Bonnes pratiques pour coder des activités avec Windows Workflow Foundation 4

Article par Jérémy Jeanson (MVP Connected System Developer)

Je suis actuellement consultant chez Peaks (société de conseil en informatique basée sur Paris, Reims et Lyon). Développant avec .net depuis 2001, j'ai vécu les différentes évolutions du Framework, des premières beta de la version 1.0 à aujourd'hui. Je surfe donc sur la vague .net depuis ses débuts, et j'adore cela.

Spécialisé dans les architectures n-tiers exploitant Windows Communication Foundation, Windows Workflow Foundation et l'interopérabilité, j'interviens régulièrement comme consultant / formateur pour des équipes désireuses d'intégrer les dernières nouveautés en la matière.

Pourquoi doit-on se soucier de la manière dont on code une activité ?

Quand on s'intéresse aux bonnes pratiques d'un langage, c'est bien souvent afin d'être plus performant : exécution plus rapide, et consommation des ressources réduite.

Dans le monde Windows Workflows Foundation 4, c'est un peu différent. On cherche surtout à éviter les pièges qui rendent l'emploi de WF4 plus lent qu'une méthode classique ou les développeurs moins productifs.

Pourquoi une telle différence ?

La principale raison est toute simple : on amalgame bien trop souvent WF4 à un langage alors qu'il s'agit d'un runtime étendant les possibilités de .net et lui offrant des fonctionnalités de workflows … ou l'inverse : offre les possibilités de .net au monde du workflow.

Un workflow ne doit donc pas être pris pour une méthode ou une fonction comme en C# ou en Visual Basic. Votre workflow a une vie qui dépasse le simple « run ». Il peut être démarré, mis en pause, arrêté, annulé… etc…

Quand on code une activité, il faut donc prendre en compte tout cela.

S'ajoute à ceci une difficulté particulière : vous ne codez pas vos activités pour vous-même. En général, la personne qui code des activités (le développeur) le fait pour un autre qui va construire les workflows (le designer). Ne sachant pas ce que le designer a vraiment en tête quand il vous demande une activité, vous ne pouvez donc pas vous dire que vous pouvez faire l'impasse sur certain états.

Le comportement de vos activités doit donc être entièrement maitrisé si on invoque l'une ou l'autre des méthodes suivantes sur l'hôte de workflows :

- Run
- Persist
- Unload
- Load
- ResumeBookmark
- Terminate
- Cancel

L'activité à tout faire.

Première règle en la matière pour bien coder une activité, il ne fait pas cherche à faire de son activité le « couteau suisse » du workflow. Gardez en tête une équation simple :

> Une activité = Une action.

Si votre activité est amenée à faire trop de choses, elle sera plus difficile à utiliser pour le designer.

Par la suite votre activité serra plus difficile à maintenir et ses évolutions prendront plus de temps à être codées. Pire encore si votre activité a besoin d'un peu de refactoring, elle posera de gros soucis

quand il faudra mettre à jour les workflows l'utilisant (WF4 n'aimant pas beaucoup de le refactoring dans sa version actuelle).

Composer plutôt que d'hériter.

Avec WF4 Il faut toujours privilégier la composition à l'héritage. C'est ce que l'on appelle communément « créer des activités composites ». En codant une activité qui utilise des activités existantes :

- On gagne du temps.
- On peut utiliser des activités qui ne font pas partie de la Toolbox de Visual Studio mais qui sont présentes dans WF4.
- On diminue la complexité de son code.
- On profite des évolutions des activités de base lorsqu'elles sont mises à jour par Microsoft.

De plus on fait la part belle à une méthode qui rompt totalement avec la manière classique de coder une méthode C#/Vb car on fournit l'implémentation d'un petit workflow.

Par exemple, voici l'implémentation d'une activité chargée de faire l'incrément d'une variable à la manière d'un i++ (i étant représenté par l'argument **To**)

```
using System;
using System.Activities;
using System.Activities.Expressions;

public class Increment<T> : Activity
{

    // Define an activity input argument of type T
    [RequiredArgument]
    public InOutArgument<T> To { get; set; }

    // Implementation cache
    private readonly Func<Activity> m_Implementation;

    /// <summary>
    /// New
    /// </summary>
    public Increment()
    {
        this.m_Implementation = new Func<Activity>(this.GetImplementation);
    }

    /// <summary>
    /// Implementation
    /// </summary>
    protected override Func<Activity> Implementation
    {
        get { return this.m_Implementation; }
        set { }
    }

    /// <summary>
    /// Get Implementation
    /// </summary>
```

```
    /// <returns></returns>
    private Activity GetImplementation()
    {
        return new Add<T, T, T>
        {
            Left = new InArgument<T>(c => this.To.Get(c)),
            Right = (T)Convert.ChangeType(1, typeof(T)),
            Result = new OutArgument<T>(c => this.To.Get(c))
        };
    }

    /// <summary>
    /// Register activity's metadata
    /// </summary>
    /// <param name="metadata"></param>
    protected override void CacheMetadata(ActivityMetadata metadata)
    {
        // Register In arguments
        RuntimeArgument arg = new RuntimeArgument("To", typeof(T),
ArgumentDirection.InOut);
        metadata.AddArgument(arg);
        metadata.Bind(this.To, arg);
        // [To] Argument must be set
        if (this.To == null)
        {
            metadata.AddValidationError(
                new System.Activities.Validation.ValidationError(
                    "[To] argument must be set!",
                    false,
                    "To"));
        }
    }
}
```

Dans cet exemple, trois éléments sont importants :

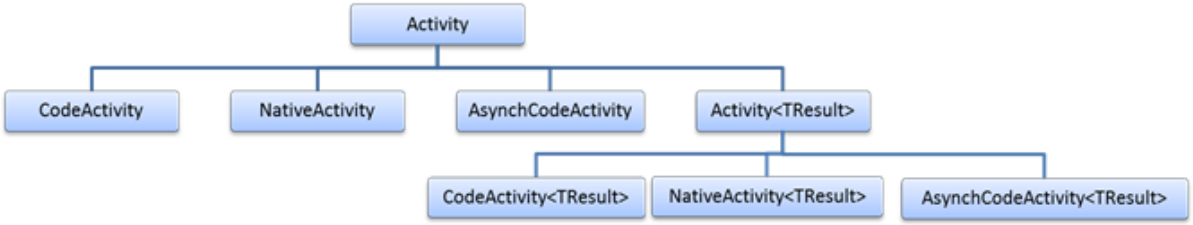
- Le constructeur **Increment()**
- La méthode qui crée l'implémentation **GetImplementation()**
- La propriété qui retourne l'implémentation de notre activité **`Implementation`**

Et on utilise une activité **Add** qui n'est pas exploitable autrement.

En respectant cette structure, vous disposez d'un canevas simple pour la création d'activités composites. Vous savez aussi que toute modification de votre activité passera par la modification d'une seule méthode : **GetImplementation()**.

Choisir la classe de base de son activité

Avant de dé coder une quelconque activité, il faut connaitre les classes pouvant servir de base à celle-ci. Ces classes et la manière dont elles héritent les unes des autres sont représentées dans le graphique suivant.



On distingue clairement ici deux branches : Activity et Activity<TResult>. La première branche (CodeActivity, NativeActivity, AsyncCodeActivity) sert pour les activités ne retournant pas de résultat (un peu comme un void C# ou Sub Vb). Et donc en toute logique la seconde branche est destinée à être employée dans des activités retournant un résultat (Func<TResult>).

Dans chaque branche, chaque activité a un rôle clairement établi :

- CodeActivity : la plus basique – elle sert à exécuter une action, mais ne peut pas servir à interagir avec des activités qui lui seraient internes (donc cette activité ne peut pas servir de base à une séquence custom).
- NativeActivity : fait justement ce que la CodeActivity ne sait pas faire : elle peut contenir des activités et déclencher leur exécution.
- AsyncCodeActivity : Comme son nom l'indique, elle est très pratique pour les opérations asynchrones. Ou opérations lourdes pouvant greffer les performances.

