1 Windows Forms - de la liaison de données à la liaison d'objets

Partie 3 : des liaisons sur les évènements

Date de publication : 03/12/2010 , Date de mise à jour : 05/12/2010

Par Saint-Ouen Olivier

Dans cet article, nous allons étudier comment gérer des liaisons sur les évènements. Pour cela nous allons créer un moteur de liaison simple à utiliser, disposant d'une extensibilité contrôlée et non intrusif sur le code utilisateur.
1 commentaire · 

Version PDF (Miroir) Version hors-ligne (Miroir)

2 Notes de lecture

Sur MSDN, la définition d'un EventHandler est : *Représente la méthode qui gérera un évènement qui n'a aucune donnée d'évènement*.
Plus précisément, le délégué connecte un évènement avec son gestionnaire. On parle alors de gestionnaire d'évènement.

Supprimé : é

Supprimé : d'événement

Dans cet article, nous utiliserons les dénominations suivantes :

- **Récepteur d'évènement**, ou **Récepteur** : pour nommer la partie déclenchée lors de l'évènement ;
- lorsque nous nommons un objet du *Framework*, celui-ci est représenté sous la forme d'un lien sur sa documentation.
  Exemple : EventHandler.

3 II. Introduction

Commençons par l'existant : nous savons que le moteur de liaison des *Windows Forms* est capable de réagir aux modifications de valeur d'une propriété.
Nous pouvons aisément en déduire que ce moteur de liaison interagit avec les évènements déclenchés par ces propriétés.

Observons la classe TypeDescriptor ; on y trouve entre autres les méthodes CreateEvent et GetEvents.

Supprimé : entre autre

Toutes ces méthodes nous renvoient à l'objet EventDescriptor.
Dans cet objet, les méthodes qui attirent tout de suite notre attention sont AddEventHandler et RemoveEventHandler .

Elles forment le cœur de la gestion des évènements et nous permettent d'attacher ou de détacher des délégués de récepteur à un évènement.
Ce qui correspond exactement à notre besoin.

Ces méthodes demandent deux arguments :

- l'objet fournissant l'évènement ;
- et le délégué de récepteur à attacher ou détacher.

Nous avons donc la possibilité de réaliser un moteur de liaison des évènements qui utilisera le modèle objet standard.

Pour d'autres exemples d'utilisation du modèle objet des liaisons, je vous invite à consulter les articles précédents de cette série :
partie 1: présentation et amélioration de l'ergonomie des liaisons ;
partie 2: des liaisons sur les composants.

4 I-A. Les fonctionnalités désirées

Maintenant que nous savons qu'il est possible de se relier aux évènements, et que nous savons comment le faire, définissons ce que doivent permettre de faire les liaisons sur les évènements.

1. En code, on peut **attacher plusieurs récepteurs** à un même évènement, donc en se liant à un évènement, on doit pouvoir en faire autant.
2. Une des erreurs les plus courantes est de relier plusieurs fois le même récepteur à un évènement.
   Dans certains cas très rares, cela peut être voulu, mais dans la majorité des cas, il s'agit d'une erreur.
   Le moteur de liaison, par défaut, doit **empêcher la liaison multiple d'un récepteur sur un évènement** tout en permettant au développeur de contourner facilement ce comportement par défaut.
3. En code, le délégué du récepteur et l'évènement partagent la même signature de méthode.
   Dans un moteur de liaison, cela est plus problématique qu'autre chose.
   Le moteur de liaison doit permettre à un récepteur d'**avoir une signature différente** de l'évènement auquel on souhaite l'attacher.
4. Au-delà des problèmes de signatures, on doit pouvoir **attacher sur un évènement autre chose qu'un EventHandler**.
   J'anticipe déjà deux exemples d'utilisation de cette fonctionnalité : relier un évènement à n'importe quelle méthode de n'importe quel objet ; et relier un évènement à une commande.
5. Tout en conservant les points précédents, on doit avoir la possibilité de **récupérer les valeurs passées par les évènements**.
6. Il arrive souvent dans le code que l'on soit obligé de déclencher une méthode de façon **asynchrone** sur un évènement ; et aussi, dans le sens inverse, que l'on soit obligé de **synchroniser** les threads avant d'appeler le code désiré. On doit pouvoir le faire aussi en liant les évènements.
7. La majorité des récepteurs sont à usage statique, et donc ; une fois attachés, toutes les informations concernant ces liaisons sont obsolètes et occupent de la mémoire pour rien.
   Le moteur de liaison doit permettre de gérer ce cas. Plus globalement, il doit permettre de **libérer la mémoire** de toutes les informations de liaisons, tout en conservant les récepteurs attachés.
8. Enfin, pour terminer, ce moteur de liaison des évènements se doit d'être **facile à utiliser**, de permettre une **extension contrôlée**, et de **ne pas être intrusif** envers les vues et composants qui l'utilisent.

Un exemple : dans le point 4, je parlais du patron de conception Commande.
C'est un des patrons de conception les plus simples, mais c'est aussi, un des patrons de conception dont je n'ai jamais vu deux fois de suite la même implémentation.
Ce genre de chose, typiquement, ne devrait pas être imposé par un moteur de liaison.

Supprimé : partage
Supprimé : é
Supprimé : qu'elle
Supprimé : code,
Supprimé : ;
Supprimé : ,

5 I-B. Le processus de liaison

Le principe d'un processus de liaison est différent d'un processus de génération de code.

En général, on peut le résumer comme suit :

Supprimé : résumé

1. définition de ce que l'on souhaite ;
2. exécution de la définition.

Le premier point concerne le *Design Time* et, si l'on fait une analogie avec la liaison de propriété, se résume à dire : *Objet1.Propriété1 reliée à Objet2.Propriété2*.

Le deuxième point concerne le *Run Time*, c'est le code exécuté en adéquation avec la définition du besoin.
Selon l'exemple ci-dessus, on obtiendrait : *Objet1.DataBindings.Add("Propriété1", Objet2, "Propriété2")*.

Ce code se retrouve habituellement dans la méthode InitializeComponent du composant qui supporte la liaison.

Un autre point à ne pas sous-estimer, c'est la capacité de manipuler les définitions des liaisons directement en code.
En effet, rien ne nous empêche de définir la ligne de code précédente de nos propres mains.

Par contre, le mécanisme réel de liaison, lui, nous est entièrement caché.
La seule chose que nous voyions (subjonctif présent) dans le code est la ligne de code précédente.
En résumé, **la définition agit comme une façade sur le traitement défini**.

Supprimé : t

6 I-C. Comment faire ?

Concernant l'ergonomie, je vois bien le même principe que dans le premier article de cette série : un onglet supplémentaire dans la *PropertyGrid* qui permet de consulter / modifier les liaisons sur les évènements.

Les onglets de propriété, en plus d'être extrêmement simple à réutiliser, sont non intrusifs pour leurs utilisateurs (s'ils sont conçus pour).

Supprimé : intrusif

Supprimé : conçu

Concernant le processus de liaison, les évènements ont une durée vie plus complexe qu'une simple liaison de propriété.
Nous devons permettre aux utilisateurs du moteur de liaison de gérer cela par un objet qui facilite ce travail via des méthodes spécifiques telles que *BindHandlers*, *UnbindHandlers* etc., etc.

Supprimé : tel

La description d'un processus de liaison standard a fait apparaître un nouveau besoin.

Supprimé : apparaitre

Les définitions de liaisons ne manipulent pas directement les objets utilisés par les liaisons, mais une abstraction de ces objets.
Dans les cas habituels, il s'agit d'un simple nom de propriété. Dans notre cas, il s'agira d'un nom d'évènement.

Nous allons devoir faire de même concernant les récepteurs.
Surtout que, vu la description des fonctionnalités souhaitées, nous n'imposons pas de type spécifique pour ces récepteurs.
L'objet qui fournira les abstractions des récepteurs sera responsable de la résolution de ces abstractions.

De plus, comme l'on souhaite que les signatures des évènements et des récepteurs puissent être différentes, nous aurons besoin d'une médiation entre ces deux objets.

Supprimé : auront

On commence à visualiser les différents intervenants utilisés par le moteur de liaison des évènements, et on commence aussi à constater sa complexité.

Afin d'éviter le traditionnel plat de spaghetti, nous allons nous baser sur l'architecture de service du concepteur *Windows Forms* pour obtenir la flexibilité et la qualité architecturale désirée.

7 III. Le moteur de liaison des évènements

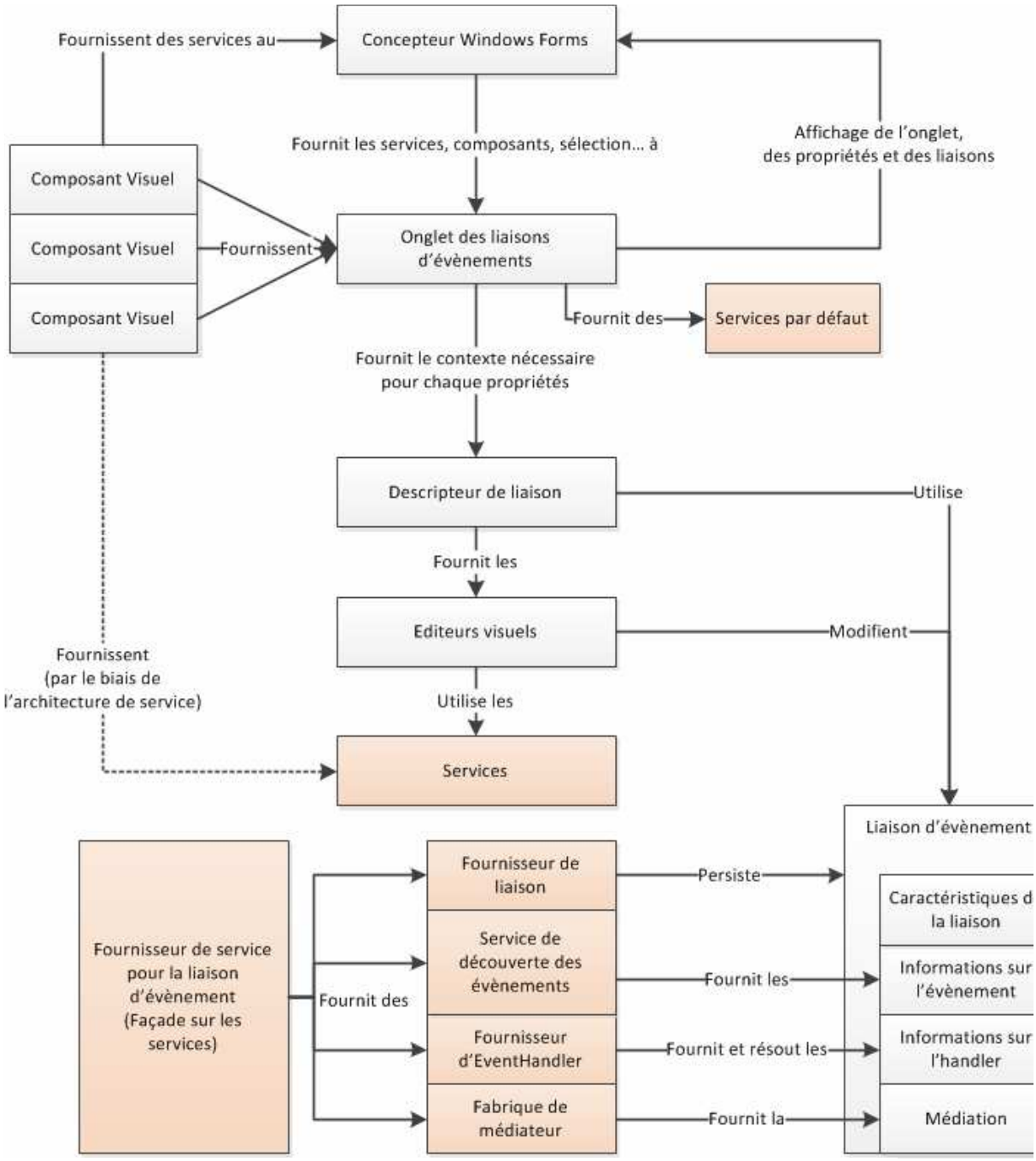
8 II-A. Principe et intervenants

Dans le schéma :

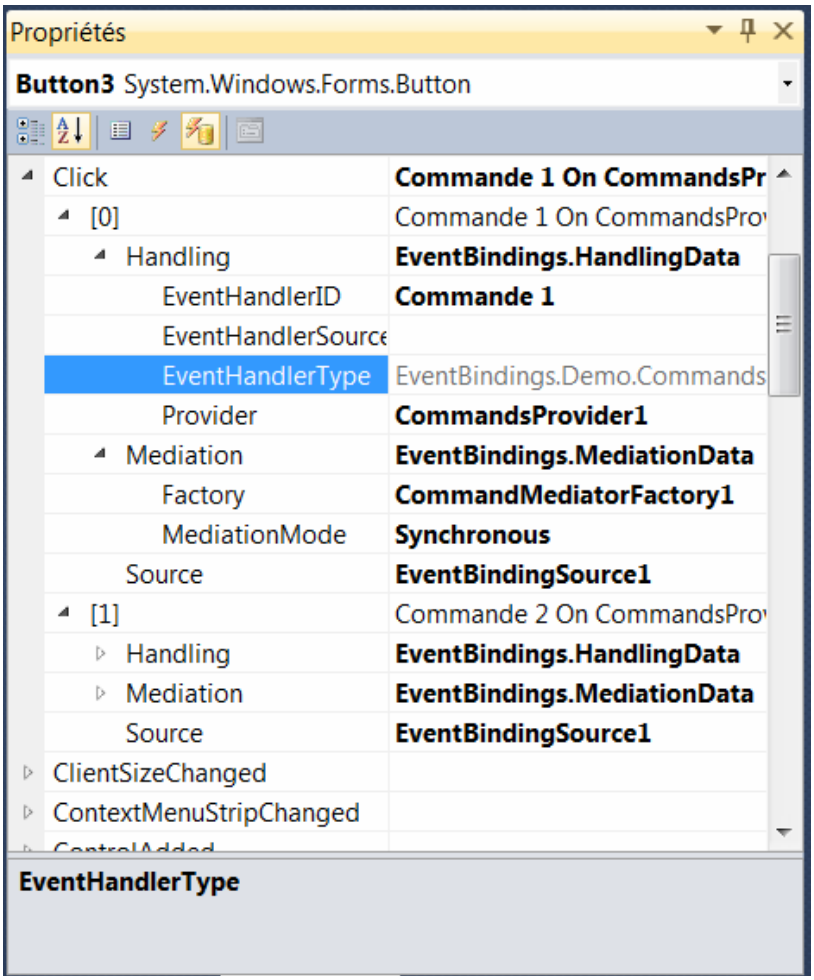
fournit le contexte pour chaque propriété (sing)

