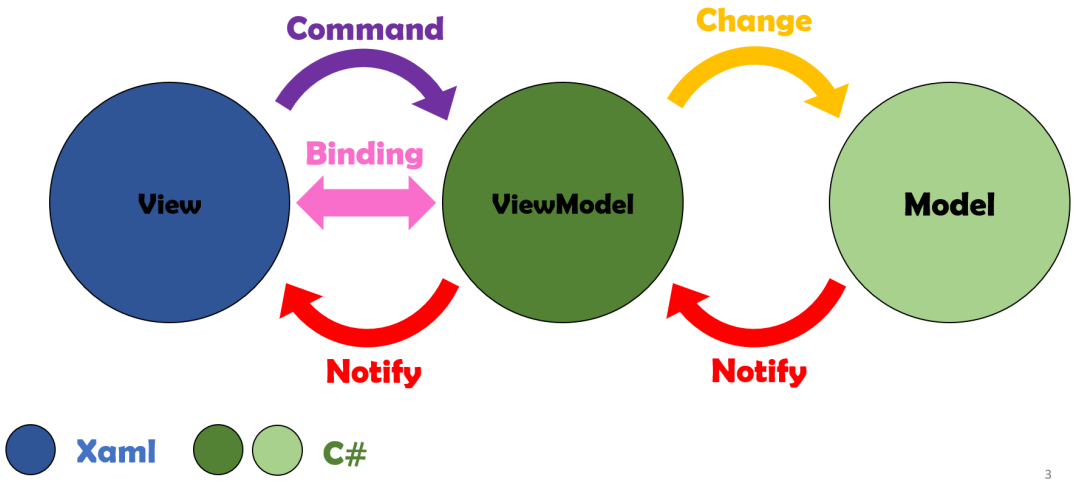


FIGURE 1 – Schéma récapitulatif d'un projet MVVM

Et voici un schéma reprenant les exigences et avantages de l'architecture MVVM

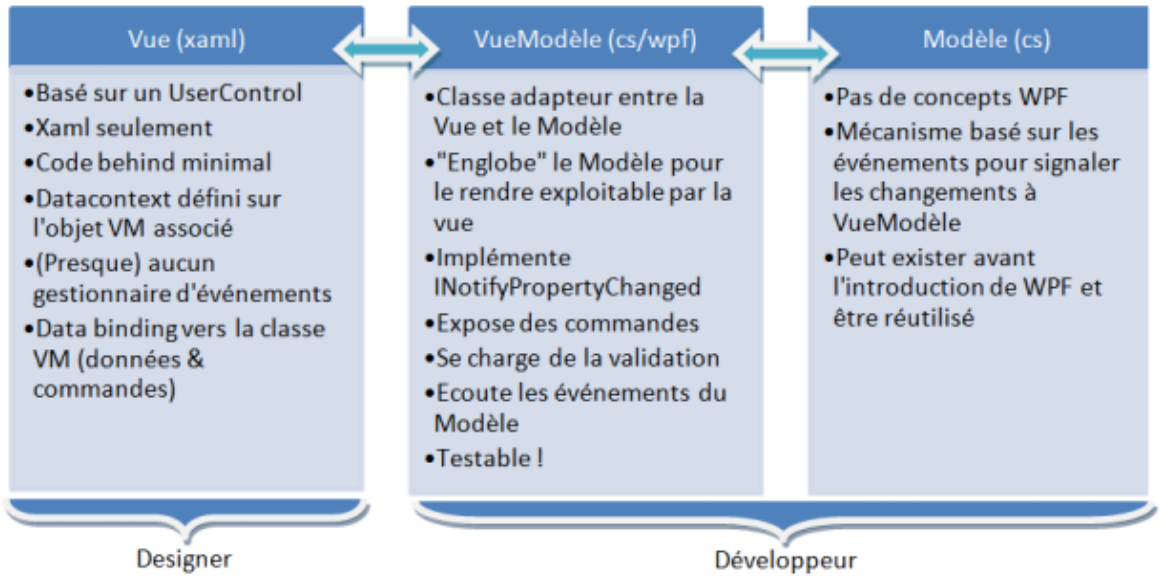


FIGURE 2 – La constitution de chaque partie de l'architecture

Par conséquent dans la conception d'un projet dont l'architecture choisie est MVVM. Chaque vues ne doit en aucun cas posséder de code behind. Ce qui violerait les principes même de séparer les différents éléments du projet, en raison du couplage fort entre le code behind et ca vue
Par conséquent il n'existe aucun événement dont la gestion se fait à travers le code behind de la vue. Pour cela nous utiliserons les command.