Voici une implémentation de chaque classe qui devrait vous permettre d'en comprendre les subtilités :

1) CodeActivity sans résultat:

A peu de chose près il s'agit d'une simple méthode Execute(), rien de particulier. Voilà pourquoi la CodeActivity est conseillée pour les traitements les plus basiques.

```
public sealed class MyCodeActivity : CodeActivity
{
    protected override void Execute(CodeActivityContext context)
    {
        // Job à exécuter
    }
}
```

2) CodeActivity<TResult> avec résultat

Sur la même base que la CodeActivity. On retourne notre résultat comme si il s'agissait d'une fonction normale.

```
public sealed class MyCodeActivityT : CodeActivity<String>
{
    protected override string Execute(CodeActivityContext context)
```

```
    {
        return "... Retour du job à exécuter";
    }
}
```

3) NativeActivity sans résultat:

Attention au type du contexte, il change !

Ce contexte a quelques méthodes de plus que celui de la CodeActivity. Il permet entre autres d'exécuter des activités. Votre activité pourra donc servir à contenir et déclencher des activités (comme le fait l'activité Sequence).

```
public sealed class MyNativeActivity : NativeActivity
{
    protected override void Execute(NativeActivityContext context)
    {
        // Job à exécuter ...
        // La méthode Shedule du contexte peut déclencher d'autres activités
    }
}
```

4) NativeActivity<TResult> avec résultat:

Comme pour la NativeActivity, le contexte est différent de la CodeActivity<TResult>. La méthode Execute est aussi différente. Le retour est présent sous la forme d'un Argument Result (promis pour ceux qui ne connaissent pas encore, je parlerai d'arguments prochainement).

Pour retourner le résultat de l'activité, il faut passer par la méthode Set() de cet argument et lui passer le contexte courant et votre retour.

```
public sealed class MyNativeActivityT : NativeActivity<String>
{
    protected override void Execute(NativeActivityContext context)
    {
        this.Result.Set(context, "...Retour du job à exécuter");
    }
}
```

5) AsyncCodeActivity sans résultat

Cette fois ci, on a une méthode **BeginExecute()** qui prend un peu plus d'arguments et un contexte d'un nouveau type. Celui-ci ne comporte pas les méthodes nécessaires à l'exécution d'activités enfantes. On reste donc à peu près sur la même base que la CodeActivity; l'aspect asynchrone en plus.

Pour que les choses soient claires, j'ai créé un petit exemple complet mettant en œuvre :

BeginExecute et **EndExecute** du pattern asynchrone.

Job(), qui sera la méthode nécessitant une exécution asynchrone.

BeginExecute() prépare l'opération asynchrone et la lance **Job ()**. A cet instant **context.UserState** sert à stocker le délégué à l'origine de l'appel.

EndExecute() est lancée quand le job est terminé. **context.UserState** permet de retrouver le délégué utilisé et de lui demander le résultat de la méthode **Job()**.

```
public sealed class MyAsyncCodeActivity : AsyncCodeActivity
{
    /// <summary>
    /// Lancement asynchrone de l'exécution du job
    /// </summary>
    /// <param name="contexte"></param>
    /// <param name="callback"></param>
    /// <param name="state"></param>
    /// <returns></returns>
    protected override IAsyncResult BeginExecute(AsyncCodeActivityContext context,
 AsyncCallback callback, object state)
    {
        Action job = new Action(this.Job);
        contexte.UserState = job;
        return job.BeginInvoke(callback, state);
    }
    /// <summary>
    /// Récupération du résultat du traitement effectué dans le job
    /// </summary>
    /// <param name="contexte"></param>
    /// <param name="result"></param>
    protected override void EndExecute(AsyncCodeActivityContext context, IAsyncRes
ult result)
    {
        Action job = context.UserState as Action;
        if (job != null)
        {
            job.EndInvoke(result);
        }
    }


    /// <summary>
    /// Job à réaliser
    /// </summary>
    private void Job()
    {
        // Job à réaliser
    }
}
```

5) AsyncCodeActivity<TResult> avec résultat:

Il s'agit là à peu près de la même chose que pour **AsyncCodeActivity**. Le retour se fait de la même manière que pour un **CodeActivity**, au détail près qu'il a lieu dans la méthode **EndExecute**.

```
public sealed class MyAsyncCodeActivityT : AsyncCodeActivity<Boolean>
{
    /// <summary>
    /// Lancement asynchrone de l'exécution du job
    /// </summary>
    /// <param name="contexte"></param>
```

```
    /// <param name="callback"></param>
    /// <param name="state"></param>
    /// <returns></returns>
    protected override IAsyncResult BeginExecute(AsyncCodeActivityContext context, AsyncCallback callba
ck, object state)
    {
        Func<Boolean> job = new Func<Boolean>(this.Job);
        contexte.UserState = job;
        return job.BeginInvoke(callback, state);
    }
    /// <summary>
    /// Récupération du résultat du traitement effectué dans le job
    /// </summary>
    /// <param name="contexte"></param>
    /// <param name="result"></param>
    protected override Boolean EndExecute(AsyncCodeActivityContext context, IAsyncResult result)
    {
        Func<Boolean> job = context.UserState as Func<Boolean>;
        if (job != null)
        {
            return job.EndInvoke(result);
        }
        return false;
    }


    /// <summary>
    /// Job à réaliser
    /// </summary>
    private Boolean Job()
    {
        // Job à réaliser
        return true;
    }
}
```

Conventions de nommage

Pour ce qui est des conventions de nommage, il n'y a pas grand-chose à redire. On reste sur les grandes lignes qui s'appliqueraient à tout autre code autre que WF :

- Les activités n'ont pas besoin de suffixe (ni de préfixe). On leur donne un nom lié à l'action qu'elles exécuteront. On reste simple.
- Les arguments n'ont pas besoin de suffixes (ni de préfixes). Comme pour une méthode normale.
- Les variables n'ont pas besoin de suffixes (ni de préfixes). Comme pour les arguments.
- Ajouter le suffixe "Scope" aux activités qui ont pour objectif de décorer les activités qu'elles sont amenées à contenir. Par exemple : CorrelationScope, CancellationScope.

Comprendre la méthode Execute… et la respecter

La méthode **Execute()** ne doit pas bloquer le contexte du workflow :

Cette méthode permettant au runtime de programmer l'exécution du workflow, l'hôte l'appelle afin que votre activité puisse utiliser les informations du contexte pour faire de petites opérations ou pour programmer l'exécution d'activités filles.

C'est pourquoi certaines opérations ne doivent pas avoir lieu dans cette méthode :

- Calculs longs.
- Méthodes consommatrices en ressources.
- Activité sur le réseau ou tout autre entrée / sortie avec System.IO.

La bonne pratique pour ce genre d'opération consiste à coder votre activité en héritant d'une AsyncCodeActivity ou AsyncCodeActivity<T>.

Par exemple pour coder une activité qui attend que le réseau soit accessible : Cette attente est une méthode qui est lancée en respectant le pattern asynchrone exploitant un IAsyncResult, on peut donc facilement la décliner.

```
using System;
using System.Activities;
using System.Net.NetworkInformation;
using System.Threading;

public sealed class WaitForNetWork : AsyncCodeActivity<Boolean>
{
    // Handle pour l'attente du réseau
    private AutoResetEvent m_AutoResetEvent;
    /// <summary>
    /// Début d'exécution de l'activité
    /// </summary>
    /// <param name="contexte"></param>
    /// <param name="callback"></param>
    /// <param name="state"></param>
    /// <returns></returns>
    protected override IAsyncResult BeginExecute(AsyncCodeActivityContext context, AsyncCallback
callback, object state)
    {
        Action job = new Action(this.DoWork);
        contexte.UserState = job;
        return job.BeginInvoke(callback, state);
    }

    /// <summary>
    /// Fin d'exécution de l'activité
    /// </summary>
    /// <param name="contexte"></param>
    /// <param name="result"></param>
    /// <returns></returns>
    protected override Boolean EndExecute(AsyncCodeActivityContext context, IAsyncResult result)
    {
        Action job = context.UserState as Action;
        job.EndInvoke(result);
        // Retourner l'état du réseau uniquement si l'activité n'a pas été annulée
        return context.IsCancellationRequested
        ? false
        : NetworkInterface.GetIsNetworkAvailable();
    }

    /// <summary>
    /// Réaction à la demande d'annulation de l'activité
    /// </summary>
    /// <param name="contexte"></param>
    protected override void Cancel(AsyncCodeActivityContext context)
    {
        // Libération du handle si on doit annuler l'activité
        if (this.m_AutoResetEvent != null)
        {
            this.m_AutoResetEvent.Set();
        }
        base.Cancel(context);
    }

    /// <summary>
    /// Travail à executer
    /// </summary>
    private void DoWork()
```

```
    {
        // Teste si on doit vraiment attendre après le réseau
        if (!NetworkInterface.GetIsNetworkAvailable())
        {
            this.m_AutoResetEvent = new AutoResetEvent(false);
            // Abonnement aux changement de disponibilité du réseau
            NetworkChange.NetworkAvailabilityChanged += new
NetworkAvailabilityChangedEventHandler(NetworkChange_NetworkAvailabilityChanged);
            // Attente de changements
            this.m_AutoResetEvent.WaitOne();
        }
    }

    /// <summary>
    /// Changement survenu sur la disponibilité du réseau
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void NetworkChange_NetworkAvailabilityChanged(object sender, NetworkAvailabilityEventArgs
e)
    {
        // Teste si le réseau est disponible
        if (e.IsAvailable)
        {
            this.m_AutoResetEvent.Set();
        }
    }
}
```