éditeurs visuels ---> modifient (ok) ----> utilisent (nok)

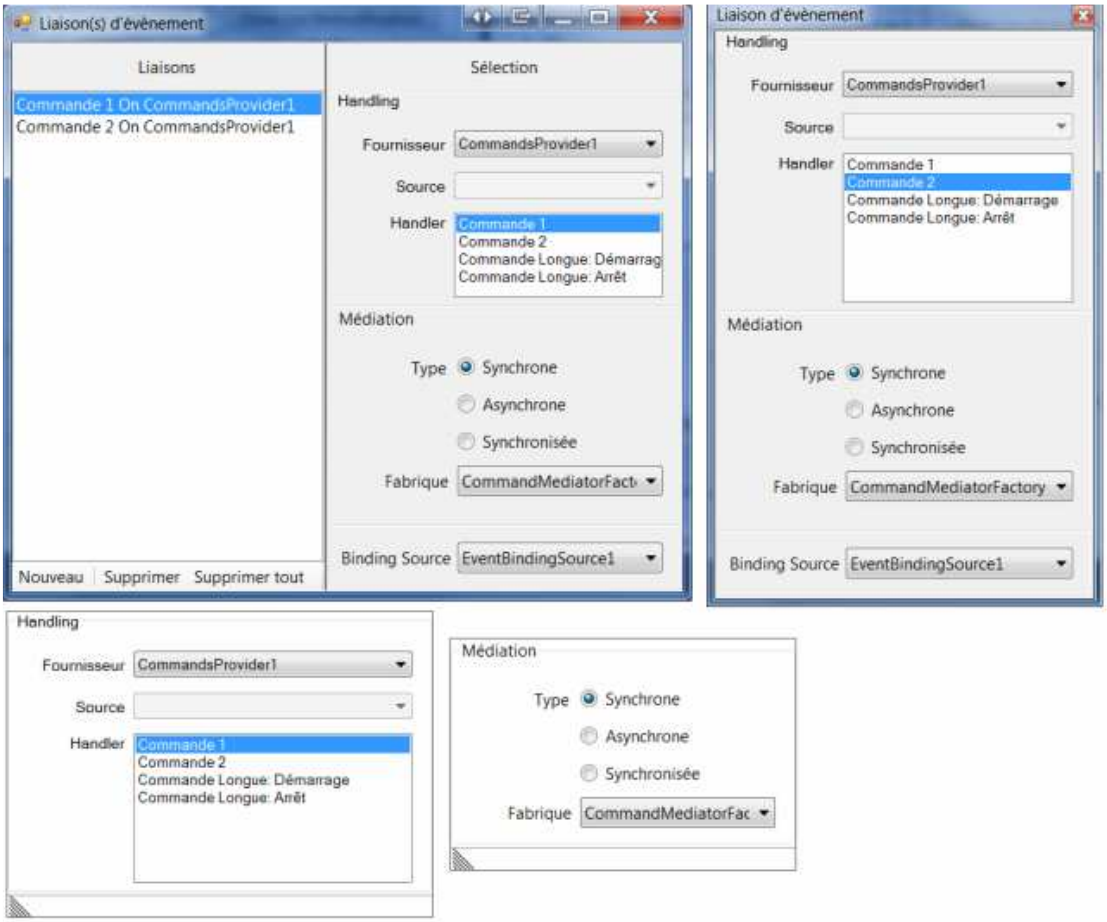
Mis en forme : Police :Times New Roman, 10 pt, Non Gras, Français (Belgique)



9 II-B. Ergonomie en "Design Time"

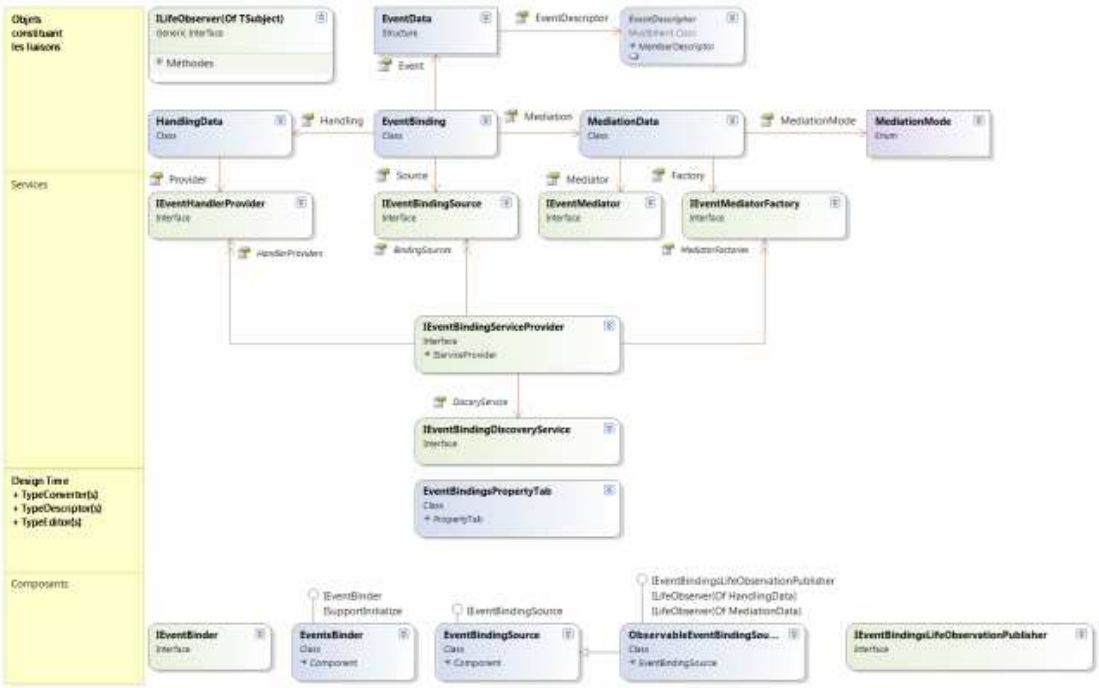


Dans l'onglet de liaison des évènements, on dispose d'une vue hiérarchique sur toutes les informations concernant les liaisons.



Chaque propriété dispose d'un éditeur visuel spécifique permettant de n'afficher que les valeurs valides et de faciliter les saisies.

10 II-C. Vue d'ensemble



Cliquer sur l'image pour l'agrandir.

Comme le montre le schéma ci-dessus, l'architecture du moteur de liaison des évènements est constituée de quatre blocs :

Supprimé : 4

- les objets constituant les liaisons ;
- les services ;
- la partie *Design Time* ;
- et les composants.

Hormis les objets constituant les liaisons, chaque responsabilité est définie par une interface, et ces interfaces, en fonction des cas, disposent d'une implémentation par défaut.

Supprimé : définit

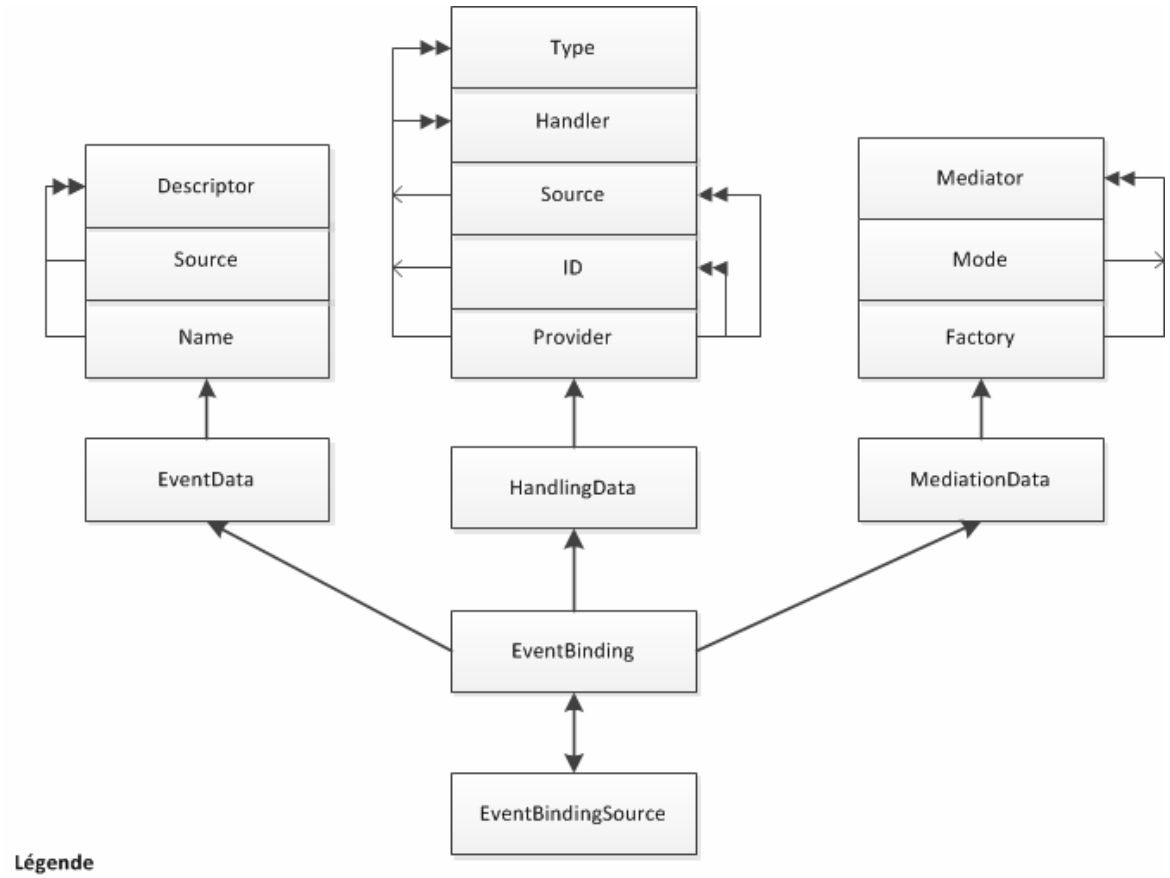
Nous allons maintenant étudier chaque bloc en détail.

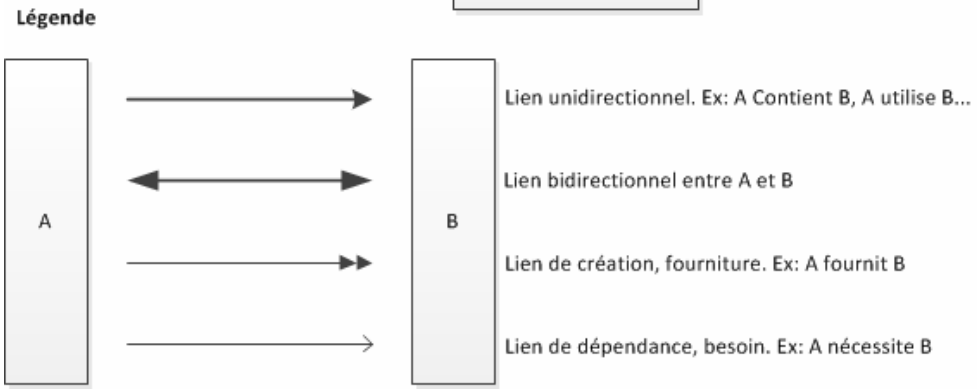
11 II-D. Les liaisons

Les objets constituant les liaisons sont séparés par responsabilité.

- Tout ce qui concerne l'évènement est dans *EventData*.
- Tout ce qui concerne les récepteurs est dans *HandlingData*.
- Tout ce qui concerne la médiation est dans *MediationData*.
- Et enfin, la liaison qui gère le tout, et à partir de laquelle on peut récupérer toutes ces informations.

12 II-D-1. Relations entre les objets de liaison

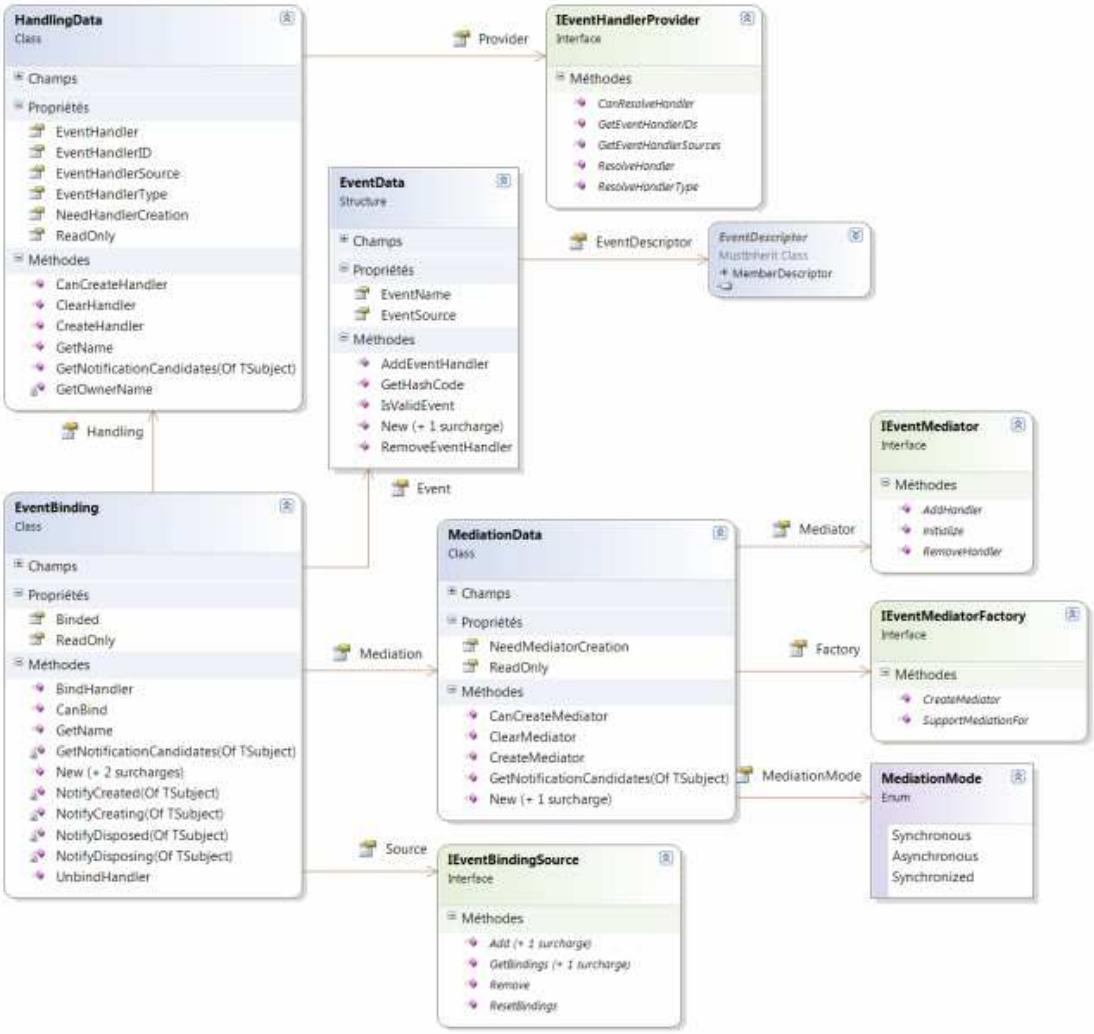




Exemples de lecture :

- *Provider* fournit *Type* en utilisant *Source* et *ID* ;
- *Source* et *Name* fournissent *Descriptor*.

