Réhostez le designer de workflow dans vos applications WPF

par Lainé Vincent (autres articles)

Date de publication : 23/03/2009

Dernière mise à jour : 23/03/2009

Dans cet article nous allons nous pencher sur la possibilité d'héberger le designer de workflow dans vos applications WPF

I - Introduction

Lorsque vous développez des applications utilisant Workflow Foundation, vous aimez avoir le designer pour le faire. Indéniablement c'est plus pratique que de devoir écrire le code à la main.

Quand votre application est terminée, vous avez sans aucun doute déjà entendu quelque chose du genre : « Euh... c'est cool ton application, le workflow validation_nouvel_article est bien mais le process a changé, il faudrait le refaire ». Dans ce cas il y a deux possibilités : Soit vous êtes consultant payé à l'heure et c'est bingo pour vous, soit vous n'êtes pas payé à l'heure et c'est le drame. De manière générale on aimerait bien, dans ces moments-là, pouvoir répondre : « ben tu n'as qu'à le faire toi-même ». Le problème c'est qu'à moins que la personne soit elle-même développeur, il y a peu de chance pour qu'elle ait Visual Studio et les compétences pour aller mettre les mains dans le cambouis du workflow. La solution au problème serait donc qu'elle ait accès à un designer comme celui de Visual Studio mais de manière indépendante et contrôlé. Et là coup de chance, c'est possible. Moyennant un peu d'effort, le framework .NET vous permet de réutiliser dans vos applications le designer de workflow utilisé par Visual Studio.

II - Au commencement il y avait l'infrastructure de service du framework

La première chose à faire lorsque l'on veut réhoster le designer de workflow, c'est de comprendre le mécanisme de gestion du design dans le framework .NET. Cet article n'ayant pas pour but de décrire en profondeur ce mécanisme, je vais me contenter d'un survol.

II-A - IServiceProvider

Les objets implémentant IServiceProvider permettent à d'autres objets d'obtenir une instance d'une classe de service. Les classes de services sont des classes fournissant des méthodes générales de manipulation de l'environnement de design.

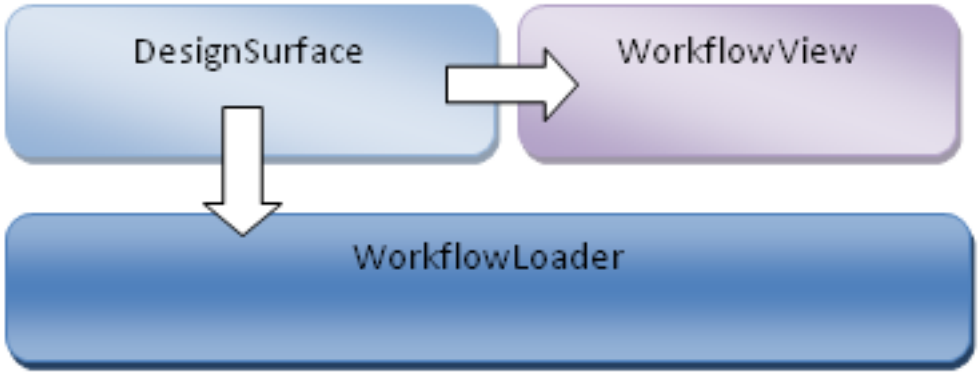
II-B - ISite

L'interface ISite n'est pas spécialement dédiée au design mais elle permet d'obtenir à partir d'un container la liste de ses composants.

III - Okay. Et mon designer de workflow dans tout ça ?

III-A - Architecture du designer de workflow

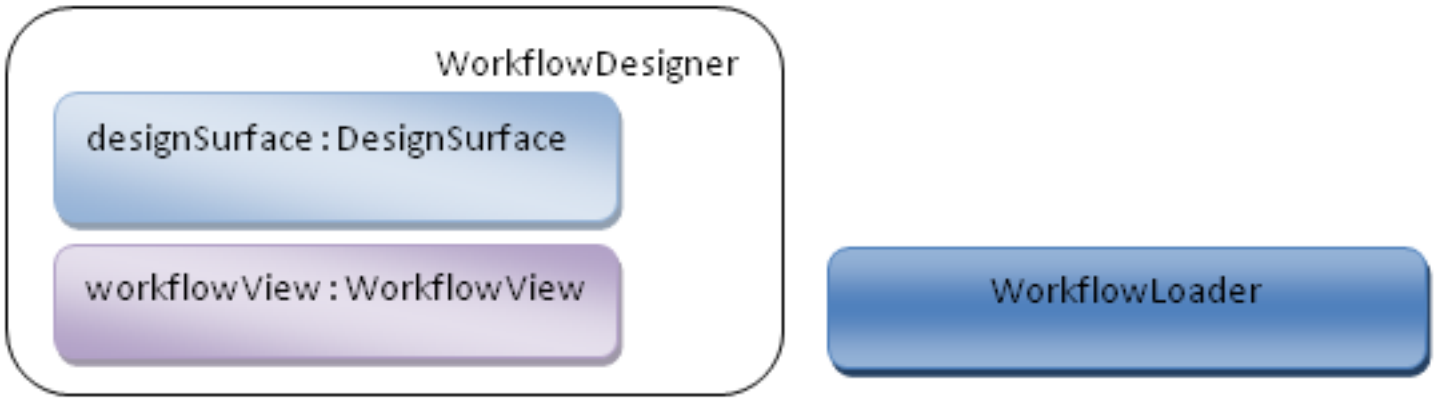
Avant de nous lancer à corps perdu dans l'écriture du code jetons un coup d'oeil sur l'architecture du designer de workflow.