Dans cet exemple, en plus d'utiliser un **l'AsyncCodeActivity<T>** comme type de base, on prend en compte l'annulation de l'activité (action Cancel sur l'hôte de workflow ou annulation d'un branche comme ce peut être le cas dans une branche d'une activité Pick).

Si votre activité doit être en mesure d'être interrompue, réécrivez toujours sa méthode Cancel afin de s'assurer d'interrompre proprement tout processus interne à votre activité.

Pour ce qui est des opérations visant à mettre en attente le workflow d'un évènement extérieur intercepté par l'hôte, il faut utiliser les bookmarks. Et surtout modifier votre activité de manière à ce qu'elle indique le fait qu'elle se réserve le droit de mettre en pause le workflow. Ceci ce fait facilement via la propriété CanInduceIdle (elle doit renvoyer True dans ce cas)

```
/// <summary>
/// Cette activité se réserve le droit de mettre en pause le workflow
/// </summary>
protected override bool CanInduceIdle
{
    get
    {
        return true;
    }
}
```

Toujours dans l'idée de respecter la méthode Execute et votre hôte de workflow, si lors d'un abandon par l'hôte de l'instance de workflow en cours (ce qui peut arriver en cas d'erreur) il faut réécrire la méthode Abort afin d'y ajouter le code que vous jugerez utile.

```
/// <summary>
/// Abandon du workflow par l'hôte
/// </summary>
/// <param name="context"></param>
protected override void Abort(NativeActivityAbortContext context)
{
    base.Abort(context);
```

```
}
```

Properties versus Arguments versus Activities

Contrairement à des classes .NET classiques, les propriétés des activités peuvent être de trois types :

- Properties (propriété classique de tout type CLR)
- Arguments (propriété de type Argument<T>)
- Activities (propriété de type Activity)

Il s'agit là d'un aspect qui peut impacter sérieusement nos performances si on n'y prend pas garde : property, argument et activity n'ont pas la même manière de "vivre" au sein de votre activité :

- L'argument n'est évalué qu'une seule fois. Il est donc inutile dans une activité de l'utiliser dans un test pour savoir si son expression a changé.
- L'activité peut être réévaluée à plusieurs reprises dans votre activité personnalisée, car on peut programmer son exécution via le contexte autant de fois qu'on le souhaite.
- La propriété type CLR (celle que vous écrivez tous les jours dans vos classes). Son utilisation est à envisager dans le sens "paramètre fixé pour un workflow". Toutes vos instances du même workflow auront toujours la même information.

Par exemple, si vous deviez coder une activité de type While dans laquelle une propriété fait office de condition régissant la continuation ou non de la boucle, elle doit être de type Activity. Ceci, car, à chaque passage, la condition doit être réévaluée.

Si vous utilisiez une propriété de type Argument<T>, elle ne serait pas réévaluée à chaque passage. Votre activité While bouclerait sans fin si l'évaluation de votre Argument<T> retournait True, ou ne bouclerait jamais si elle retournait False.

Body versus Activities

À force d'utiliser les activités de base de Windows Workflow Foundation, vous avez peut-être constaté qu'elles utilisaient couramment des noms tel que Body ou Children pour désigner leurs propriétés de type Activity.

Ceci est en fait une pratique de l'équipe Microsoft travaillant sur WF4. Le principe est relativement simple :

- Si notre activité doit programmer l'exécution d'une seule activité, on nommera cette propriété Body. Tout comme Microsoft l'a fait avec les activités While, ForEach…etc..
- Si notre activité doit programmer l'exécution de plusieurs activités (comme une séquence custom) on choira un nom simple et personnalisé autre que Body. Le nom Activities étant quand même conseillé si vous exposez une collection d'activités.

Ce principe de nommage n'est pas en soi une obligation, mais le fait de l'utiliser facilite la vie des développeurs qui emploieront vos activités. C'est bien connu : quand quelque chose ressemble à ce que l'on connait, on s'y adapte plus facilement.

Mais l'utilisation du terme Body, n'est pas anodine. Elle survient comme une piqûre de rappel, pour nous sensibiliser au fait que lorsque l'on code une activité, on doit chercher à faire simple et laisser à l'utilisateur la possibilité de décider.

A partir du moment que votre activité peut contenir une activité, il est tout à fait probable que cette activité contienne une ou plusieurs autres activités… chacun sa responsabilité : ne cherchez donc pas à chaque fois à coder une séquence. Ajoutez une propriété de type Activity nommé Body, et laissez l'utilisateur décider du reste.

On doit chercher en permanence à faire au plus simple. Avec le temps et l'expérience, vous vous rendrez vite compte que l'on code bien plus souvent des activités ayant des propriétés nommées Body que Activities. Donc plus d'activités hébergeant une activité choisie par l'utilisateur qu'une série d'activités dont vous avez la responsabilité.

Les Variables et Scopes de Variables

Avant de commencer à parler de scope, petite définition de celui-ci : Un Scope de variables est un réceptacle qui est en mesure de stocker / recevoir / exposer des variables afin de les rendre accessible aux activités enfant de l'activité possédant le scope (un bon exemple : la séquence et sa collection de variables)

Avoir une activité qui sert de scope de variables n'a rien de bien compliqué. Votre activité a juste besoin d'avoir une propriété publique de type Collection<Variable> nommée Variables.

Toujours dans l'idée de faire simple, on ne doit ajouter une collection de variables à une activité que si celles-ci doivent être utilisées par les activités enfants. Si l'on n'est pas certain que les activités enfant ont besoin d'un scope, on ne doit pas en ajouter. On laisse alors à l'utilisateur la possibilité d'ajouter lui-même un scope (Sequence, While, Parallel par exemple qui peut englober votre activité).

Pour ce qui est des variables internes aussi appelées ImplementationVariables (ne cherchez pas dans la MSDN, ce n'est pas une classe), il s'agit d'un cas bien particulier.

Le principe est simple : vos activités peuvent avoir des variables internes classiques (Int32, String… etc…) qui en sont pas accessibles via l'extérieur de votre activité. Ces variables n'étant pas de type Variable<T>, elles ne font pas partie des metadata déclarées dans votre workflow. Leur état ne pourra donc pas persister avec votre workflow et donc elles ne pourront pas être ravivées. Pour évider ceci, il suffit donc de faire ce que l'on appelle « déclarer l'implémentation d'une variable ».

Exemple : pour une activité MySequence disposant d'un index, on déclare une variable interne et on enregistre sa metadata via la méthode AddImplementationVariable() de la NativeActivityMetadata.

```
public class MySequence : NativeActivity
{
    private Variable<Int32> m_Index;

    public MySequence()
    {
        this.m_Index = new Variable<Int32>();
    }
    protected override void CacheMetadata(NativeActivityMetadata metadata)
    {
        // déclaration de l'"ImplementationVariable"
        metadata.AddImplementationVariable(this.m_Index);
    }

    protected override void Execute(NativeActivityContext context)
    {
        // ... code inutile pour cet exemple
    }
}
```

Du fait que la variable soit déclarée comme faisant partie des metadatas, m_Index pourra être restaurée dans son état courant si votre activité venait à persister, et non pas dans son état initial.