13 II-D-2. Diagramme de classe



Cliquer sur l'image pour l'agrandir.

Une liaison représente un évènement relié à un récepteur par une médiation.
Les informations sur les évènements sont nécessaires dès la création de la liaison. Par contre, toutes les autres informations sont accessibles en lecture / écriture tant que les objets utiles à la liaison ne sont pas créés.
C'est la raison pour laquelle ces objets disposent d'une propriété *ReadOnly* pour indiquer leur état.

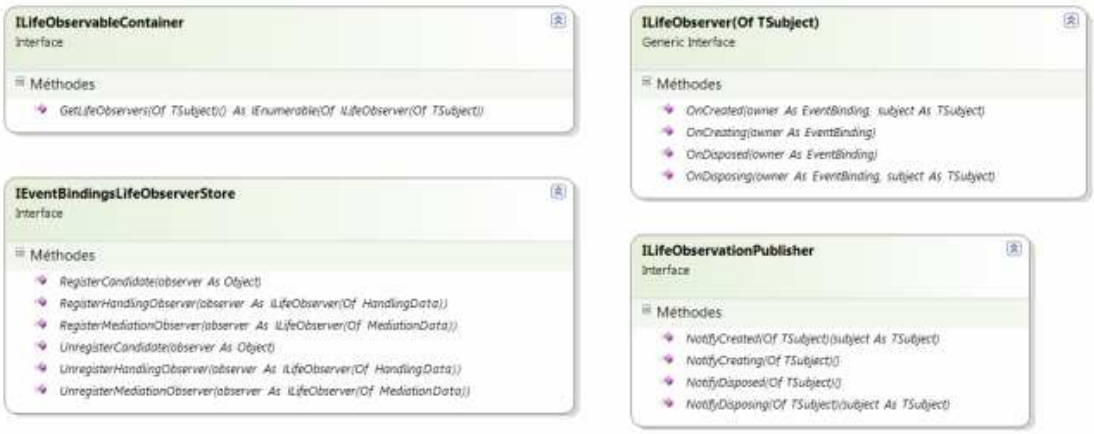
Pour ajouter un autre récepteur à un évènement, il faut créer une seconde liaison.
Une liaison est persistante si, et seulement si, elle est rattachée à une source *via* sa propriété *Source*.
Une liaison propose aussi toutes les méthodes nécessaires pour gérer le lien qu'elle représente. Ces méthodes sont *BindHandler*, *UnbindHandler*, et *CanBind*.

14 II-D-3. Observation des liaisons

Comme chaque objet constituant une liaison ne crée l'objet qu'il définit que lorsque c'est nécessaire, un mécanisme d'observation permet de surveiller la durée de vie de ces objets.

Supprimé : r

II-D-3-a. Interfaces utilisées par le processus d'observation



Cliquer sur l'image pour l'agrandir.

Ce processus d'observation s'inspire librement du patron de conception *Observer* ou Observateur en français.
Il ne peut que s'en inspirer puisque la définition du problème diffère sensiblement.

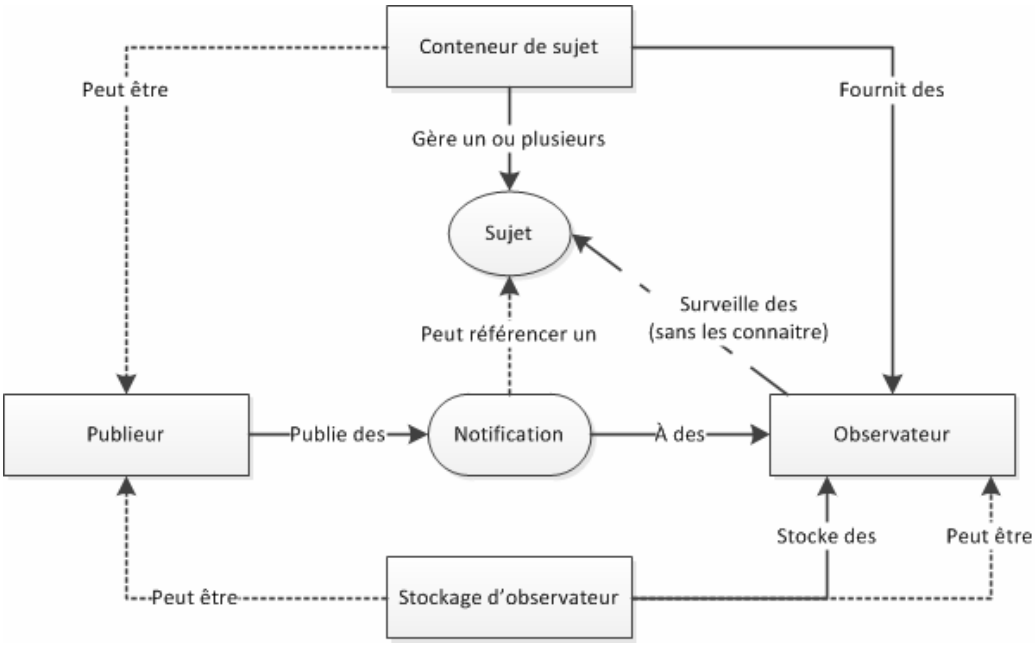
Un Observateur est conçu pour surveiller les changements d'état d'un sujet précis.
Dans notre cas, nous n'observons pas un sujet précis, et en plus, nous n'observons pas un sujet, mais sa durée de vie ; autrement dit, nous surveillons la création et la suppression d'un sujet inconnu.
De plus, nous disposons d'une architecture de service non contrainte ; ce qui signifie qu'une implémentation de service peut correspondre à plus d'un service.

Supprimé : s

Si on admet que le sujet observé est "la durée de vie d'un sujet inconnu", on peut considérer les conteneurs comme équivalent au sujet dans le patron de conception Observateur.
??? corrections peut-être intempestives Désolé???
Les conteneurs peuvent fournir des observateurs qui observent autre chose que les sujets gérés par le conteneur. Par contre, un conteneur est un conteneur de sujets, et un fournisseur potentiel d'observateurs, mais pas un conteneur d'observateurs. Autrement dit, un conteneur fournit des observateurs si, et seulement si, ses dépendances sont des observateurs.

Comme aucun des éléments classiques d'un Observateur dans cette implémentation ne permet d'ajouter des observateurs externes à ces éléments, un objet de stockage est introduit pour permettre à des observateurs externes de se relier au processus observé sans pour autant s'impliquer dans le processus observé.

II-D-3-b. Principe du processus d'observation

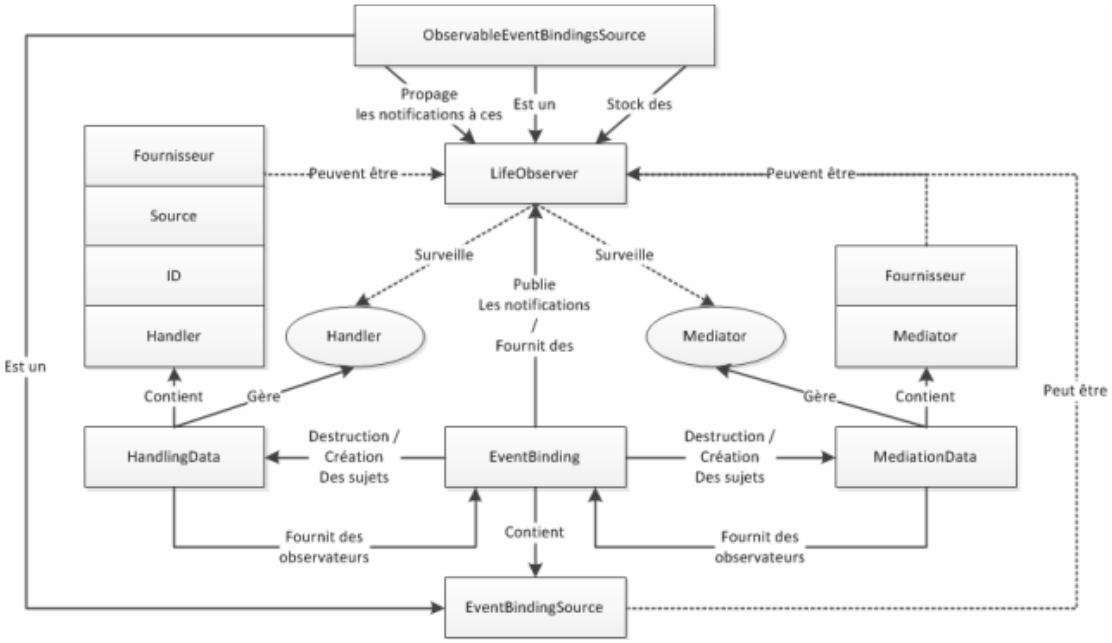


Stokage d'observateurs ??? conteneur de sujets sans les connaître

On constate que les observateurs, s'ils ne souhaitent pas être fortement couplés au conteneur, ne peuvent connaître le sujet réellement observé que lors d'une notification.

Supprimé : i

II-D-3-c. Application du principe aux objets de liaisons



Cliquer sur l'image pour l'agrandir.

Ce schéma peut paraître compliqué à première vue.
Cette relative complexité est due à la présence de deux sujets observés (*Handler* et *Mediator*).
Ce qui signifie que ce schéma représente deux processus d'observation.
De plus, un observateur extérieur est représenté dans ce schéma (*ObservableEventBindingsSource*).

Supprimé : paraitre

Supprimé : er

De par la conception des interfaces d'observation, tout objet observable agit comme un composite masquant la structure réelle des objets le composant pour ses observateurs.
Ce qui a pour conséquence, qu'il suffit de connaître un seul objet observable pour pouvoir observer n'importe quel objet observable sous-jacent, et ce, quelle que soit sa position réelle dans la structure des objets.

Supprimé : s

Supprimé : i

Une liaison (*EventBinding*) est un objet observable constitué d'autres objets observables et qui sait publier des notifications.
Chaque élément observable fournit ses propres observateurs, mais seule une liaison déclenche et gère les notifications de tous les sujets qui dépendent d'elle.
Les notifications sont déclenchées sur la liaison / déliaison d'un évènement via les méthodes *BindHandler* et *UnbindHandler*.

Supprimé : seul

Concrètement, on peut, par exemple, avoir un fournisseur de récepteur qui observe la création / suppression du médiateur qui va lui être assigné.
Pour ce faire, ce fournisseur devra implémenter l'interface d'observation correspondante, à savoir *ILifeObserver(Of MediationData)*.
Son propriétaire (*EventBinding* dans notre cas), se chargera des notifications.

Supprimé : correspondant

Les sujets observables via une liaison sont :

- *HandlingData* pour surveiller la vie du récepteur ;
- et *MediationData* pour surveiller la vie du médiateur.

Les informations concernant un récepteur, à savoir : son identifiant, son propriétaire et son type réel sont entièrement libres. Seul le fournisseur de récepteur a un type imposé (*IEventHandlerProvider*).
Les informations concernant la médiation ont toutes des types imposés (*IEventMediator* et *IEventMediatorFactory*).

15 II-D-4. Le code important des objets de liaison

Supprimé : s

Le code des objets de liaison n'a rien de particulier, il s'agit d'une composition classique entre plusieurs objets.

On notera tout de même que la composition porte sur les objets formant la définition, pas sur ce qu'ils définissent.
Autrement dit, la suppression d'une définition ne signifie pas la suppression de ce qui a été défini.