3.1.1 Command simple et Command générique

Pour palier au problèmes des événements voici donc le principe des **Commandes.**
"Les commandes ont plusieurs fonctions. La première est de séparer la sémantique et l'objet qui appelle une commande de la logique qui exécute la commande."
source : msdn.microsoft.com

De plus en mettant en place une structure mvvm puisque le principe d'une telle architecture et de séparer la vue des modèle via les viewmodel et qu'un des principes fondamental de cette architecture est de n'avoir aucun code behind alors le mécanisme de command s'impose.

Première méthode :

Cette méthode consiste à créer une class implémentant l'interface ICommand
Cette interface met en évidence ceci :

```
1 public event EventHandler CanExecuteChanged;
2
3 //Definis si la commande peut etre execute : true si oui false dans le
    cas contraire.
4 public bool CanExecute(object parameter){
5 }
```

```
6
7  //Cette methode definis ce que la commande doit executer. Lorsque la
       commande est appele cette methode est execute.
8  public void Execute(object parameter){
9  }
```

Voici donc l'implémentation de base d'une commande.
Par conséquent lorsque l'on souhaite créer notre propre commande. Alors il nous suffit de créer une Class NomDeMaCommand qui implémente l'interface ICommand et d'y mettre les caractéristiques spécifique i.e Dans la méthode Execute y mettre ce que l'on doit effectuer lors du déclenchement de la commande et dans la méthode CanExecute les conditions nécessaire pour avoir la commande active ou bien pour la bloquer.
Ensuite pour pouvoir lier une action de l'utilisateur à cette commande l'on va dans un premier temps créer un membre et une propriété de type ICommand :

```
1  private ICommand nomDeMaCommande;
2  public ICommand NomDeMaCommande
3  {
4  get { return nomDeMaCommande; }
5  }
```

Une fois fait il nous suffit d'instancier la commande dans le constructeur. Puis pour permettre de déclencher la commande l'on va devoir faire un binding entre un élément wpf comme un bouton par exemple sur la propriété définis plus tôt : NomDeMaCommande

Voici donc l'ensemble des élément nécessaire à l'implémentation d'une commande.
Seulement Problème :
Dans le cas où l'on possède un nombre important de commande l'on doit par conséquent implémenter un nombre important de class propre à chaque commande et par conséquent alourdir le projet et y consacrer un temps bien trop important.

C'est pourquoi l'on va donc implémenter une class plus générique nommé RelayCommand Cette class va nous permettre d'éviter de créer à chaque moment une nouvelle commande. Et pour ce faire l'on va utiliser le principe des déléguer. Et dans notre cas des délégués précis et déjà implémenter par Microsoft.

Deuxième méthode :

Relay Command

Le principe pour la class RelayCommand reste le même c'est-à-dire que cette class implémente l'interface ICommand Elle possède donc les méthodes Execute , CanExecute et le gestionnaire d'événement CanExecuteChanged
Pour mettre en place cette class générique pour les commande nous avons besoin de mettre en avant ce qu'est un délégué.