Utiliser la méthode CacheMetadata

La méthode CacheMetadata est un élément crucial de votre activité. Non seulement elle permet d'ajouter des fonctionnalités à votre activité, mais en plus elle a un impact direct sur les performances de celle-ci.

Comment ?

L'implémentation par défaut de la méthode CacheMetadata utilise la réflexion pour déclarer comme faisant partie des Metadata de votre activité, tout élément public de celle-ci ayant besoin du contexte d'exécution du workflow (Activity, Variable, Argument …). Cette implémentation a l'avantage d'être la plus générique possible, mais a l'inconvénient d'avoir un coût non négligeable (ressources et performance) du fait de la réflexion.

Quand on code une activité personnalisée, il faut donc réécrire sa méthode CacheMetadata.

Voici donc quelques exemples de code qui vous permettront de coder proprement vos méthodes CacheMetaData :

- L'ajout d'activités aux metadata :

```
public Activity MyActivity { get; set; }
public Collection<Activity> MyActivities { get; set; }
protected override void CacheMetadata(NativeActivityMetadata metadata)
{    // Ajout d'une activité

    metadata.AddChild(this.MyActivity);
    // Ajout de plusieurs activités
    if (this.MyActivities != null && this.MyActivities.Count > 0)
    {
        this.MyActivities.ToList().ForEach(a => metadata.AddChild(a));
        // Merci Linq ;)
    }
}
```

- L'ajout de variables aux metadata :

```
public Variable<String> MyVariable { get; set; }public Collection<Variable> MyVariables { get; set; }
protected override void CacheMetadata(NativeActivityMetadata metadata)
{
    // Ajout d'un variable
    metadata.AddVariable(this.MyVariable);
    // Ajout de plusieurs varaibles
    if (this.MyVariables != null && this.MyVariables.Count > 0)
    {
        this.MyVariables.ToList().ForEach(v => metadata.AddVariable(v));
        // Merci Linq ;)
    }
}
```

- L'ajout d'arguments aux metadata :

```
public InArgument<String> MyStringArgument { get; set; }
protected override void CacheMetadata(NativeActivityMetadata metadata)
{
    // Ajout d'un argument
    RuntimeArgument arg = new RuntimeArgument("MyInArgument", typeof(String), ArgumentDirection.In);
    metadata.AddArgument(arg);
    metadata.Bind(this.MyInArgument, arg);
}
```

- L'ajout d'arguments dont le type est inconnu (la plus tordue je vous l'accorde, mais cela m'est déjà arrivé) :

```
public InOutArgument MyArgument { get; set; }
protected override void CacheMetadata(NativeActivityMetadata metadata)
{
    // Ajout d'un argument de type inconnu
    Type argType = this.MyStringArgument == null
        ? typeof(Object)
        : this.MyArgument.ArgumentType;

    RuntimeArgument arg = new RuntimeArgument("MyArgument", argType, ArgumentDirection.InOut);
    metadata.AddArgument(arg);
    metadata.Bind(this.MyStringArgument, arg);
}
```

Note importante : pas de variables, ni d'activités dans les metadatas produites par vos CodeActivities (restons logiques).

Sérialisation XAML

XAML et Windows Workflow Foundation sont étroitement liés. Si on souhaite optimiser ses activités, il faut donc se soucier du code XAML généré quand celles-ci sont intégrées à un workflow XAML.

Il y a donc là quelques règles à respecter :

- Le **pattern Creat/Set/use** : cela ne parle certainement pas à grand monde. En fait l'idée est toute simple. Étant donné que la déclaration de votre activité va devoir passer par une sérialisation puis un désérialisation, il est impératif que vos activités aient une structure propre, avec : un constructeur par défaut et des propriétés sérialisées avec des accesseurs

Get et Set. En théorie on doit déjà avoir tout cela… même si bien souvent on néglige le constructeur par défaut (dommage pour certaines optimisations comme l'instanciation de variables interne readonly)

- Utiliser l'attribut **[DefautValue(…)]** afin de minimiser le XAML à écrire. Si le designer ne change pas la valeur de votre propriété par rapport à son état par défaut, celle-ci n'ajoutera pas de XAML car il est sous-entendu qu'elle utilise sa valeur par défaut.

- Utiliser l'attribut **[DependsOn("…")].** Il s'agit là d'une idée purement esthétique dont l'objectif est de contraindre le XAML à être écrit dans un ordre précis. Pour cela on donne à l'attribut le nom de la propriété à laquelle on veut que la propriété succède… c'est une histoire de goûts. Ceci fonctionne sur tout type de propriété.

- Utiliser l'attribut **[ContentProperty]** afin d'indiquer la propriété censée faire office de contenu de votre activité. Quand on n'a qu'un Body, ou une collection Activities, on va la désigner comme ContentProperty. Ceci réduit grandement le XAML.

Afin de présenter les bienfaits de ces pratiques, j'ai repris mon activité EntityScope que j'ai présentée dernièrement. Voici donc son code respectant les bonnes pratiques :

```
using System;
using System.Activities;
using System.ComponentModel;
using System.Data.Objects;
using System.Windows.Markup;
namespace MyLib.WF4.EntityFramework
{
    /// <summary>
    /// Activity based on NativeActivity<TResult>
    /// </summary>
    [ContentProperty("Body")]
    public sealed class EntityScope : NativeActivity
    {
        public const String ObjectContextName = "ObjectContext";

        [DefaultValue(null)]
        [RequiredArgument]
        [Browsable(true)]
        [DependsOn("SaveChanges")]
        public InArgument<ObjectContext> ObjectContext { get; set; }

        [DefaultValue(true)]
        [Browsable(true)]
        public Boolean SaveChanges { get; set; }

        [DefaultValue(null)]
        [Browsable(false)]
        public Activity Body { get; set; }

        public EntityScope()
        {
            this.SaveChanges = true;
        }

        /// <summary>
        /// Execute
        /// </summary>
        /// <param name="context">WF context</param>
        /// <returns></returns>
        protected override void Execute(NativeActivityContext context)
        {
            if (this.Body != null)
```

```
            {
                ObjectContext obj = this.ObjectContext.Get(context);
                context.Properties.Add(ObjectContextName, obj);
                context.ScheduleActivity(this.Body, new CompletionCallback(this.BodyCompletionCallback)
);
            }
        }

        /// <summary>
        /// Body Completion Callback
        /// </summary>
        /// <param name="contexte"></param>
        /// <param name="completedInstance"></param>
        private void BodyCompletionCallback(NativeActivityContext context, ActivityInstance completedIn
stance)
        {
            ObjectContext c = this.ObjectContext.Get(context);
            if (c != null)
            {
                if (this.SaveChanges)
                {
                    c.SaveChanges();
                }
                c.Dispose();
                c = null;
            }
        }

        /// <summary>
        /// Register activity's metadata
        /// </summary>
        /// <param name="metadata"></param>
        protected override void CacheMetadata(NativeActivityMetadata metadata)
        {
            // [ObjectContext] Argument must be set
            if (this.ObjectContext == null)
            {
                metadata.AddValidationError(
                new System.Activities.Validation.ValidationError(
                "[ObjectContext] argument must be set!",
                false,
                "ObjectContext"));
            }
            else
            {
                RuntimeArgument arg = new RuntimeArgument(ObjectContextName, typeof(ObjectContext), Arg
umentDirection.In);
                metadata.AddArgument(arg);
                metadata.Bind(this.ObjectContext, arg);
            }
            // [Body] Argument must be set
            if (this.Body == null)
            {
                metadata.AddValidationError(
                new System.Activities.Validation.ValidationError(
                "[Body] argument must be set!",
                false,
                "Body"));
            }
            else
            {
                metadata.AddChild(this.Body);
            }
        }
    }
}
```

Si j'insère cette activité dans un workflow en laissant ses propriétés par défaut j'aurai le XAML suivant :

```
<local:EntityScope />
```

Si je change ces propriétés :

```
<local:EntityScope ObjectContext="[New DemoModel()]" SaveChanges="False" />
```

Et si je change ces propriétés et que j'insère une séquence dans le body:

```
<local:EntityScope ObjectContext="[New DemoModel()]" SaveChanges="False">
   <Sequence />
</local:EntityScope>
```

Les attributs sont dans l'ordre voulu et la séquence qui se trouve dans le body devient le contenu de mon activité

Et si je retire mes attributs destinés à optimiser le XAML, dans les trois mêmes situations, j'aurai les codes suivants :

```
<local:EntityScope ObjectContext="{x:Null}" SaveChanges="True">
   <x:Null />
</local:EntityScope>

<local:EntityScope ObjectContext="[New DemoModel()]" SaveChanges="False">
   <x:Null />
</local:EntityScope>

<local:EntityScope ObjectContext="[New DemoModel()]" SaveChanges="False">
   <local:EntityScope>
      <Sequence />

   </local:EntityScope>
</local:EntityScope>
```

Je crois que le constat est clair. Sur une aussi petite activité, le XAML est déjà bien plus important et ceci est valable quel que soit la situation des différentes propriétés de votre activité.