Supprimé : ;

Supprimé : définit

Voici quelques extraits de ce code, ceux que je considère comme pouvant avoir un intérêt.

```
Public Class EventBinding

    Public ReadOnly Property [ReadOnly] As Boolean
        Get
            Return Me._Binded
        End Get
    End Property

    Public Property Source As IEventBindingSource
        Get
            Return Me._Source
        End Get
        Set(ByVal value As IEventBindingSource)
            If Me._Source IsNot Nothing Then Me._Source.Remove(Me)
            Me._Source = value
            If Me._Source IsNot Nothing Then Me._Source.Add(Me)
        End Set
    End Property

    Public Function CanBind() As Boolean
        If Not Me._Event.IsValidEvent Then Return False
        If Me._Handling Is Nothing OrElse Not
Me._Handling.CanCreateHandler Then Return False
        If Me._Mediation Is Nothing OrElse
            Not Me.Mediation.CanCreateMediator(Me._Event,
Me._Handling) Then Return False
        Return True
    End Function
    Public Sub BindHandler()
        If Me._Handling.NeedHandlerCreation Then
            Me.NotifyCreating(Of HandlingData)()
            Me._Handling.CreateHandler()
            Me.NotifyCreated(Of HandlingData)(Me._Handling)
        End If
        If Me._Mediation.NeedMediatorCreation Then
            Me.NotifyCreating(Of MediationData)()
            Me._Mediation.CreateMediator(Me._Event, Me._Handling)
```

```
            Me.NotifyCreated(Of MediationData)(Me._Mediation)
        End If
        Me._Mediation.Mediator.AddHandler()
        Me._Binded = True
    End Sub
    Public Sub UnbindHandler()
        If Me._Mediation.Mediator IsNot Nothing Then
            Me.NotifyDisposing(Of MediationData)(Me._Mediation)
            Me._Mediation.Mediator.RemoveHandler()
            Me._Mediation.ClearMediator()
            Me.NotifyDisposed(Of MediationData)()
        End If
        Me.NotifyDisposing(Of HandlingData)(Me._Handling)
        Me._Handling.ClearHandler()
        Me.NotifyDisposed(Of HandlingData)()
        Me._Binded = False
    End Sub

End Class
```

On notera que la propriété *Source* est modifiable tout le temps. Ceci afin de permettre une gestion fine de l'occupation mémoire par les utilisateurs du moteur de liaison.

```
Public Class HandlingData

    Public Sub ClearHandler()
        Me._EventHandler = Nothing
    End Sub
    Public Function CanCreateHandler() As Boolean
        If Me._Provider Is Nothing Then Return False
        Return Me._Provider.CanResolveHandler(Me._EventHandlerID,
Me._EventHandlerSource)
    End Function
    Public Sub CreateHandler()
        Me._EventHandler =
Me._Provider.ResolveHandler(Me._EventHandlerID,
Me._EventHandlerSource)
    End Sub

    Public ReadOnly Property EventHandler As Object
        Get
            Return Me._EventHandler
        End Get
    End Property
    Public ReadOnly Property EventHandlerType As Type
        Get
            If Me._EventHandlerType Is Nothing AndAlso
Me.CanCreateHandler Then
                Me._EventHandlerType =
Me._Provider.ResolveHandlerType(
                    Me._EventHandlerID,
                    Me._EventHandlerSource)
            End If
            Return Me._EventHandlerType
        End Get
    End Property
    Public ReadOnly Property NeedHandlerCreation As Boolean
        Get
            Return Me._EventHandler Is Nothing
        End Get
    End Property
    Public ReadOnly Property [ReadOnly] As Boolean
```

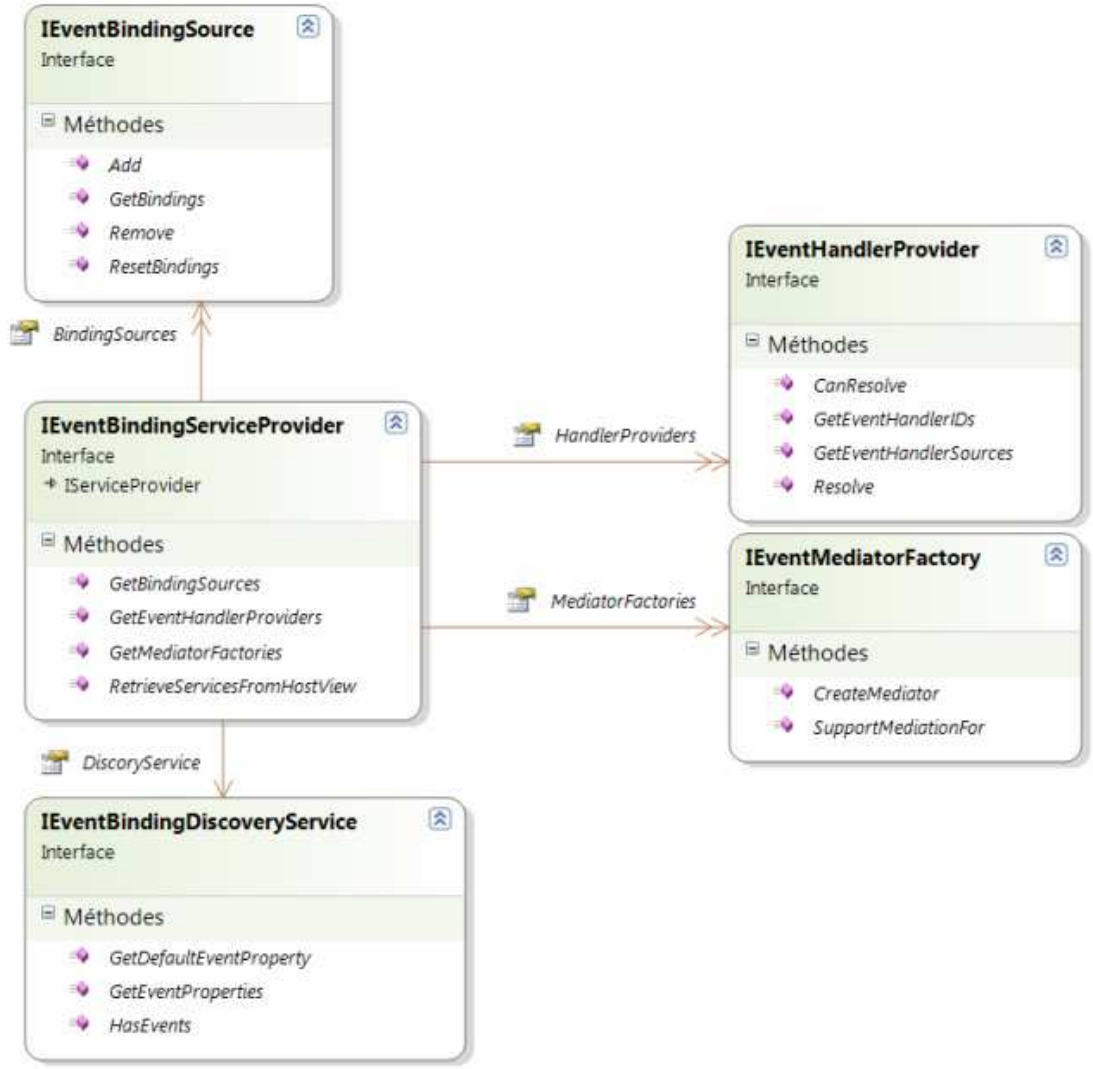
```
            Get
                Return Me._EventHandler IsNot Nothing
            End Get
        End Property

    End Class
```

On peut observer dans ce code la logique de gestion d'un récepteur.
Toutes les propriétés modifiables de cet objet sont dépendantes de la propriété *ReadOnly* et donc, ne sont modifiables que si le délégué du récepteur n'a pas été créé.

```
    Public Class MediationData

        Public Sub ClearMediator()
            Me._Mediator = Nothing
        End Sub
        Public Function CanCreateMediator(ByVal [event] As EventData,
            ByVal handler As HandlingData) As Boolean

            If Me._Factory Is Nothing Then Return False
            Return Me._Factory.SupportMediationFor([event], handler,
    Me._MediationMode)
        End Function
        Public Sub CreateMediator(ByVal [event] As EventData, ByVal
    handler As HandlingData)
            Me._Mediator = Me._Factory.CreateMediator([event], handler,
    Me._MediationMode)
        End Sub

        Public ReadOnly Property [ReadOnly] As Boolean
            Get
                Return Me._Mediator IsNot Nothing
            End Get
        End Property
        Public ReadOnly Property NeedMediatorCreation As Boolean
            Get
                Return Me._Mediator Is Nothing
            End Get
        End Property
        Public ReadOnly Property Mediator As IEventMediator
            Get
                Return Me._Mediator
            End Get
        End Property

    End Class
```

Même chose que dans le code précédent, mais concernant la médiation. Cette fois, les propriétés modifiables dépendent de l'existence du médiateur.

16 II-E. Les services



Un **IEventBindingServiceProvider** est une façade permettant l'accès à tous les services utilisés par le moteur de liaison. Cette façade hérite de l'interface IServiceProvider afin de pouvoir être utilisée en lieu et place du IServiceProvider habituel.

La façade fournie par défaut s'initialise automatiquement en fonction des services implémentés par les composants de la vue et encapsule le IServiceProvider du concepteur visuel.

Un **IEventBindingSource** permet le stockage et la restitution des liaisons. C'est ce service qui est responsable de la persistance en code des définitions de liaison.

Un **IEventHandlerProvider** représente un fournisseur de récepteur ; il est responsable de l'abstraction des récepteurs, ainsi que de la résolution de ces abstractions.
Dans le cas d'un fournisseur exposant des récepteurs, mais ne possédant pas ces récepteurs (fonctionnement par extension), il peut aussi fournir les différentes sources de récepteur supportées.

Un **IEventMediatorFactory** est une fabrique de médiateur.

Un **IEventBindingDiscoveryService** est un service de découverte des évènements. C'est ce service qui est responsable de l'exposition des évènements sous forme de propriété *via* des descripteurs de propriété.

L'implémentation du service de découverte des évènements fournie par défaut expose tous les évènements d'un objet sans les filtrer, et utilise l'ergonomie vue précédemment pour l'édition des liaisons.
Le comportement en *Design Time* peut être modifié intégralement en fournissant une autre implémentation de ce service.

17 II-F. Le "Design Time"

Le *Design Time* est composé d'un onglet de propriété fournissant les liaisons des évènements et d'une multitude d'éditeurs visuels.

Supprimé : d'éditeur

L'onglet de propriété s'appelle *EventBindingsPropertyTab*. Son fonctionnement est extrêmement basique.

Il initialise ou récupère une façade sur son constructeur.

```
Public Sub New(ByVal sp As IServiceProvider)
    MyBase.New()
    If sp Is Nothing Then Throw New ArgumentNullException("sp")
    Me._ServiceProvider = sp.GetOrCreateEventBindingServiceProvider
    Me._ServiceProvider.RetrieveServicesFromHostView()
End Sub
```

Puis, pour tout ce qui concerne les propriétés, fait appel au service de découverte fourni par la façade.

Supprimé : appelle

Supprimé : fournit

```
Public Overrides Function CanExtend(ByVal extendee As Object) As
Boolean
    Return Me.DiscoveryService.HasEvents(extendee)
End Function

Public Overrides Function GetDefaultProperty(ByVal component As
Object) As PropertyDescriptor
    Return Me.DiscoveryService.GetDefaultEventProperty(component)
End Function

Public Overloads Overrides Function GetProperties(
    ByVal component As Object,
    ByVal attributes() As Attribute) As PropertyDescriptorCollection

    Return Me.DiscoveryService.GetEventProperties(component,
attributes)
End Function
Public Overrides Function GetProperties(ByVal component As Object)
As PropertyDescriptorCollection
    Return Me.DiscoveryService.GetEventProperties(component)
End Function
```

Sa seule intelligence est la suivante.

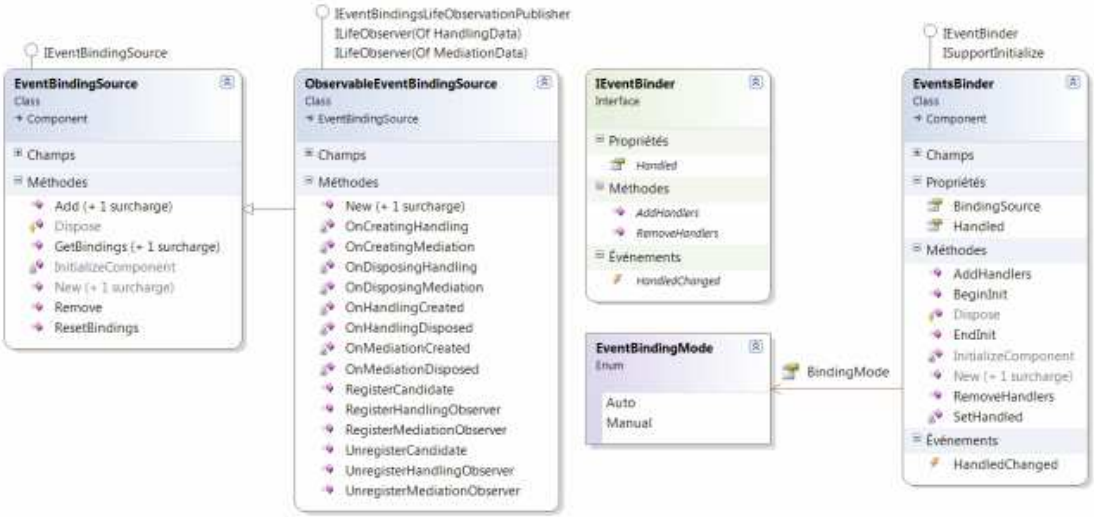
```
Public Overrides Function GetProperties(
    ByVal context As ITypeDescriptorContext,
    ByVal component As Object,
    ByVal attributes() As Attribute) As PropertyDescriptorCollection

    If context.IsRootComponent(component) Then
        Return Me.DiscoveryService.GetEventProperties(component,
attributes)
    ElseIf TypeOf component Is EventBinding() Then
        Dim Converter As New ArrayConverter
        Return Converter.GetProperties(context, component,
attributes)
    Else
        Return TypeDescriptor.GetProperties(component, attributes)
    End If
End Function
```

18 II-G. Les composants

Les composants sont des objets que l'on va poser dans notre vue pour y ajouter des fonctionnalités.
Leur utilisation est facultative, leur présence est juste là pour faciliter l'utilisation du moteur de liaison des évènements.


Si vous n'utilisez pas ceux-là, il vous faudra fournir vous-mêmes les implémentations des services requis par votre usage du moteur de liaison.

19 II-G-1. Les composants fournis par défaut



Cliquer sur l'image pour l'agrandir.

IEventBinder représente l'interface par défaut d'un objet responsable de la gestion de plusieurs liaisons.
Son implémentation par défaut (*EventsBinder*), propose un mode de liaison automatique qui lie les évènements après l'initialisation via la méthode EndInit.