Les **délégués** sont des types de références qui prennent des méthodes en paramètre. Une fois que le délégué est appelé ou invoqué, les méthodes dîtes abonné sont appelées à leur tour.
Pour créer un déléguer rien de plus simple :

Etape 1 : On créer un délégué contenant le type de méthode qu'il va référencer

Etape 2 : On créer une instance du délégué

Etape 3 : Puis on abonne ou désabonne des méthode compatible avec le délégué

```
1  //Etape 1 :
2  //Ici notre delegue determine les methode qui peuvent etre reference par
       le delegue
3  // ici ce sont des methode retournant rien et prenant en parametre une
       chaine de caractere
4  public deleguate void MyDeleguate(String str);
5
6  //Etape 2 :
7  //L'on creer une instance de mon delegue.
8  public MyDeleguate del;
9
10 //Etape 3 :
11 //L'on abonne nos methode via un +=
12 //On les desabonne via un -=
13 //On l'abonne en suprimmant toute les reference des methode precedement
       abonne via le =
14 del +=  nomdeLamethode ;
15 del -=  nomdeLamethode ;
16 del =   nomdeLamethode ;
```

Voici donc comment créer un délégué. Pour pouvoir éxécuter les méthodes il suffit donc juste de faire un del.invoke();

L'on peut donc par la suite créer des **délégué générique.**
En voici des exemples :

```
1  //Ce delegue prend en parametre des types generique T est retourne un
       element du meme type i.e T
2  public deleguate T MyDelguate<T>(T arg)
3
4  //Ici meme cas que le precedent cependant le type de retour n'est pas
       forcement le meme que celui du parametre.
5  public deleguate TRes MyDelguate<T , TRes>(T arg)
```

Voici donc des délégué générique. Et c'est sur cela que va se baser la généricité de notre class RelayCommand.
Dans le principe nous allons dans notre class RelayCommand non plus implémenter le nom de la méthode que l'on va par la suite appelé mais un délégué auquel sera référencé

notre méthode à éxécuter. L'on pourrait passer par un délégué générique comme vu auparavant. Avec comme méthode référencé une méthode ne prenant en paramètre rien et renvoyant en paramètre void. Et faire de même pour les méthodes que l'on devrais exécuter dans la partie CanExecute de notre commande. Ici l'on devrait avoir une méthode renvoyant un boolean et ne prenant rien en paramètre du moins pour notre implémentation. Si nécessaire l'on pourrait rajouter des a paramètre générique T.

Seulement voila en C# des délégué de ce genre sont déjà implémenter.
Il s'agit des délégué **Func & Action**.
Func & Action sont des délégués déjà définis permettant d'éliminer la complexité de la création de délégué générique. Ces délégués peuvent prendre jusqu'à 16 paramètre génériques.

Action Prend jusqu'à 16 paramètre et retourne void.

```
publi delegate void Action<in T>(T arg );
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
...
```

Func Prend jusqu'à 16 paramètre et retourne une valeur de Type TResult

```
public delegate TResult Func<in T, out TResult>( T arg );
public delegate TResult Func<in T1, in T2, out TResult>( T1 arg1
    , T2 arg2);
...
```

Par conséquent l'on va par le biais du délégué **Func** "référencé les fonctions à exécuter dans la partie CanExecute. Ce délégué prendra en paramètre rien et retournera un boolean. **true :** si la commadne peut être exécuter, **false :** si elle ne peut pas être exécuter.

```
public class RelayCommand : ICommand
{
readonly Action _execute = null;
readonly Func<bool> _canExecute = null;

public RelayCommand(Action execute) : this(execute, null){
}

public RelayCommand(Action execute, Func<bool> canExecute)
{
if (execute == null)
throw new ArgumentNullException("Exception car execute == null");

_execute = execute;
_canExecute = canExecute;
}

public bool CanExecute(object parameter)
{
return _canExecute == null ? true : _canExecute();
}

public event EventHandler CanExecuteChanged
{
add { CommandManager.RequerySuggested += value; }
```

```
26 remove { CommandManager.RequerySuggested -= value; }
27 }
28
29 public void Execute(object parameter)
30 {
31 _execute();
32 }
33 }
```

Voici donc la composition de notre class RelayCommand
A noter que les délégué Action et Func sont en readonly pour ne pas permettre la modification des méthodes référencées.
L'on aurait aussi faire _execute.invoke(); au lieu d'appeler les méthodes par _execute();

4 ASP .NET (Razor) Application

5 Application on Raspbian