Les propriétés d'exécution (ExecutionProperties)

ExecutionProperties : quel nom barbare pour une chose si pratique. Vous ne le savez certainement pas, mais vous en avez peut-être déjà utilisé. L'objectif de ces "propriétés" est de permettre le partage d'informations entre une activité parente et ses enfants. En général les ExecutionProperties sont utilisées dans une activité de type "Scope".

Le meilleur exemple est très certainement le CorrelationScope avec sa propriété CorrelatesWith. Quand on affecte un CorrelationHandle à cette propriété, toute activité sachant utiliser cette ExecutionProperty, utilisera sa valeur en lieu et place de sa propre propriété CorrelatesWith (si celle-ci n'est pas déjà définie). On réduit alors le travail à faire lorsque l'on design un workflow, et on améliore considérablement l'expérience utilisateur.

Plutôt que de partir sur de l'abstrait, j'ai décidé de présenter ici une petite partie d'un projet personnel permettant la manipulation de données via Entity Framework. Pour l'exemple je ne présenterai ici que 4 activités basiques :

- EntityScope : Permet d'avoir un scope chargé de partager un ObjectContext entre plusieurs activités (ceci par le biais d'une Execution Property)

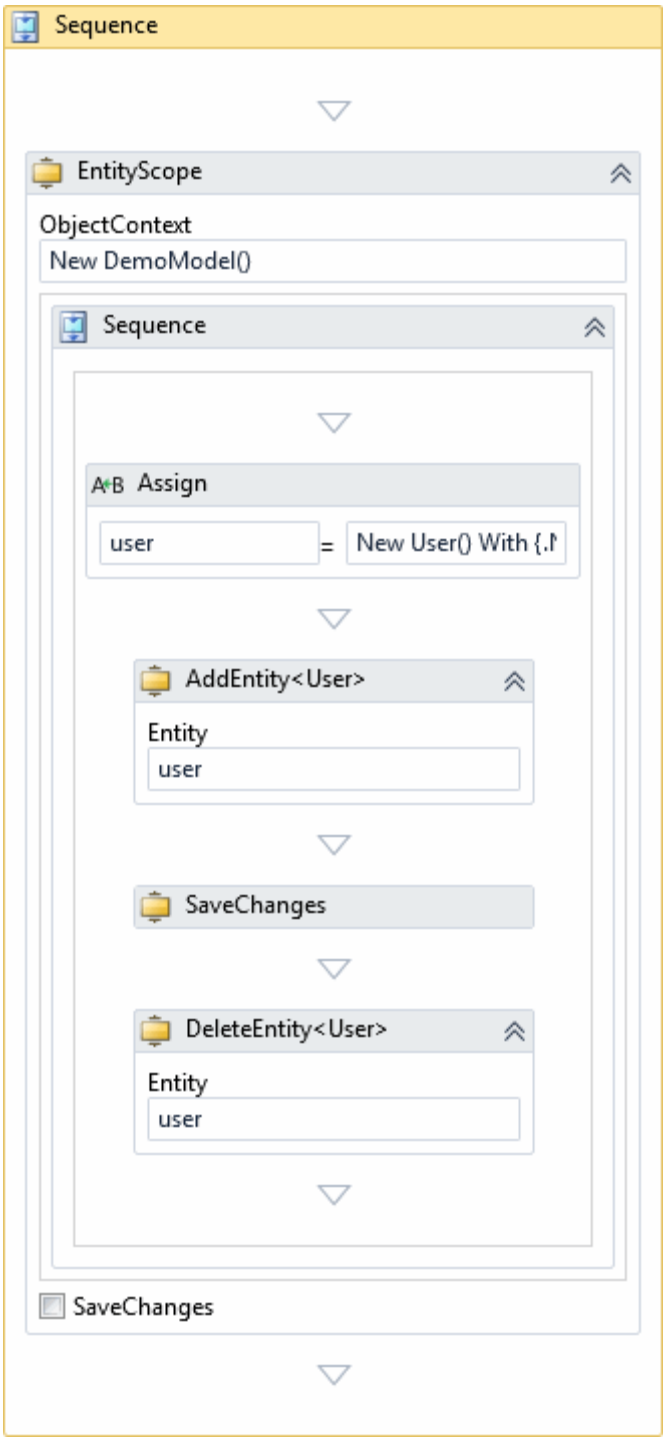
• AddEntity<T> : Ajoute à un ObjectContext une entité de type T héritant bien entendu d'EntityObject .

• DeleteEntity<T> : Supprimer d'un ObjectContext une entité de type T héritant bien entendu d'EntityObject.

• SaveChanges : Execute la méthode SaveChanges de l'ObjectContext

Ah la différence du sample WF4 qui contient un scénario WF+EF, j'ai voulu coder des activités pouvant fonctionner dans un scope avec ExecutionPropertie aussi bien qu'en dehors.

Voici un petit Workflow mettant en scène tout ce petit monde :



Pour commencer, je vais décrire l'EntityScope et sa manière d'ajouter une ExecutionProperty.

Première notion à retenir : les Execution Properties sont accessibles via la collection Properties d'un NativeActivityContext. Pour ajouter / consulter cette collection, il faut donc obligatoirement avoir une activité héritant de NativeActivity. Cette collection est similaire à un ensemble clé+valeur. Pour ajouter une propriété, on utilise donc une méthode Add(), à laquelle on passer une clé sous forme de String et une valeur qui dans le cas présent est mon ObjectContext.

Voici donc le code de mon EntityScope :

```
using System;
```

```
using System.Activities;
using System.ComponentModel;
using System.Data.Objects;
using System.Windows.Markup;

namespace MyLib.WF4.EntityFramework
{
    /// <summary>
    /// Activity based on NativeActivity<TResult>
    /// </summary>
    [ContentProperty("Body")]
    public sealed class EntityScope : NativeActivity
    {
        public const String ObjectContextName = "ObjectContext";
        [DefaultValue(null)]
        [RequiredArgument]
        [Browsable(true)]
        public InArgument<ObjectContext> ObjectContext { get; set; }
        [DefaultValue(true)]
        [Browsable(true)]
        public Boolean SaveChanges { get; set; }
        [DefaultValue(null)]
        [Browsable(false)]
        public Activity Body { get; set; }
        public EntityScope()
        {
            this.SaveChanges = true;
        }
        /// <summary>
        /// Execute
        /// </summary>
        /// <param name="context">WF context</param>
        /// <returns></returns>
        protected override void Execute(NativeActivityContext context)
        {
            if (this.Body != null)
            {
                ObjectContext obj = this.ObjectContext.Get(context);
                context.Properties.Add(ObjectContextName, obj);
                context.ScheduleActivity(this.Body, new CompletionCallback(this.BodyCompletionCallback)
);
            }
        }
        /// <summary>
        /// Body Completion Callback
        /// </summary>
        /// <param name="contexte"></param>
        /// <param name="completedInstance"></param>
        private void BodyCompletionCallback(NativeActivityContext context, ActivityInstance completedIn
stance)
        {
            ObjectContext c = this.ObjectContext.Get(context);
            if (c != null)
            {
                if (this.SaveChanges)
                {
                    c.SaveChanges();
                }
                c.Dispose();
                c = null;
            }
        }
        /// <summary>
        /// Register activity's metadata
        /// </summary>
        /// <param name="metadata"></param>
        protected override void CacheMetadata(NativeActivityMetadata metadata)
        {
            // [ObjectContext] Argument must be set
            if (this.ObjectContext == null)
            {
                metadata.AddValidationError(
                new System.Activities.Validation.ValidationError(
                "[ObjectContext] argument must be set!",
```

```
                false,
                "ObjectContext"));
            }
            else
            {
                RuntimeArgument arg = new RuntimeArgument(ObjectContextName, typeof(ObjectContext), Arg
umentDirection.In);
                metadata.AddArgument(arg);
                metadata.Bind(this.ObjectContext, arg);
            }
            // [Body] Argument must be set
            if (this.Body == null)
            {
                metadata.AddValidationError(
                new System.Activities.Validation.ValidationError(
                "[Body] argument must be set!",
                false,
                "Body"));
            }
            else
            {
                metadata.AddChild(this.Body);
            }
        }
    }
}
```

Afin de faciliter le reste du travail (à savoir la récupération des ExecutionProperties) j'ai codé une interface et une méthode d'extension pour les classes implémentant cette interface (mes autres activités).