EventBindingSource est l'implémentation par défaut du service *IEventBindingSource*. Elle propose un stockage simple des liaisons à base de listes ???, un accès indexé par *EventData* à ces liaisons (à base de dictionnaire), ainsi qu'une sérialisation par CodeDom des définitions de liaison.

ObservableEventBindingSource étend *EventBindingSource* en lui ajoutant les fonctionnalités d'observation vues précédemment.

20 II-G-2. Le code des composants

EventBindingSource

```
<DesignerSerializer(GetType(Design.EventBindingSourceSerializer),
GetType(CodeDomSerializer))>
Public Class EventBindingSource
    Implements IEventBindingSource

    Private _EventBindings As New Dictionary(Of EventData, List(Of
EventBinding))
    Private _Bindings As New List(Of EventBinding)

    Public Function Add(ByVal eventName As String,
                        ByVal eventSource As Object,
                        ByVal handlerID As Object,
                        ByVal handlerProvider As
IEventHandlerProvider,
                        ByVal handlerSource As Object,
```

```
                        ByVal mediationFactory As
IEventMediatorFactory,
                        ByVal mediationMode As MediationMode
                        ) As EventBinding Implements
IEventBindingSource.Add
        Dim ToAdd As New EventBinding(eventName, eventSource)
        ToAdd.Handling = New HandlingData With {
            .Provider = handlerProvider, .EventHandlerSource =
handlerSource, .EventHandlerID = handlerID}
        ToAdd.Mediation = New MediationData(mediationFactory,
mediationMode)
        ToAdd.Source = Me
        Return ToAdd
    End Function
    Public Sub Add(ByVal binding As EventBinding) Implements
IEventBindingSource.Add
        If binding Is Nothing Then Throw New
ArgumentNullException("binding")
        If Me._Bindings.Contains(binding) Then Exit Sub
        Me._Bindings.Add(binding)
        Dim Index As List(Of EventBinding) = Nothing
        If Not Me._EventBindings.TryGetValue(binding.Event, Index)
Then
            Index = New List(Of EventBinding)
            Me._EventBindings.Add(binding.Event, Index)
        End If
        Index.Add(binding)
    End Sub
    Public Sub Remove(ByVal binding As EventBinding) Implements
IEventBindingSource.Remove
        If binding Is Nothing Then Throw New
ArgumentNullException("binding")
        Dim Index As List(Of EventBinding) = Nothing
        If Me._EventBindings.TryGetValue(binding.Event, Index) Then
Index.Remove(binding)
        Me._Bindings.Remove(binding)
    End Sub

    Public Sub ResetBindings(ByVal [event] As EventData)
        Implements IEventBindingSource.ResetBindings

        Dim Index As List(Of EventBinding) = Nothing
        If Not Me._EventBindings.TryGetValue([event], Index) Then
Exit Sub
        For Each item In Index
            Me._Bindings.Remove(item)
        Next
        Index.Clear()
        Me._EventBindings.Remove([event])
    End Sub
    Public Function GetBindings(ByVal [event] As EventData) As
IEnumerable(Of EventBinding)
        Implements IEventBindingSource.GetBindings

        Dim Index As List(Of EventBinding) = Nothing
        If Not Me._EventBindings.TryGetValue([event], Index) Then
Return Nothing
        Return Index.ToArray
    End Function
    Public Function GetBindings() As IEnumerable(Of EventBinding)
        Implements IEventBindingSource.GetBindings
```

```
            Return Me._Bindings.ToArray
        End Function
    End Class
```

Un simple stockage des liaisons avec un indexage par évènement / composant.

La persistance est gérée par un sérialiser CodeDom qui ne dépend que de l'interface *IEventBindingSource*, et que l'on peut donc réutiliser comme bon nous semble.

Ce sérialiser produit le code suivant pour chaque liaison :

```
Me.EventBindingSource1.Add("Click", Me.Button4, "Close",
Me.MethodBindingProvider1,
    Me, Me.MethodBindingProvider1,
EventBindings.MediationMode.Synchronous)
```

ObservableEventBindingSource

```
Public Class ObservableEventBindingSource
    Inherits EventBindingSource
    Implements IEventBindingsLifeObserverStore
    Implements ILifeObserver(Of HandlingData)
    Implements ILifeObserver(Of MediationData)

    Private _HandlingObservers As New List(Of ILifeObserver(Of
HandlingData))
    Private _MediationObservers As New List(Of ILifeObserver(Of
MediationData))

    Public Sub RegisterCandidate(ByVal observer As Object)
        Implements IEventBindingsLifeObserverStore.RegisterCandidate

        If observer Is Nothing Then Exit Sub
        If TypeOf observer Is ILifeObserver(Of HandlingData) Then
            Me.RegisterHandlingObserver(DirectCast(observer,
ILifeObserver(Of HandlingData)))
        End If
        If TypeOf observer Is ILifeObserver(Of MediationData) Then
            Me.RegisterMediationObserver(DirectCast(observer,
ILifeObserver(Of MediationData)))
        End If
    End Sub
    Public Sub RegisterHandlingObserver(ByVal observer As
ILifeObserver(Of HandlingData))
        Implements
IEventBindingsLifeObserverStore.RegisterHandlingObserver

        If observer Is Nothing Then Exit Sub
        Me._HandlingObservers.AddDistinct(observer)
    End Sub
    Public Sub RegisterMediationObserver(ByVal observer As
ILifeObserver(Of MediationData))
        Implements
IEventBindingsLifeObserverStore.RegisterMediationObserver

        If observer Is Nothing Then Exit Sub
        Me._MediationObservers.AddDistinct(observer)
    End Sub

    Public Sub UnregisterCandidate(ByVal observer As Object)
```

```
        Implements
IEventBindingsLifeObserverStore.UnregisterCandidate

        If observer Is Nothing Then Exit Sub
        If TypeOf observer Is ILifeObserver(Of HandlingData) Then
            Me.UnregisterHandlingObserver(DirectCast(observer,
ILifeObserver(Of HandlingData)))
        End If
        If TypeOf observer Is ILifeObserver(Of MediationData) Then
            Me.UnregisterMediationObserver(DirectCast(observer,
ILifeObserver(Of MediationData)))
        End If
    End Sub
    Public Sub UnregisterHandlingObserver(ByVal observer As
ILifeObserver(Of HandlingData))
        Implements
IEventBindingsLifeObserverStore.UnregisterHandlingObserver

        If observer Is Nothing Then Exit Sub
        Me._HandlingObservers.Remove(observer)
    End Sub
    Public Sub UnregisterMediationObserver(ByVal observer As
ILifeObserver(Of MediationData))
        Implements
IEventBindingsLifeObserverStore.UnregisterMediationObserver

        If observer Is Nothing Then Exit Sub
        Me._MediationObservers.Remove(observer)
    End Sub

    Private Sub OnCreatingHandling(ByVal owner As EventBinding)
        Implements ILifeObserver(Of HandlingData).OnCreating

        Me._HandlingObservers.NotifyCreating(owner)
    End Sub
    Private Sub OnHandlingCreated(ByVal owner As EventBinding, ByVal
subject As HandlingData)
        Implements ILifeObserver(Of HandlingData).OnCreated

        Me._HandlingObservers.NotifyCreated(owner, subject)
    End Sub
    Private Sub OnDisposingHandling(ByVal owner As EventBinding,
ByVal subject As HandlingData)
        Implements ILifeObserver(Of HandlingData).OnDisposing

        Me._HandlingObservers.NotifyDisposing(owner, subject)
    End Sub
    Private Sub OnHandlingDisposed(ByVal owner As EventBinding)
        Implements ILifeObserver(Of HandlingData).OnDisposed

        Me._HandlingObservers.NotifyDisposed(owner)
    End Sub

    Private Sub OnCreatingMediation(ByVal owner As EventBinding)
        Implements ILifeObserver(Of MediationData).OnCreating

        Me._MediationObservers.NotifyCreating(owner)
    End Sub
    Private Sub OnMediationCreated(ByVal owner As EventBinding,
ByVal subject As MediationData)
        Implements ILifeObserver(Of MediationData).OnCreated
```

```
            Me._MediationObservers.NotifyCreated(owner, subject)
        End Sub
        Private Sub OnDisposingMediation(ByVal owner As EventBinding,
    ByVal subject As MediationData)
            Implements ILifeObserver(Of MediationData).OnDisposing

            Me._MediationObservers.NotifyDisposing(owner, subject)
        End Sub
        Private Sub OnMediationDisposed(ByVal owner As EventBinding)
            Implements ILifeObserver(Of MediationData).OnDisposed

            Me._MediationObservers.NotifyDisposed(owner)
        End Sub
    End Class
```

Dans ce code, une source de liaison (la même que précédemment) qui permet de stocker des observateurs extérieurs, et qui propage les notifications à ces observateurs.

EventsBinder

```
    Public Class EventsBinder
        Implements IEventBinder
        Implements ISupportInitialize

        Public Property BindingMode As EventBindingMode
        Public Property BindingSource As IEventBindingSource
        <DefaultValue(CStr(Nothing))>
        <AttributeProvider(GetType(IListSource))>
        Public Property DataSource As Object

        Public Sub AddHandlers() Implements IEventBinder.AddHandlers
            If Me.BindingSource Is Nothing Then Throw New
    NullReferenceException("BindingSource")
            For Each Binding In Me.BindingSource.GetBindings
                If Not Binding.CanBind Then Continue For
                Binding.BindHandler()
            Next
            Me.SetHandled(True)
        End Sub
        Public Sub RemoveHandlers() Implements
    IEventBinder.RemoveHandlers
            If Me.BindingSource Is Nothing Then Throw New
    NullReferenceException("BindingSource")
            For Each Binding In Me.BindingSource.GetBindings
                If Not Binding.CanBind Then Continue For
                Binding.UnbindHandler()
            Next
            Me.SetHandled(False)
        End Sub

        Public Sub BeginInit() Implements
    System.ComponentModel.ISupportInitialize.BeginInit
        End Sub
        Public Sub EndInit() Implements
    System.ComponentModel.ISupportInitialize.EndInit
            If Me.DesignMode Then Exit Sub
            If Me.BindingMode <> EventBindingMode.Auto Then Exit Sub
            If Me.MustUseDataSourceEvent Then
                Me.ManageDataSource()
            Else
```

```
            Me.AddHandlers()
        End If
    End Sub

    Private Sub ManageDataSource()
        AddHandler DirectCast(Me.DataSource,
BindingSource).DataSourceChanged, AddressOf OnDataSourceChanged
    End Sub
    Private Function MustUseDataSourceEvent() As Boolean
        Return Me.DataSource IsNot Nothing AndAlso TypeOf
Me.DataSource Is BindingSource
    End Function

    Private Sub OnDataSourceChanged(ByVal sender As Object, ByVal e
As EventArgs)
        If Me._Handled Then Me.RemoveHandlers()
        Me.AddHandlers()
    End Sub
End Class
```

Une implémentation basique de l'interface *IEventBinder* qui permet de gérer les liaisons d'une source de liaison manuellement, ou automatiquement si utilisée avec une source de données.

21 IV. Démonstration

Nous savons maintenant que les utilisateurs du moteur de liaison doivent fournir des implémentations pour la médiation et la gestion des récepteurs. Nous savons aussi que ces utilisateurs peuvent fournir des implémentations de tous les services utilisés s'ils le souhaitent.

Dans cette démonstration, nous allons utiliser les services fournis par défaut, et implémenter le nécessaire afin de pouvoir relier les évènements à des actions ou fonctions présentes sur les objets composant la vue.

Supprimé : fournit

22 III-A. Les types d'évènement

En DotNet, on peut distinguer quatre types d'évènement, et donc quatre types de délégué pour les récepteurs.

Supprimé : 4

Supprimé : 4

Plus d'information à ce sujet sur MSDN.
Il est bon de savoir qu'un évènement est du type de son délégué.

Les évènements standard ___(en principe, l'adjectif reste invariable, à toi de choisir : le pluriel est toléré. Par contre, le substantif prend « s » au pluriel)

Supprimé : Standards

```
Public Event Standard As EventHandler
Private Sub OnStandardEvent(ByVal sender As Object, ByVal e As
EventArgs)
```

Mis en forme : Français (France)

```
End Sub
Dim StandardHandler As New EventHandler(AddressOf OnStandardEvent)
```

Les évènements génériques

```
Public Event Generic As EventHandler(Of MouseEventArgs)
Private Sub OnGenericEvent(ByVal sender As Object, ByVal e As
MouseEventArgs)
End Sub
Dim GenericHandler As New EventHandler(Of MouseEventArgs)(AddressOf
OnGenericEvent)
```

Les évènements standard mais custom

Supprimé : S
Supprimé : s
Supprimé : C

```
Public Delegate Sub CustomEventHandler(ByVal sender As Object, ByVal
e As MouseEventArgs)
Public Event Custom As CustomEventHandler
Private Sub OnCustomEvent(ByVal sender As Object, ByVal e As
MouseEventArgs)
End Sub
Dim CustomHandler As New CustomEventHandler(AddressOf OnCustomEvent)
```

Les évènements exotiques

Supprimé : Exotiques

```
Public Delegate Sub ExoticEventHandler(ByVal x As Int32, ByVal y As
Int32, ByRef cancel As Boolean)
Public Event Exotic As ExoticEventHandler
Private Sub OnExoticEvent(ByVal x As Int32, ByVal y As Int32, ByRef
cancel As Boolean)
End Sub
Dim ExoticHandler As New ExoticEventHandler(AddressOf OnExoticEvent)
```

Ces appellations n'ont rien d'officiel, mais je devais distinguer ces types de par leur différence d'utilisation. Donc je devais les nommer.

Supprimé : distingués

Vous aurez compris qu'un évènement est de type standard s'il possède deux arguments :

Supprimé : Standard
Supprimé : 2

- un argument *Sender* de type Object ;
- un argument de type EventArgs ;

et si ces deux arguments sont passés par valeur.

Un évènement dit **exotique** peut avoir autant d'arguments qu'il le souhaite, et il peut aussi passer ses arguments par référence, par contre, un évènement ne peut en aucun cas avoir de valeur de retour (un évènement ne peut être une fonction).