Ce code me permet de récupérer l'ObjectContext de mon activité, qu'il soit défini via une ExecutionPropertie ou par l'argument de l'activité implémentant l'interface.

```
using System.Activities;
using System.Data.Objects;
namespace MyLib.WF4.EntityFramework
{
    /// <summary>
    /// Interface des activity utilisant un Objectcontext
    /// </summary>
    interface IEntityActivity
    {
        InArgument<ObjectContext> ObjectContext { get; set; }
    }
    internal static class EntityActivityExtension
    {
        /// <summary>
        /// Retourner l'ObjectContext de l'activité
        /// </summary>
        /// <param name="activity"></param>
        /// <param name="contexte"></param>
        /// <returns></returns>
        public static ObjectContext GetObjectContext(this IEntityActivity activity, NativeActivityConte
xt context)
        {
            if (activity.ObjectContext == null
            || activity.ObjectContext.Expression == null)
            {
                ObjectContext objectContext = context.Properties.Find(EntityScope.ObjectContextName)
                as ObjectContext;
                if (objectContext == null)
                {
                    throw new ValidationException("'ObjectContext' ne peut être vide!");
                }
                else
                {
                    return objectContext;
                }
            }
```

```
            else
            {
                return activity.ObjectContext.Get(context);
            }
        }
    }
}
```

On retrouve ici une approche similaire aux extensions de WF4.

Utilisé dans une activité simple telle que le SaveChanges, cela donne ce code :

```
using System;
using System.Activities;
using System.ComponentModel;
using System.Data.Objects;
namespace MyLib.WF4.EntityFramework
{
    /// <summary>
    /// Activity based on NativeActivity<TResult>
    /// </summary>
    public sealed class SaveChanges : NativeActivity<Int32>, IEntityActivity
    {
        [Browsable(true)]
        [DefaultValue(null)]
        public InArgument<ObjectContext> ObjectContext { get; set; }
        /// <summary>
        /// Execute
        /// </summary>
        /// <param name="context">WF context</param>
        /// <returns></returns>
        protected override void Execute(NativeActivityContext context)
        {
            ObjectContext objectContext = this.GetObjectContext(context);
            Int32 result = objectContext.SaveChanges();
            // Return value
            this.Result.Set(context, result);
        }
        /// <summary>
        /// Register activity's metadata
        /// </summary>
        /// <param name="metadata"></param>
        protected override void CacheMetadata(NativeActivityMetadata metadata)
        {
            // Register In arguments
            RuntimeArgument objectContextArg = new RuntimeArgument("ObjectContext", typeof(ObjectContex
t), ArgumentDirection.In);
            metadata.AddArgument(objectContextArg);
            metadata.Bind(this.ObjectContext, objectContextArg);
            // Register Out arguments
            RuntimeArgument resultArg = new RuntimeArgument("Result", typeof(Int32), ArgumentDirection.
Out);
            metadata.AddArgument(resultArg);
            metadata.Bind(this.Result, resultArg);
        }
    }
}
```

Rien de bien compliqué, on est même dans l'extrêmement simple, et pourtant on tire profit d'une ExecutionProperties.

Les activités AddEntity et DelteEntity sont basées sur le même principe. D'où un code relativement simple :

AddEntity :

```
using System.Activities;
using System.ComponentModel;
using System.Data.Objects;
using System.Data.Objects.DataClasses;
namespace MyLib.WF4.EntityFramework
{
    /// <summary>
    /// Activity based on NativeActivity<TResult>
    /// </summary>
    public sealed class AddEntity<T> : NativeActivity, IEntityActivity where T : EntityObject
    {
        [RequiredArgument]
        [Browsable(true)]
        [DefaultValue(null)]
        public InArgument<T> Entity { get; set; }
        [Browsable(true)]
        [DefaultValue(null)]
        public InArgument<ObjectContext> ObjectContext { get; set; }
        /// <summary>
        /// Execute
        /// </summary>
        /// <param name="context">WF context</param>
        /// <returns></returns>
        protected override void Execute(NativeActivityContext context)
        {
            // Obtain the runtime value of the Text input argument
            T entity = context.GetValue(this.Entity);
            ObjectContext objectContext = this.GetObjectContext(context);
            ObjectSet<T> objectSet = objectContext.CreateObjectSet<T>();
            objectSet.AddObject(entity);
        }
        /// <summary>
        /// Register activity's metadata
        /// </summary>
        /// <param name="metadata"></param>
        protected override void CacheMetadata(NativeActivityMetadata metadata)
        {
            // Register In arguments
            RuntimeArgument objectContextArg = new RuntimeArgument("ObjectContext", typeof(ObjectContex
t), ArgumentDirection.In);
            metadata.AddArgument(objectContextArg);
            metadata.Bind(this.ObjectContext, objectContextArg);
            // Register In arguments
            RuntimeArgument arg = new RuntimeArgument("Entity", typeof(T), ArgumentDirection.In);
            metadata.AddArgument(arg);
            metadata.Bind(this.Entity, arg);
            // [Entity] Argument must be set
            if (this.Entity == null)
            {
                metadata.AddValidationError(
                new System.Activities.Validation.ValidationError(
                "'Entity' argument must be set!",
                false,
                "Entity"));
            }
        }
    }
}
```

DeleteEntity :

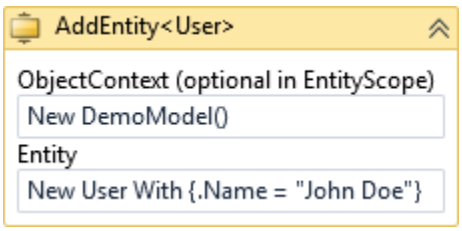
```
using System.Activities;
using System.ComponentModel;
using System.Data.Objects;
using System.Data.Objects.DataClasses;
namespace MyLib.WF4.EntityFramework
{
    /// <summary>
    /// Activity based on NativeActivity<TResult>
    /// </summary>
    public sealed class DeleteEntity<T> : NativeActivity, IEntityActivity where T : EntityObject
    {
```

```
        [RequiredArgument]
        [Browsable(true)]
        [DefaultValue(null)]
        public InArgument<T> Entity { get; set; }
        [Browsable(true)]
        [DefaultValue(null)]
        public InArgument<ObjectContext> ObjectContext { get; set; }
        /// <summary>
        /// Execute
        /// </summary>
        /// <param name="context">WF context</param>
        /// <returns></returns>
        protected override void Execute(NativeActivityContext context)
        {
            // Obtain the runtime value of the Text input argument
            T entity = context.GetValue(this.Entity);
            ObjectContext objectContext = this.GetObjectContext(context);
            objectContext.DeleteObject(entity);
        }
        /// <summary>
        /// Register activity's metadata
        /// </summary>
        /// <param name="metadata"></param>
        protected override void CacheMetadata(NativeActivityMetadata metadata)
        {
            // Register In arguments
            RuntimeArgument objectContextArg = new RuntimeArgument("ObjectContext", typeof(ObjectContex
t), ArgumentDirection.In);
            metadata.AddArgument(objectContextArg);
            metadata.Bind(this.ObjectContext, objectContextArg);
            // Register In arguments
            RuntimeArgument arg = new RuntimeArgument("Entity", typeof(T), ArgumentDirection.In);
            metadata.AddArgument(arg);
            metadata.Bind(this.Entity, arg);
            // [Entity] Argument must be set
            if (this.Entity == null)
            {
                metadata.AddValidationError(
                new System.Activities.Validation.ValidationError(
                "'Entity' argument must be set!",
                false,
                "Entity"));
            }
        }
    }
}
```

Là où les choses deviennent intéressantes, c'est à partir du moment où l'on veut un designer lié à l'activité qui tire parti du fait d'être dans ou hors d'un EntityScope…

La situation "dans un scope" est celle qui est représentée par la toute première capture de cet article.