Supprimé : Exotique
Supprimé : d'argument

23 III-B. La médiation

Pour cette démonstration, nous allons gérer les cas de médiation suivants :

Supprimé : s

- d'un côté, les évènements standard (nommés *Standard*, *Générique* et *Custom* si on utilise les appellations précédentes) ;
- de l'autre, des fonctions ou actions avec un nombre d'arguments allant de 0 à 2 au maximum.

Nous allons gérer ces cas de médiation dans les différents modes prévus par la médiation, à savoir : Synchrone, Asynchrone et Synchronisé.

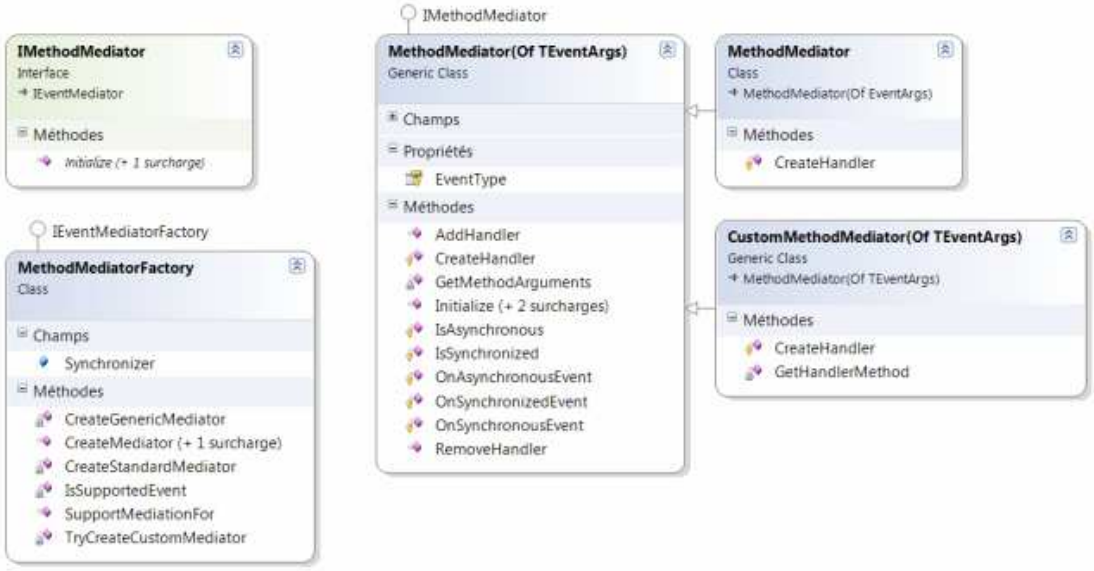
Les arguments des méthodes sont au choix, et sans ordre fixe.

Supprimé : x

- Dans le cas d'une fonction, la valeur de retour est tout simplement ignorée.
- Pour recevoir l'argument *Sender* de l'évènement dans la méthode, celle-ci doit déclarer un argument nommé "sender" (la casse importe peu) de type *Object*.
- Pour recevoir l'argument *e (l'EventArgs)*, la méthode doit déclarer un argument capable de recevoir la valeur produite par l'évènement. Son nom n'a aucune importance.

Les interfaces à implémenter pour fournir une médiation sont *IEventMediator* et *IEventMediatorFactory*.
Voici leurs définitions :

Cliquer sur l'image pour l'agrandir.
Voici leurs implémentations dans la démonstration :



Cliquer sur l'image pour l'agrandir.

L'implémentation principale est un médiateur générique qui supporte les évènements *génériques* et dont l'argument générique est le type de l'argument (*EventArgs*) utilisé par l'évènement. Puis nous spécialisons cette implémentation par héritage pour les évènements ~~*standard*~~ et ~~*custom*~~.

Supprimé : *S*

Supprimé : *s*

Supprimé : *Custom*

La fabrique fait le choix du médiateur à utiliser en fonction du type d'évènement.

Le code de la fabrique de médiateur

```
Public Class MethodMediatorFactory
    Implements IEventMediatorFactory

    Public Synchronizer As ISynchronizeInvoke

    Public Function SupportMediationFor(ByVal [event] As EventData,
        ByVal handling As HandlingData,
        ByVal mediationMode As MediationMode) As Boolean
        Implements IEventMediatorFactory.SupportMediationFor

        If mediationMode = EventBindings.MediationMode.Synchronized
AndAlso
            Me.Synchronizer Is Nothing Then Return False
        If Not Me.IsSupportedEvent([event].EventDescriptor) Then
Return False
        If handling Is Nothing Then Return False
        Return handling.EventHandlerType.IsType(Of MethodInfo)()
    End Function
    Private Function IsSupportedEvent(ByVal descriptor As
EventDescriptor) As Boolean
        If descriptor Is Nothing Then Return False
        If descriptor.IsEventHandler Then Return True
        If descriptor.IsGenericEventHandler Then Return True
        If descriptor.IsCustomEventHandler Then Return True
        Return False
    End Function
```

```
        Public Function CreateMediator(ByVal [event] As EventData,
            ByVal handling As HandlingData,
            ByVal mediationMode As MediationMode) As IEventMediator
            Implements IEventMediatorFactory.CreateMediator

            Dim Mediator = Me.CreateMediator([event])
            If Mediator Is Nothing Then Throw New NotSupportedException
            Mediator.Initialize([event])
            Mediator.Initialize(DirectCast(handling.EventHandler,
    MethodInfo),
                handling.EventHandlerSource)
            Mediator.Initialize(mediationMode, Me.Synchronizer)
            Return Mediator
        End Function
        Private Function CreateMediator(ByVal [event] As EventData) As
    IMethodMediator
            If [event].EventDescriptor.IsEventHandler Then Return
    Me.CreateStandardMediator
            If [event].EventDescriptor.IsGenericEventHandler Then Return
    Me.CreateGenericMediator([event])
            Return Me.TryCreateCustomMediator([event])
        End Function
        Private Function CreateStandardMediator() As IMethodMediator
            Return New MethodMediator
        End Function
        Private Function CreateGenericMediator(ByVal [event] As
    EventData) As IMethodMediator
            Dim MediatorArg =
    [event].EventDescriptor.EventType.GetGenericArguments(0)
            Dim MediatorGenDef = GetType(MethodMediator(Of ))
            Dim MediatorType = MediatorGenDef.MakeGenericType(New Type()
    {MediatorArg})
            Dim Mediator =
    DirectCast(Activator.CreateInstance(MediatorType), IMethodMediator)
            Return Mediator
        End Function
        Private Function TryCreateCustomMediator(ByVal [event] As
    EventData) As IMethodMediator
            Dim MediatorArg =
    [event].EventDescriptor.GetCustomEventArgsType
            If MediatorArg Is Nothing Then Return Nothing
            Dim MediatorGenDef = GetType(CustomMethodMediator(Of ))
            Dim MediatorType = MediatorGenDef.MakeGenericType(New Type()
    {MediatorArg})
            Dim Mediator =
    DirectCast(Activator.CreateInstance(MediatorType), IMethodMediator)
            Return Mediator
        End Function

    End Class
```

La fabrique se contente de choisir le type de médiateur en adéquation avec le type d'évènement et de construire ces médiateurs à l'aide du type de l'argument (*EventArgs*) utilisé par l'évènement.

Pour ce faire elle utilise des extensions dont voici le code.

```
    <Extension()>
    Public Function IsEventHandler(ByVal ED As EventDescriptor) As
    Boolean
        Return ED.EventType Is GetType(EventHandler)
```

```
End Function

<Extension()>
Public Function IsGenericEventHandler(ByVal ED As EventDescriptor)
As Boolean
    If Not ED.EventType.IsGenericType Then Return False
    Dim GenDefinition = ED.EventType.GetGenericTypeDefinition
    Return GenDefinition Is GetType(EventHandler(Of ))
End Function

<Extension()>
Public Function IsCustomEventHandler(ByVal ED As EventDescriptor) As
Boolean
    Dim InvokeMethod = ED.EventType.GetMethod("Invoke")
    If InvokeMethod Is Nothing Then Return False
    If InvokeMethod.ReturnType IsNot GetType(Void) Then Return False
    Dim Parameters = InvokeMethod.GetParameters
    If Parameters.IsNullOrEmpty Then Return False
    If Parameters.Count <> 2 Then Return False
    Return Parameters(0).ParameterType Is GetType(Object) AndAlso
        Parameters(1).ParameterType.IsType(Of EventArgs)()
End Function
```

L'astuce pour les évènements *custom* est d'utiliser la signature de la méthode **Invoke** du délégué correspondant au récepteur.
Cette astuce provient de MSDN.

Supprimé : C

Le code des médiateurs

```
Public Class MethodMediator(Of TEventArgs As EventArgs)
    Implements IMethodMediator

    Public Sub [AddHandler]() Implements IEventMediator.AddHandler
        If Me._Handler IsNot Nothing Then Exit Sub
        Me._Handler = Me.CreateHandler
        Me._Event.AddEventHandler(Me._Handler)
    End Sub
    Public Sub [RemoveHandler]() Implements
IEventMediator.RemoveHandler
        If Me._Handler Is Nothing Then Exit Sub
        Me._Event.RemoveEventHandler(Me._Handler)
        Me._Handler = Nothing
    End Sub

    Protected Overridable Function CreateHandler() As [Delegate]
        If Me.IsSynchronized Then
            Return New EventHandler(Of TEventArgs)(AddressOf
Me.OnSynchronizedEvent)
        ElseIf Me.IsAsynchronous Then
            Return New EventHandler(Of TEventArgs)(AddressOf
Me.OnAsynchronousEvent)
        Else
            Return New EventHandler(Of TEventArgs)(AddressOf
Me.OnSynchronousEvent)
        End If
    End Function
    Protected Function IsSynchronized() As Boolean
        Return Me._Synchronizer IsNot Nothing
    End Function
    Protected Function IsAsynchronous() As Boolean
        Return Me._Asynchronous
```

```
    End Function

    Protected Sub OnSynchronousEvent(ByVal sender As Object, ByVal e
As TEventArgs)
        Dim Arguments = Me.GetMethodArguments(sender, e)
        Me._Method.Invoke(Me._MethodSource, Arguments)
    End Sub
    Protected Sub OnAsynchronousEvent(ByVal sender As Object, ByVal
e As TEventArgs)
        Static Invoker As New EventHandler(Of TEventArgs)(AddressOf
Me.OnSynchronousEvent)
        Invoker.BeginInvoke(sender, e, Nothing, Nothing)
    End Sub
    Protected Sub OnSynchronizedEvent(ByVal sender As Object, ByVal
e As TEventArgs)
        If Me._Synchronizer.InvokeRequired Then
            Static Invoker As New EventHandler(Of
TEventArgs)(AddressOf Me.OnSynchronousEvent)
            Me._Synchronizer.Invoke(Invoker, New Object() {sender,
e})
        Else
            Me.OnSynchronousEvent(sender, e)
        End If
    End Sub
    Private Function GetMethodArguments(ByVal sender As Object,
ByVal e As TEventArgs) As Object()
        If Not Me._Method.HasParameters Then Return Nothing
        Dim Parameters = Me._Method.GetParameters
        Select Case Parameters.Count
            Case 1
                Dim Parameter = Parameters(0)
                Dim Argument = Parameter.GetArgumentValue(sender, e)
                Return New Object() {Argument}
            Case 2
                Dim Argument1 =
Parameters(0).GetArgumentValue(sender, e)
                Dim Argument2 =
Parameters(1).GetArgumentValue(sender, e)
                Return New Object() {Argument1, Argument2}
            Case Else
                Throw New NotSupportedException
        End Select
    End Function

End Class
```

Le principe est le suivant :
chaque mode de médiation correspond à une méthode ;
ces méthodes utilisent l'argument générique du médiateur pour avoir la signature adéquate.

- La synchronisation est fournie à l'aide de l'interface ISynchronizeInvoke.
- La méthode asynchrone utilise la méthode **BeginInvoke** de l'objet EventHandler(Of TEventArgs).
- La méthode synchrone utilise la méthode **Invoke** de MethodInfo


Les méthodes synchronisée et asynchrone ne font qu'un rappel sur la méthode synchrone en utilisant les moyens décrits ci-dessus.

Une méthode de fabrique a le rôle de créer le bon type de délégué et de choisir la méthode à utiliser en fonction du mode de médiation.

Les médiateurs pour les évènements standard et custom spécialisent cette classe via la méthode de fabrique :

Supprimé : S

Supprimé : s

Supprimé : Custom

```
Public Class MethodMediator
    Inherits MethodMediator(Of EventArgs)

    Protected Overrides Function CreateHandler() As System.Delegate
        If Me.IsSynchronized Then
            Return New EventHandler(AddressOf
Me.OnSynchronizedEvent)
        ElseIf Me.IsAsynchronous Then
            Return New EventHandler(AddressOf
Me.OnAsynchronousEvent)
        Else
            Return New EventHandler(AddressOf Me.OnSynchronousEvent)
        End If
    End Function

End Class

Public Class CustomMethodMediator(Of TEventArgs As EventArgs)
    Inherits MethodMediator(Of TEventArgs)

    Private Function GetHandlerMethod() As MethodInfo
        Dim Flags = Reflection.BindingFlags.Instance Or
Reflection.BindingFlags.NonPublic
        If Me.IsSynchronized Then
            Return Me.GetType.GetMethod("OnSynchronizedEvent",
Flags)
        ElseIf Me.IsAsynchronous Then
            Return Me.GetType.GetMethod("OnAsynchronousEvent",
Flags)
        Else
            Return Me.GetType.GetMethod("OnSynchronousEvent", Flags)
        End If
    End Function
    Protected Overrides Function CreateHandler() As [Delegate]
        Dim HandlerMethod = Me.GetHandlerMethod
        Return [Delegate].CreateDelegate(Me.EventType, Me,
HandlerMethod)
    End Function

End Class
```

Certains diront que j'utilise la réflexion, que c'est lent, etc., etc.
Je précise tout de même que les parties les plus lentes ne sont utilisées que lors de la création des délégués des récepteurs.
Le code exécuté à chaque évènement est le code de sélection des arguments et l'appel de la méthode (par réflexion).