Hors du scope on préférera avoir un design tel que celui-ci :



Ce qui facilite la saisie d'un ObjectContext.

Alors, comment faire?

Premièrement, on code un designer XAML simple contenant l'interface visuelle complète :

```
<sap:ActivityDesigner x:Class="MyLib.WF4.EntityFramework.Design.AddEntityDesigner"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:sap="clr-namespace:System.Activities.Presentation;assembly=System.Activities.Presentation"
xmlns:sapv="clr-
namespace:System.Activities.Presentation.View;assembly=System.Activities.Presentation" xmlns:sapc="clr-
namespace:System.Activities.Presentation.Converters;assembly=System.Activities.Presentation"
xmlns:s="clr-namespace:System;assembly=mscorlib"
xmlns:ef="clr-namespace:System.Data.Objects;assembly=System.Data.Entity">
    <sap:ActivityDesigner.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="pack://application:,,,/MyLib.WF4.EntityFramework.Design;com
ponent/Themes/Generic.xaml" />
            </ResourceDictionary.MergedDictionaries>
            <sapc:ArgumentToExpressionConverter x:Key="ArgumentToExpressionConverter"/>
        </ResourceDictionary>
    </sap:ActivityDesigner.Resources>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <TextBlock Grid.Row="0" x:Name="ObjectContextLabel" Text="ObjectContext (optional in EntityScop
e)" />
        <sapv:ExpressionTextBox Grid.Row="1" x:Name="ObjectContext"
        OwnerActivity="{Binding Path=ModelItem}"
        Expression="{Binding Path=ModelItem.ObjectContext, Mode=TwoWay,
        Converter={StaticResource ResourceKey=ArgumentToExpressionConverter},
        ConverterParameter=In}" ExpressionType="{x:Type ef:ObjectContext}" />
        <TextBlock Grid.Row="2" Text="Entity" />
        <sapv:ExpressionTextBox Grid.Row="3" x:Name="Entity"
        OwnerActivity="{Binding Path=ModelItem}"
        Expression="{Binding Path=ModelItem.Entity, Mode=TwoWay,
        Converter={StaticResource ResourceKey=ArgumentToExpressionConverter},
        ConverterParameter=In}" />
    </Grid>
</sap:ActivityDesigner>
```

Et on y ajoute une logique permettant de retrouver l'éventuel EntityScope parent et donc de masquer les contrôles inutiles :

```
using System;
using System.Activities.Presentation.Model;
using System.Windows;
namespace MyLib.WF4.EntityFramework.Design
{
    // Logique d'interaction pour AddEntityDesigner.xaml
    public partial class AddEntityDesigner
    {
        public AddEntityDesigner()
        {
            InitializeComponent();
            this.Loaded += new RoutedEventHandler(AddEntityDesigner_Loaded);
        }
        void AddEntityDesigner_Loaded(object sender, RoutedEventArgs e)
        {
            // Type EF manipulé
            Type t = this.ModelItem.Properties["Entity"].PropertyType.GetGenericArguments()[0];
            this.Entity.ExpressionType = t;
            Visibility visibility = IsInEntityScope(this.ModelItem)
            ? Visibility.Collapsed
            : Visibility.Visible;
            this.ObjectContext.Visibility = visibility;
            this.ObjectContextLabel.Visibility = visibility;
        }
```

```
        private static Boolean IsInEntityScope(ModelItem modelItem)
        {
            if (modelItem.Parent == null)
            {
                return false;
            }
            else
            {
                if (modelItem.Parent.ItemType == typeof(EntityScope))
                {
                    return true;
                }
                else
                {
                    return IsInEntityScope(modelItem.Parent);
                }
            }
        }
    }

}
```

Le secret se trouve dans la simple petite méthode IsInEntityScope(). Celle-ci a pour mission de parcourir l'arbre XAML représentant le Workflow à la recherche d'un éventuel EntityScope. Évidemment il faut aime le proxy ModelItem. Mais avec un peu de pratique, on se rend vite compte que ce n'est pas très compliqué.

Donc, si je résume :

Avec ce type de code et les ExecutionProperties, on peut facilement échanger des données entre activités (parents et enfants) sans que la personne chargée de designer le workflow n'ai besoin de passer son temps à faire du copier-coller. De plus, nos activités peuvent fonctionner sans ExecutionProperty.

La validation et les contraintes

Il arrive que vos activités aient à répondre à des contraintes complexes concernant leur entourage ou leurs éléments internes. Le plus souvent, on cherchera à interdire une activité de contenir un type particulier ou d'avoir une propriété non définie.

Pour répondre à ce genre de situation, il existe plusieurs approches. La plus simple consiste à utiliser à placer des arguments sur les propriétés de nos activités.

- On peut utiliser l'attribut [RequiredArguments] pour indiquer les arguments devant avoir une valeur. Simple et efficace (mais ne fonctionne que sur les arguments, pas sur des propriétés CLR ou des activités… donc attention !!!).
- On peut utiliser l'attribut [OverloadGroups] pour regrouper des attributs requis. Si on a plusieurs groupes sur une activité, il suffit qu'un groupe ait ses propriétés affectées pour valider l'activité. Très pratique, ceci évite de monter une logique de psychopathe dans les méthodes CacheMetaData de vos activités.

Cette approche étant la plus évidente, je ne m'attarderai pas dessus. Je vous encourage à lire la MSDN qui a déjà un très bon exemple sur ces deux sujets : Arguments obligatoires et groupes surchargés.

Arrive alors la solution un peu moins évidente, mais qui aura l'avantage d'être utilisable avec des activités et les propriétés CLR : la notification d'erreur via les metadata.

Cette approche passe bien évidemment par la réécriture de la méthode CacheMetadata.

Par exemple, l'activité suivante a deux arguments dont on souhaite forcer l'affectation :

```
using System;
using System.Activities;
using System.Activities.Validation;
using System.Diagnostics;

/// <summary>
/// Activité qui permet d'exécuter un programme
/// </summary>
public class ExecuterUnProgram : CodeActivity
{
    #region "Déclarations"
    private InArgument<String> m_Program;
    private InArgument<String> m_Arguments;
    #endregion

    #region "Constructeur / destructeur"
    public ExecuterUnProgram() { }
    #endregion

    #region "Propriétés"
    /// <summary>
    /// Program à exécuter
    /// </summary>
    public InArgument<String> Program
    {
        get { return this.m_Program; }
        set { this.m_Program = value; }
    }

    /// <summary>
    /// Arguments du Program à exécuter
    /// </summary>
    public InArgument<String> Arguments
    {
        get { return m_Arguments; }
        set { m_Arguments = value; }
    }

    #endregion

    #region "Méthodes"

    protected override void CacheMetadata(CodeActivityMetadata metadata)
    {
        // Test l'état de l'argument "Program"
        if (this.m_Program == null)
        {
            // Si vide on ajoute une erreur aux metadatas
            metadata.AddValidationError(
                new ValidationError(
                    "La propriété [Program] ne dois pas être vide.",
                    false,
                    "Program"));
        }
        else
        {
            // Si non vide on déclare l'argument dans les metadatas
            RuntimeArgument arg =new RuntimeArgument("Program", typeof (string), ArgumentDirection.In);
            metadata.AddArgument(arg);
            metadata.Bind(this.Program, arg);
        }

        // Teste l'état de l'argument "Arguments"
        if (this.m_Arguments == null)
```

```
        {
            // Si vide on ajoute une erreur aux metadatas
            metadata.AddValidationError(
                new ValidationError(
                    "La propriété [Arguments] ne dois pas être vide.",
                    false,
                    "Arguments"));
        }
        else
        {
            // Si non vide on déclare l'argument dans les metadatas
            RuntimeArgument arg = new RuntimeArgument("Arguments", typeof(string), ArgumentDirection.In
);
            metadata.AddArgument(arg);
            metadata.Bind(this.Arguments, arg);
        }
    }

    /// <summary>
    /// Action  executée par l'activité
    ///    /// </summary>
    /// /// <param name="contexte"></param>
    protected override void Execute(CodeActivityContext context)
    {
        Process.Start(this.m_Program.Get(contexte), this.m_Arguments.Get(contexte));
    }

    #endregion
}
```

Par ce cas, on peut facilement voir à quel point la manipulation des metadata pour vérifier la situation de l'activité peut s'avérer simple.

Dernier mode de validation envisageable : **les contraintes**. Les contraintes sont des workflows particuliers qui peuvent s'ajouter à une liste présente dans chaque activité (liste de contraintes). Les workflows de cette liste sont exécutés avant toute utilisation de l'activité.

Par exemple, pour une activité telle que :

```
public class MyActivity : Activity
{
    public Activity Cildren1 { get; set; }
    public Activity Cildren2 { get; set; }

    public MyActivity()
    {
        this.Constraints.Add(MyContraints.GetCantBeEmpty<MyActivity>(c => c.Cildren1));
        this.Constraints.Add(MyContraints.GetCantBeEmpty<MyActivity>(c => c.Cildren2));
    }
}
```

J'ai codé une classe statique MyContraints qui me retourne des contraintes génériques que j'ai définies. Dans le cas présent, il s'agit de s'assurer que mes propriétés ont bien été affectées.
Ce qui donne le code suivant :

```
using System;
using System.Activities;
using System.Activities.Statements;
using System.Activities.Validation;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;
using System.Reflection;
public static partial class MyContraints
{
    public static Constraint GetCantBeEmpty<T>(Expression<Func<T, Object>> property) where T : Activity
```

```
    {
        DelegateInArgument<T> myActivity = new DelegateInArgument<T>();
        DelegateInArgument<ValidationContext> context = new DelegateInArgument<ValidationContext>();

        // Récupération de la propriété
        PropertyInfo propertyInfo = (PropertyInfo)(((MemberExpression)property.Body).Member);

        return new Constraint<T>
        {
            Body = new ActivityAction<T, ValidationContext>
            {
                Argument1 = myActivity,
                Argument2 = context,
                Handler = new AssertValidation
                {
                    // Test devant être vrai
                    Assertion =
                        new InArgument<Boolean>(
                        (c) => propertyInfo.GetValue(myActivity.Get(c), null) != null),
                    // Message affiché dans le designer de WorkFlow
                    Message = new InArgument<String>(
                        (c) => "La propriété [" + propertyInfo.Name + "] ne dois pas être vide."),
                    // Propriété à mettre en évidence dans le designer
                    PropertyName = propertyInfo.Name
                }
            }
        };
    }
}
```

En soit, il n'y a rien de bien compliqué ici. Le code spécifique à notre contrainte se trouve en fait dans la propriété Handler. C'est ici que l'on trouve l'activité AssertValidation, qui comme son nom l'indique, va faire remonter le fait qu'une condition (propriété Assertion) soit valide ou non. Dans le cas présent on teste si la valeur de la propriété choisie n'est pas nulle.

Mais on peut très bien travailler sur des scénarios un peu plus évolués. Par exemple, ajouter une contrainte interdisant à notre activité de contenir plus de X activités (activités enfants des enfants comprises).

Ce qui donne un code tel que celui-ci :

```
public static Constraint GetNoMoreThanXChildren<T>(Int32 x) where T : Activity
{
    DelegateInArgument<T> myActivity = new DelegateInArgument<T>();
    DelegateInArgument<ValidationContext> context = new DelegateInArgument<ValidationContext>();
    Variable<IEnumerable<Activity>> children = new Variable<IEnumerable<Activity>>();

    return new Constraint<T>
    {
        Body = new ActivityAction<T, ValidationContext>
        {
            Argument1 = myActivity,
            Argument2 = context,
            Handler = new Sequence
            {
                Variables =
                {
                    children
                },
                Activities =
                    {
                        // Récupération de la liste des activités enfantes
                        new GetChildSubtree
                        {
                            ValidationContext = context,
                            Result = children
                        },
                        // Test final
                        new AssertValidation
                        {
```

```
                            Assertion = new InArgument<Boolean>(c => children.Get(c).Count() < x ),
                            Message = new InArgument<String>(String.Format("Cette activité ne peut pas
contenir plus de {0} activités!",x-1)),
                            PropertyName = new InArgument<String>(c => myActivity.Get(c).DisplayName)
                        }
                    }
                }
            }
        };
}
```

Si on décortique ce workflow, on trouve une activité GetChildSubTree chargée de récupérer la liste des activités contenues. Ensuite, on retrouve notre AssertValidation qui teste si le nombre de ces enfants n'est pas plus important que le maximum autorisé.

Partons maintenant sur un cas un peu plus délicat : interdire un type d'activité. Évidemment le workflow sera un peu plus complexe.

```
public static Constraint GetConstraintNoActivityOfTypeU<T,U>() where T : Activity where U:Activity
{
    // L'activité qui a la contrainte
    DelegateInArgument<T> myActivity = new DelegateInArgument<T>();

    // Le contexte de validation
    DelegateInArgument<ValidationContext> context = new DelegateInArgument<ValidationContext>();
    Variable<IEnumerable<Activity>> children = new Variable<IEnumerable<Activity>>();
    Variable<Int32> i = new Variable<Int32>("i", 0);
    Variable<Boolean> writeLineExist = new Variable<Boolean>("result", false);

    return new Constraint<T>
    {
        Body = new ActivityAction<T, ValidationContext>
        {
            Argument1 = myActivity,
            Argument2 = context,
            Handler = new Sequence
            {
                Variables = { children, i, writeLineExist },
                Activities =
                {
                    // Récupération de la liste des activités enfantes
                    new GetChildSubtree
                    {
                        ValidationContext = context,
                        Result = children
                    },
                    // Boucle tant que l'on a
                    // pas trouvé un WriteLine
                    // ou que la liste n'a pas été parcourue
                    new While(c =>
                        writeLineExist.Get(c) == false &&
                        i.Get(c) < children.Get(c).Count())
                        {
                            Body = new Sequence
                            {
                                Activities =
                                {
                                    // Test si on a un WriteLine dans le 'children'
                                    new If(c =>
                                        children.Get(c).ElementAt(i.Get(c)).GetType() == typeof(U))
                                        {
                                            // Si oui on affect un Boolean true au result
                                            Then = new Assign<Boolean>{
                                                To=writeLineExist,
                                                Value=true
                                            }
                                        },
                                        // Incrémentation de i (i++)
                                        new Assign<Int32>
                                        {
                                            To = i,
                                            Value =new InArgument<int>(c =>
```

```
                                        i.Get(c) +1)
                                }
                            }
                        }
                    },
                    // Test final
                    new AssertValidation
                    {
                        Assertion = new InArgument<Boolean>(
                            c => !writeLineExist.Get(c)),
                            Message =new InArgument<String>("Cette activité ne peut pas contenir d'
activité de type " + typeof(U).ToString()),
                            PropertyName = new InArgument<String>(c => myActivity.Get(c).DisplayNam
e)
                    }
                }
            }
        }
    };
}
```

Si on y regarde de plus près, on garde l'idée d'énumérer l'ensemble des activités intégrées dans notre activité et on teste si elles ne sont pas du type interdit (U). Ensuite on compte le nombre d'activités de type (U). Si on en a, il y a erreur.

Je n'irai pas jusqu'à dire que les contraintes sont une chose simple. Certes, leur écriture est un peu particulière, mais quand elle est bien maitrisée, vous êtes en mesure de répondre à tous les besoins. Et vous n'avez aucun besoin d'utiliser une technologie ou des méthodes périphériques, tout est inclus dans vos activités. Vous restez donc le seul maitre à bord.

« Happy End »

J'ai bien conscience que les pratiques présentées ici ne sont pas toutes des plus évidentes à assimiler, je vous encourage donc à les utiliser une à une. Plus vous maitriserez les techniques présentées ici et plus vos activités seront :

- Performantes
- Maintenables
- Faciles à utiliser
- Maitrisées

Allez-y donc progressivement et vous verrez très vite les bienfaits que vous en tirerez.

Références :

Endpoint.tv - Workflow and Custom Activities - Best Practices : Série de 5 vidéos énonçant quelques bonnes pratiques pour coder une activité pour Windows Workflow Foundation.

Mon blog sur Codes-Sources.com D'où j'ai tiré plusieurs exemples, corrigés et mis à jour pour l'occasion.