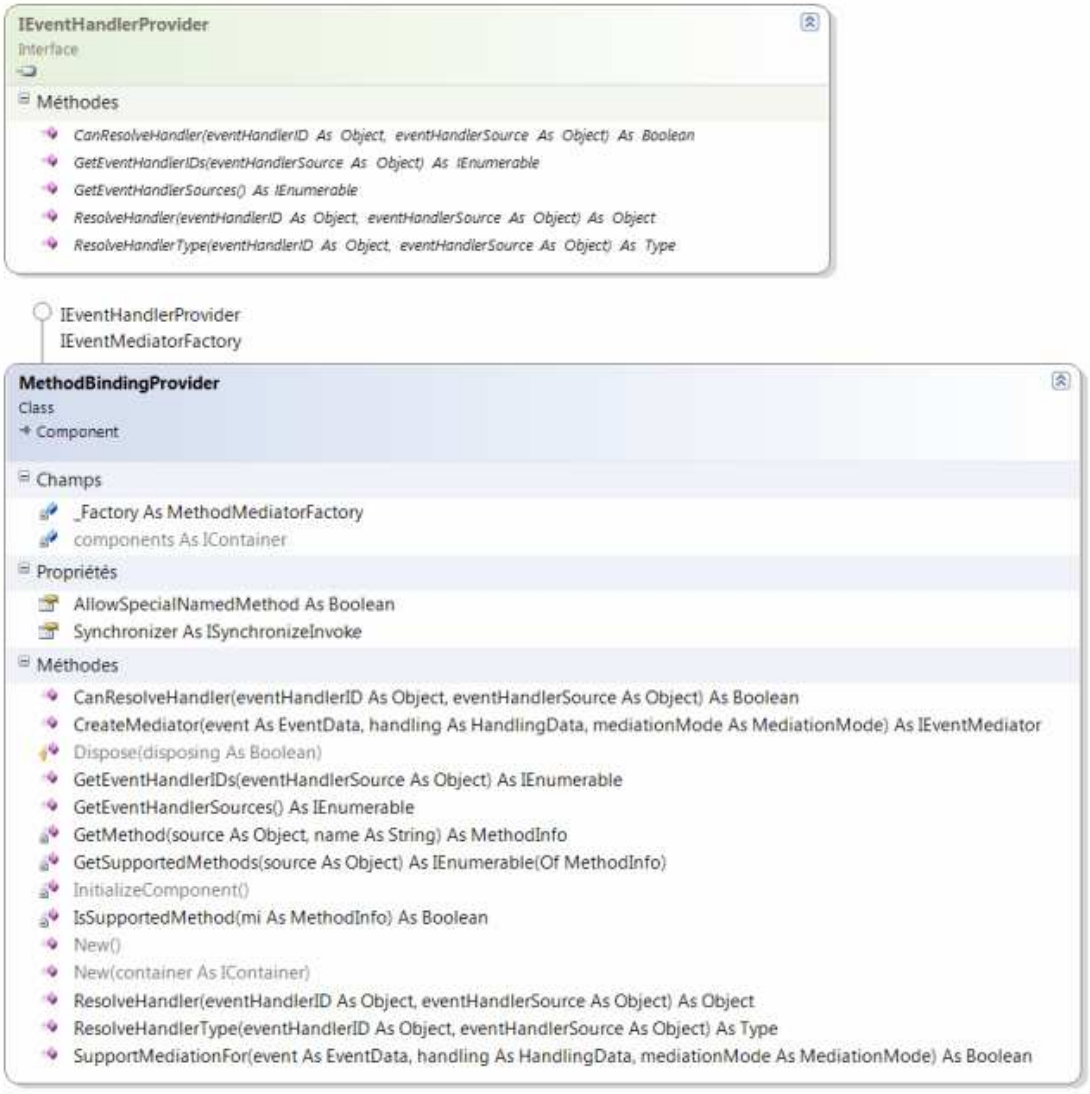
Dans les cas bien précis qui demandent une intention particulière sur les performances, rien n'empêche de gérer les évènements par le code sans passer par la liaison des évènements.

24 III-C. Les récepteurs

La gestion des récepteurs dans cette démonstration se fait par le composant *MethodBindingProvider*.

Supprimé : ce

Ce composant implémente l'interface *IEventHandlerProvider* et encapsule la fabrique de médiateur précédente.



Cliquer sur l'image pour l'agrandir.

Son code est le suivant :

```
Public Class MethodBindingProvider
    Implements IEventHandlerProvider
    Implements IEventMediatorFactory
```

```
    Private _Factory As New MethodMediatorFactory

    Public Property AllowSpecialNamedMethod As Boolean = True
    Public Property Synchronizer As ISynchronizeInvoke
        Get
            Return Me._Factory.Synchronizer
        End Get
        Set(ByVal value As ISynchronizeInvoke)
            Me._Factory.Synchronizer = value
        End Set
    End Property

    Public Function CanResolveHandler(ByVal eventHandlerID As
Object,
        ByVal eventHandlerSource As Object) As Boolean
        Implements IEventHandlerProvider.CanResolveHandler

        If eventHandlerSource Is Nothing Then Return False
        If Not TypeOf eventHandlerID Is String Then Return False
        Dim SupportedMethods =
Me.GetSupportedMethods(eventHandlerSource)
        Dim Name = DirectCast(eventHandlerID, String)
        Return SupportedMethods.Contains(Function(M)
                                             Return
String.Equals(M.Name, Name,

StringComparison.InvariantCultureIgnoreCase)
                                         End Function)
    End Function
    Public Function GetEventHandlerIDs(ByVal eventHandlerSource As
Object) As IEnumerable
        Implements IEventHandlerProvider.GetEventHandlerIDs

        If eventHandlerSource Is Nothing Then Return Nothing
        Dim SupportedMethods =
Me.GetSupportedMethods(eventHandlerSource)
        Dim Names As New Specialized.StringCollection
        For Each method In SupportedMethods
            Names.Add(method.Name)
        Next
        Return Names
    End Function

    Public Function GetEventHandlerSources() As IEnumerable
        Implements IEventHandlerProvider.GetEventHandlerSources

        If Me.Site Is Nothing Then Return Nothing
        Dim Host = Me.Site.GetService(Of IDesignerHost)()
        If Host Is Nothing Then Return Nothing
        Return Host.Container.Components
    End Function

    Public Function ResolveHandler(ByVal eventHandlerID As Object,
        ByVal eventHandlerSource As Object) As Object
        Implements IEventHandlerProvider.ResolveHandler

        If Not TypeOf eventHandlerID Is String Then Return False
        Return Me.GetMethod(eventHandlerSource,
DirectCast(eventHandlerID, String))
    End Function
```

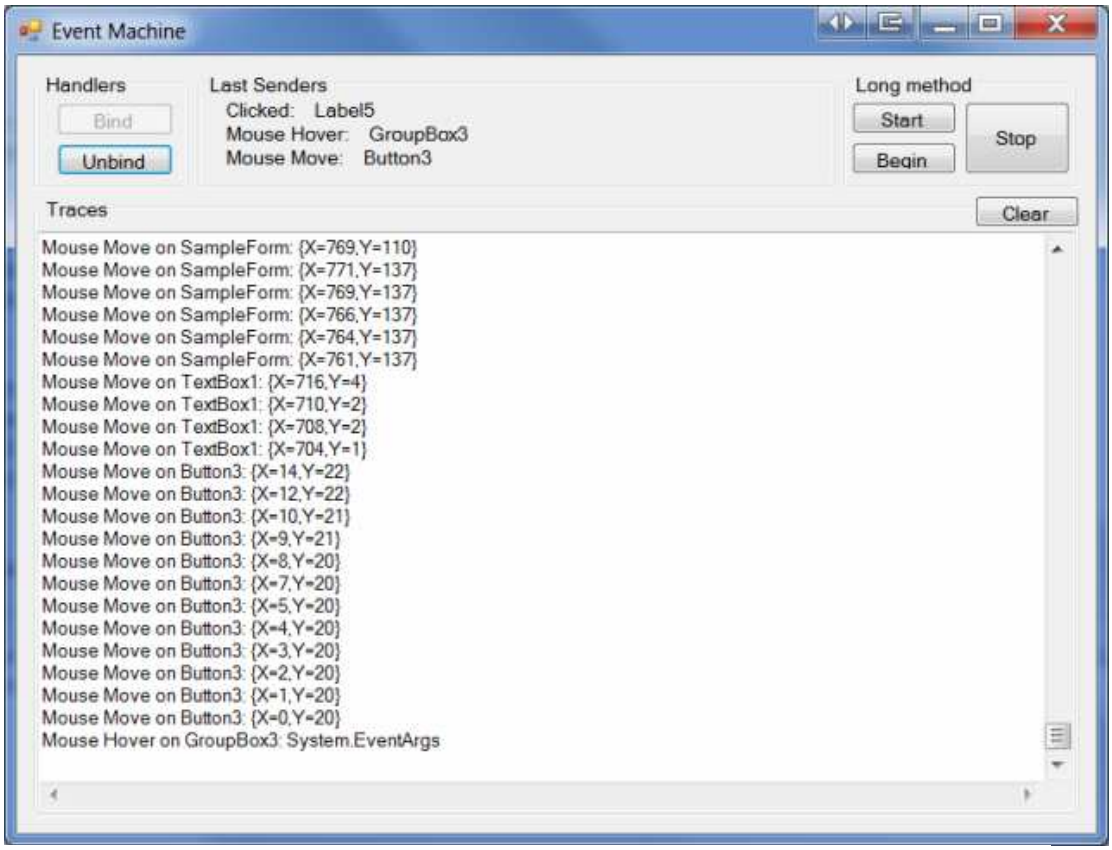
```
    Public Function ResolveHandlerType(ByVal eventHandlerID As
Object,
        ByVal eventHandlerSource As Object) As Type
        Implements IEventHandlerProvider.ResolveHandlerType

        If Not TypeOf eventHandlerID Is String Then Return Nothing
        Dim Handler = Me.GetMethod(eventHandlerSource,
DirectCast(eventHandlerID, String))
        If Handler Is Nothing Then Return Nothing
        Return Handler.GetType
    End Function

    Private Function GetSupportedMethods(ByVal source As Object) As
IEnumerable(Of MethodInfo)
        Dim SupportedMethods As New List(Of MethodInfo)
        Dim Names As New Specialized.StringCollection
        Dim Methods = source.GetType.GetMethods
        For Each method In Methods
            If Names.Contains(method.Name) Then Continue For
            If method.IsAbstract Then Continue For
            If method.IsConstructor Then Continue For
            If method.IsGenericMethodDefinition Then Continue For
            If method.ContainsGenericParameters Then Continue For
            If Not Me.AllowSpecialNamedMethod AndAlso
method.IsSpecialName Then Continue For
            If Not Me.IsSupportedMethod(method) Then Continue For
            SupportedMethods.Add(method)
            Names.Add(method.Name)
        Next
        Return SupportedMethods
    End Function
    Private Function IsSupportedMethod(ByVal mi As MethodInfo) As
Boolean
        Dim Parameters = mi.GetParameters
        If Parameters.IsNullOrEmpty Then Return True
        Return Parameters.Count <= 2
    End Function
    Private Function GetMethod(ByVal source As Object, ByVal name As
String) As MethodInfo
        If source Is Nothing Then Return Nothing
        Dim SupportedMethods = Me.GetSupportedMethods(source)
        Return SupportedMethods.Take(Function(M)
                                         Return
String.Equals(M.Name, name,

StringComparison.InvariantCultureIgnoreCase)
                                     End Function)
    End Function

    Public Function SupportMediationFor(ByVal [event] As EventData,
                                        ByVal handling As
HandlingData,
                                        ByVal mediationMode As
MediationMode) As Boolean
                                        Implements
IEventMediatorFactory.SupportMediationFor

        Return Me._Factory.SupportMediationFor([event], handling,
mediationMode)
    End Function
    Public Function CreateMediator(ByVal [event] As EventData,
```

```
                                    ByVal handling As HandlingData,
                                    ByVal mediationMode As
MediationMode) As IEventMediator
                                    Implements
IEventMediatorFactory.CreateMediator

        Return Me._Factory.CreateMediator([event], handling,
mediationMode)
    End Function

End Class
```

L'abstraction des récepteurs par des identifiants se fait en utilisant les noms des méthodes, les récepteurs sont des MethodInfo.
La propriété *AllowSpecialNamedMethod* permet de spécifier si oui ou non on souhaite utiliser les méthodes ayant un nom spécial (comme les accesseurs Get/Set d'une propriété).

Les surcharges de méthodes ne sont pas supportées, et dans le cas d'une surcharge, seule la première méthode compatible trouvée via la réflexion sera disponible.

25 III-D. L'application de démonstration



Cliquer sur l'image pour l'agrandir.

L'application de démonstration trace tous les évènements Click, MouseMove et MouseHover des contrôles présents dans l'écran ci-dessus.

Une source de données est utilisée pour fournir les différentes propriétés nécessaires au fonctionnement de la démonstration, ainsi que les méthodes des traitements et des traces.

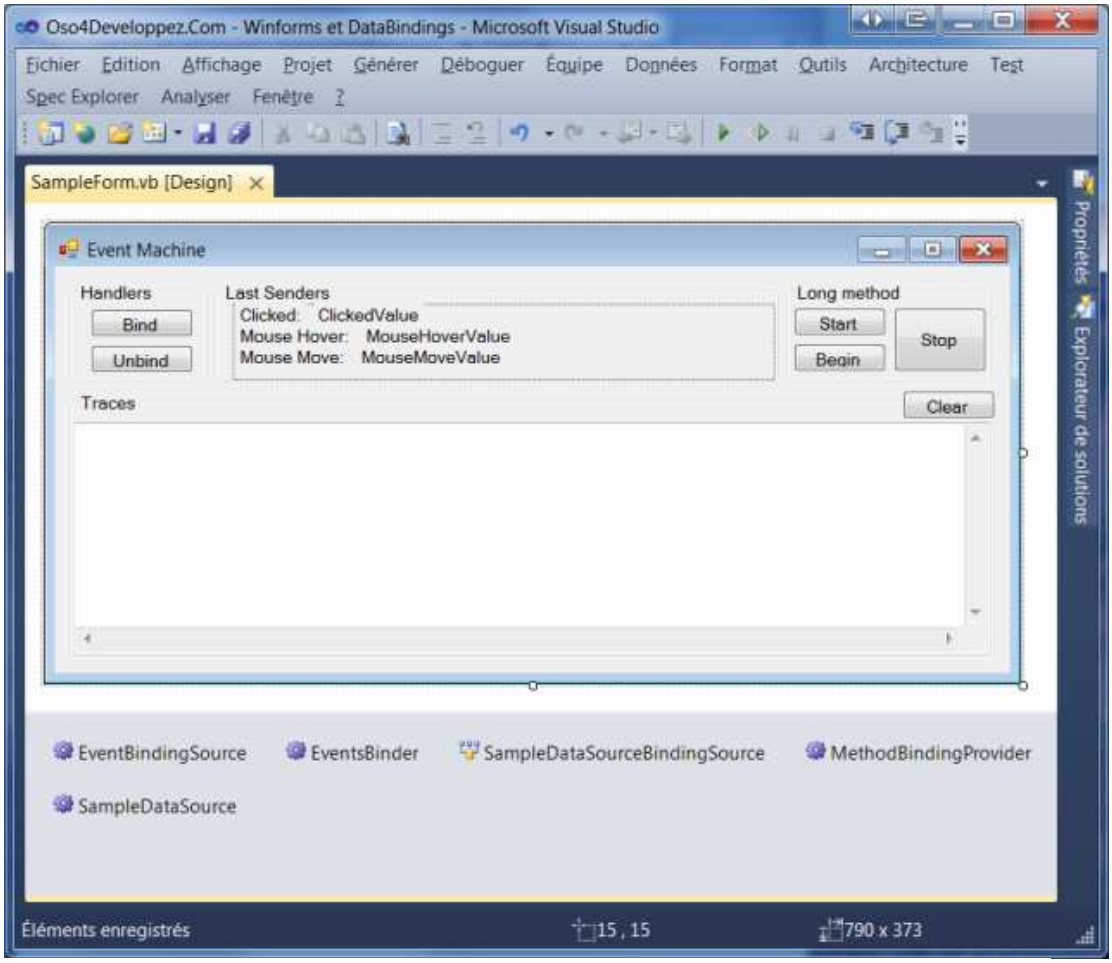
Le code de l'écran est le suivant :

```
Public Class SampleForm

    Public Sub New()

        ' Cet appel est requis par le concepteur.
        InitializeComponent()

        ' Ajoutez une initialisation quelconque après l'appel
InitializeComponent().
        If Me.DesignMode Then Exit Sub
        Me.SampleDataSource.Owner = Me
        Me.SampleDataSourceBindingSource.DataSource =
Me.SampleDataSource
    End Sub

    Public Sub SetText(ByVal value As String)
        If Me.InvokeRequired Then
            Static Del As New Action(Of String)(AddressOf SetText)
            Me.Invoke(Del, New Object() {value})
            Exit Sub
        End If
        Me.Text = value
    End Sub

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Button1.Click
        Me.SampleDataSource.BindHandlers()
    End Sub

End Class
```

Le bouton *Bind Handlers* est relié classiquement par le code pour des raisons évidentes.

Tout le reste se fait par liaison de propriété ou par liaison d'évènement.

En *Design Time*, nous obtenons :



Cliquer sur l'image pour l'agrandir.

Le code des liaisons est le suivant :

```
'
'EventBindingSource
'
Me.EventBindingSource.Add("MouseHover", Me.GroupBox3,
"TraceMouseHover",
    Me.MethodBindingProvider, Me.SampleDataSource,
Me.MethodBindingProvider,
    EventBindings.MediationMode.Synchronous)
Me.EventBindingSource.Add("MouseHover", Me, "TraceMouseHover",
    Me.MethodBindingProvider, Me.SampleDataSource,
Me.MethodBindingProvider,
    EventBindings.MediationMode.Synchronous)
Me.EventBindingSource.Add("MouseMove", Me, "TraceMouseMove",
    Me.MethodBindingProvider, Me.SampleDataSource,
Me.MethodBindingProvider,
    EventBindings.MediationMode.Synchronous)
Me.EventBindingSource.Add("Click", Me, "TraceClick",
    Me.MethodBindingProvider, Me.SampleDataSource,
Me.MethodBindingProvider,
```

```
    EventBindings.MediationMode.Synchronous)
Me.EventBindingSource.Add("Click", Me.TextBox1, "TraceClick",
    Me.MethodBindingProvider, Me.SampleDataSource,
Me.MethodBindingProvider,
    EventBindings.MediationMode.Synchronous)
Me.EventBindingSource.Add("MouseMove", Me.TextBox1,
"TraceMouseMove",
    Me.MethodBindingProvider, Me.SampleDataSource,
Me.MethodBindingProvider,
    EventBindings.MediationMode.Synchronous)
Me.EventBindingSource.Add("MouseHover", Me.TextBox1,
"TraceMouseHover",
    Me.MethodBindingProvider, Me.SampleDataSource,
Me.MethodBindingProvider,
    EventBindings.MediationMode.Synchronous)
```

 Je ne représente ici que les premières, il y en a une trentaine en tout.

Source de données utilisée par l'exemple

```
<PropertyTab(GetType(Design.EventBindingsPropertyTab),
PropertyTabScope.Document)>
Public Class SampleDataSource
    Implements INotifyPropertyChanged

    Public Event PropertyChanged(ByVal sender As Object, ByVal e As
PropertyChangedEventArgs)
        Implements INotifyPropertyChanged.PropertyChanged

    Private _LastClickedSender As Object
    Private _LastMouseHoverSender As Object
    Private _LastMouseMoveSender As Object

    Public Property Owner As SampleForm
    Public Property Binder As IEventBinder
    Public Property TraceTextBox As TextBox
    Public Property LongMethodIteration As Int32 = 50000
    Public Property StopLongMethod As Boolean
    Public ReadOnly Property CanBindHandlers As Boolean
        Get
            Return Not Me._Binder.Handled
        End Get
    End Property
    Public ReadOnly Property CanUnbindHandlers As Boolean
        Get
            Return Me._Binder.Handled
        End Get
    End Property
    Public ReadOnly Property LastClickedSender As String
        Get
            Return Me.TryGetName(Me._LastClickedSender)
        End Get
    End Property
    Public ReadOnly Property LastMouseHoverSender As String
        Get
            Return Me.TryGetName(Me._LastMouseHoverSender)
        End Get
    End Property
    Public ReadOnly Property LastMouseMoveSender As String
        Get
```

```
            Return Me.TryGetName(Me._LastMouseMoveSender)
        End Get
    End Property

    Private Function TryGetName(ByVal sender As Object) As String
        If TypeOf sender Is Control Then
            Return DirectCast(sender, Control).Name
        End If
        Return String.Empty
    End Function
    Private Sub NotifyPropertyChanged(ByVal propertyName As String)
        RaiseEvent PropertyChanged(Me, New
PropertyChangedEventArgs(propertyName))
    End Sub

    Public Sub ClearTraces()
        Me._TraceTextBox.Clear()
    End Sub
    Public Sub BindHandlers()
        Me._Binder.AddHandlers()
        Me.NotifyPropertyChanged("CanBindHandlers")
        Me.NotifyPropertyChanged("CanUnbindHandlers")
    End Sub
    Public Sub UnbindHandlers()
        Me._Binder.RemoveHandlers()
        Me.NotifyPropertyChanged("CanBindHandlers")
        Me.NotifyPropertyChanged("CanUnbindHandlers")
    End Sub

    Private Sub AddTrace(ByVal eventName As String, ByVal senderName
As String, ByVal args As String)
        If Not String.IsNullOrEmpty(senderName) Then
            Me.AddTrace(String.Format("{0} on {1}: {2}", eventName,
senderName, args))
        Else
            Me.AddTrace(String.Format("{0}: {2}", eventName, args))
        End If
    End Sub
    Public Sub AddTrace(ByVal value As String)
        Me._TraceTextBox.AppendText(value + ControlChars.CrLf)
    End Sub
    Public Sub TraceClick(ByVal sender As Object)
        Me.AddTrace("Click", Me.TryGetName(sender), "")
        Me._LastClickedSender = sender
        Me.NotifyPropertyChanged("LastClickedSender")
    End Sub
    Public Sub TraceMouseHover(ByVal sender As Object, ByVal e As
EventArgs)
        Me.AddTrace("Mouse Hover", Me.TryGetName(sender),
e.ToString)
        Me._LastMouseHoverSender = sender
        Me.NotifyPropertyChanged("LastMouseHoverSender")
    End Sub
    Public Sub TraceMouseMove(ByVal e As MouseEventArgs, ByVal
sender As Object)
        Me.AddTrace("Mouse Move", Me.TryGetName(sender),
e.Location.ToString)
        Me._LastMouseMoveSender = sender
        Me.NotifyPropertyChanged("LastMouseMoveSender")
    End Sub
```

```
    Public Sub InterromptLongMethod()
        Me.StopLongMethod = True
    End Sub
    Public Sub LongMethod()
        Me.StopLongMethod = False
        If Me.LongMethodIteration < 0 Then Me.LongMethodIteration =
-Me.LongMethodIteration
        For i = 0 To Me.LongMethodIteration
            Me._Owner.SetText(
                String.Format("Commande longue, étape: {0}/{1}", i,
Me.LongMethodIteration))
            If Me.StopLongMethod Then
                Me._Owner.SetText("Stopped")
                Exit For
            End If
        Next
        Me.StopLongMethod = True
    End Sub
End Class
```

La source de données fournit l'onglet des liaisons d'évènements, les différentes méthodes de gestion des traces, ainsi que les propriétés et méthodes utilisées par la démonstration.

On observera que les méthodes reliées aux évènements permettent de tester les différents cas de figure :

- méthode sans argument ;
- méthode aux arguments inversés ;
- méthode aux arguments incomplets ;
- et bien évidemment, méthode aux arguments correspondants.

On observera aussi, que la seule intrusion du moteur de liaison dans la source de données est la déclaration de l'onglet de liaison des évènements par l'attribut PropertyTab.

26 V. Conclusion

Nous venons de montrer dans l'étude de la démonstration que le côté non intrusif du moteur de liaison des évènements est réel. Nous devons implémenter des interfaces spécifiques du moteur de liaison que si nous avons besoin d'ajouter de nouvelles fonctionnalités ou de modifier les fonctionnalités proposées par défaut.

27 IV-A. Qu'en est-il des autres fonctionnalités prévues au début de cet article ?

Attacher plusieurs récepteurs sur un seul évènement ?
Cette fonctionnalité était prévue dès le début de la conception du moteur de liaison.

Aucun problème quant à sa réalisation et à son utilisation, que ce soient des liaisons multiples ou une liaison et du code, etc., etc.
Toutefois, comme tout évènement Multicast, nous n'avons aucune garantie sur l'ordre d'exécution des récepteurs.

Supprimé : quand

Les liaisons multiples par erreur ?
Le moteur de liaison empêche toute liaison multiple d'une liaison sur un évènement.

Les liaisons multiples volontaires ?
Là, c'est plus compliqué, il faut faire une nouvelle liaison et l'attacher à une autre source de liaison que la liaison existante. On est clairement dans un manque du moteur de liaison, mais vu l'intérêt de la fonctionnalité, je ne trouve pas çà gênant, bien au contraire.

Les liaisons multiples correspondent à un même récepteur relié plusieurs fois sur un même évènement.
À ne pas confondre avec plusieurs liaisons sur un seul évènement (avec donc plusieurs récepteurs).

Les signatures et la récupération des valeurs ?
Le respect obligatoire des signatures entre les évènements et leurs récepteurs est brisé. Dans le cas où les arguments de ces signatures sont compatibles, on récupère les valeurs passées par l'évènement.
Succès complet donc, surtout que cette capacité dépend directement de l'implémentation de la médiation, et est donc au choix de l'utilisateur du moteur de liaison.

Gestion synchrone, asynchrone et synchronisée ?
Dans la démonstration, tous les modes de médiation sont gérés. De façon basique, certes, mais ils sont gérés.
Liberté totale à ce niveau pour l'utilisateur du moteur de liaison. Il peut même ne pas s'en servir, s'il n'en a pas besoin.

Gestion de la durée de vie et de la mémoire ?
La démonstration ne permet pas de tester ces fonctionnalités.
Toutefois, le moteur de liaison permet une gestion fine de la durée de vie des récepteurs, des médiateurs et des liaisons. Il permet donc une gestion fine de la mémoire.
La réponse est donc : cela dépend entièrement de l'utilisation et de l'implémentation des différents services.

Facilité d'utilisation ?
L'interface en *Design Time* peut certainement être améliorée. Notamment en remplissant automatiquement les choix lorsque l'on ne dispose que d'un seul choix ou en supportant la sélection multiple des propriétés.
Il peut aussi être simplifié, en retirant l'édition à tous les niveaux hiérarchiques. Pour faire la démonstration, je n'ai utilisé que le premier niveau d'édition.
A voir à l'usage.

28 IV-B. Finalité et intérêt

Tous les points fonctionnels ont été abordés, je ne vois pas de blocage empêchant l'utilisation d'un tel moteur de liaison des évènements. Surtout que celui-ci est entièrement compatible avec la gestion des évènements habituelle par le code. Il est

aussi entièrement compatible avec la liaison des données fournies par les Windows Forms.

Supprimé : fournie

Les deux utilisés ensemble réduisent fortement les dépendances entre les modèles de données et leurs représentations graphiques.

Cela peut-être utile pour toute personne désirant s'orienter vers des patrons de conception comme le MVC, le MVP ou le plus récent MVVM de WPF. Ou encore, plus simplement, pour les personnes comme moi, qui conçoivent ou qui souhaitent concevoir leur GUI à base de liaisons.

Supprimé : é

Je ne sais pas si vous trouvez un intérêt à un tel moteur de liaison des évènements, mais en tout cas, j'espère qu'en lisant cet article, vous saurez en créer un, si nécessaire.

À bientôt pour la suite de la série "Windows Forms : De la liaison de données à la liaison d'objets".
Je vous laisse digérer cet article très conséquent. (mets plutôt « dense » : conséquent est, à mon avis, impropre, même s'il est souvent employé dans le sens « important, volumineux, que tu lui donnes ici)

À vous les studios... de développement.

Sources Les sources de la preuve de faisabilité.
1 commentaire