Architecture du designer de workflow

Le designer de workflow est composé de trois classes principales. La classe DesignSurface n'est pas spécifique au designer de workflow mais fait partie de l'infrastructure des designers .NET. Cette classe s'occupe de gérer l'environnement de conception et de faire communiquer les services. La classe WorkflowView est la classe de base pour l'affichage du designer de workflow. C'est elle que nous allons ajouter en tant que composant dans notre UserControl. Pour finir la classe WorkflowLoader est une classe perso qui implémente WorkflowDesignerLoader. Elle va nous permettre de personnaliser le chargement et la sauvegarde du workflow.

Voici l'architecture complète du UserControl de design de workflow :



Architecture du UserControl de designer de workflow

III-B - WorkflowLoader : Gestion du chargement et de la sauvegarde du workflow

Nous allons commencer par écrire la classe gérant le chargement du workflow et de son concepteur.

Pour cela nous allons créer une classe héritant de WorflowDesignerLoader :

Notre Loader

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Workflow.ComponentModel.Design;

namespace WorkflowDesigner.Workflow
{
    internal class WorkflowLoader : WorkflowDesignerLoader
    {
  public override string FileName
        {
            get {
                return string.Empty;
            }
        }

        public override System.IO.TextReader GetFileReader(string filePath)
        {
            return null;
        }

        public override System.IO.TextWriter GetFileWriter(string filePath)
        {
            return null;
        }
    }
}
```

Etant donné que WorkflowDesignerLoader est une classe abstraite nous sommes obligés d'implémenter les 3 membres. Pour l'instant ils ne nous intéressent pas, notre designer de workflow ne s'occupant pas de charger ou sauvegarder la définition du workflow à partir d'un fichier.

La méthode virtuelle PerfomLoad est plus intéressante dans notre cas. C'est dans cette méthode que le chargement du workflow se fait en réalité. Nous allons la surcharger afin de créer l'activité de base des workflows que nous créerons.

Fonction d'initialisation du workflow dans notre loader

```
protected override void
 PerformLoad(System.ComponentModel.Design.Serialization.IDesignerSerializationManager
 serializationManager)
{
 base.PerformLoad(serializationManager);

 //Obtention du designer associé à ce loader
 IDesignerHost designerHost = (IDesignerHost)GetService(typeof(IDesignerHost));

 if (designerHost == null)
  throw new Exception("Impossible d'obtenir le service de design");

 Activity rootActivity = new SequentialWorkflowActivity();
 designerHost.Container.Add(rootActivity, "root");

 designerHost.Activate();
}
```

Les deux points d'intérêt de ce code se situent au niveau de l'obtention de l'instance du designer qui se fait grâce à un service implémentant IDesignerHost et au niveau de l'ajout de l'activité racine dans le designer qui utilise le Container. A ce stade tout ce qu'il faut pour charger un designer vide est en place. Il est temps de commencer à se préoccuper de notre UserControl.

III-C - WorkflowDesigner : Notre UserControl WPF de gestion du designer de Workflow

Notre projet étant un projet WPF, il est naturel de créer un UserControl WPF.

Le code C# de notre UserControl

```
public partial class WorkflowDesigner : UserControl
{

 private DesignSurface designSurface;
 private WorkflowView workflowView;

 public WorkflowDesigner()
 {
  InitializeComponent();
 }
}
```

Le designer de Workflow étant un composant Winform, il va falloir utiliser le composant WinformHost afin de permettre l'affichage du designer dans notre UserControl WPF.

Le code XAML de notre UserControl

```
<UserControl x:Class="WorkflowDesigner.Workflow.WorkflowDesigner"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >
    <WindowsFormsHost x:Name="panel"></WindowsFormsHost>
</UserControl>
```

Nous avons maintenant tous les éléments en place pour permettre le chargement et l'affichage de notre designer vide. Pour cela nous allons créer une méthode spécifique afin de pouvoir contrôler le moment où le designer sera chargé.

La méthode LoadWorkflowDesigner de notre UserControl

```
public void LoadWorflowDesigner()
{
 WorkflowLoader loader = new WorkflowLoader();

 this.designSurface = new DesignSurface();
 this.designSurface.BeginLoad(loader);

 //Récupération du designer
 IDesignerHost designerHost = designSurface.GetService(typeof(IDesignerHost)) as IDesignerHost;
 if (designerHost == null)
 return;
 IRootDesigner rootDesigner = designerHost.GetDesigner(designerHost.RootComponent) as IRootDesigner;
 if (rootDesigner == null)
 return;

 this.workflowView = rootDesigner.GetView(ViewTechnology.Default) as WorkflowView;
 this.panel.Child = this.workflowView;
}
```

Voyons en détail ce code :

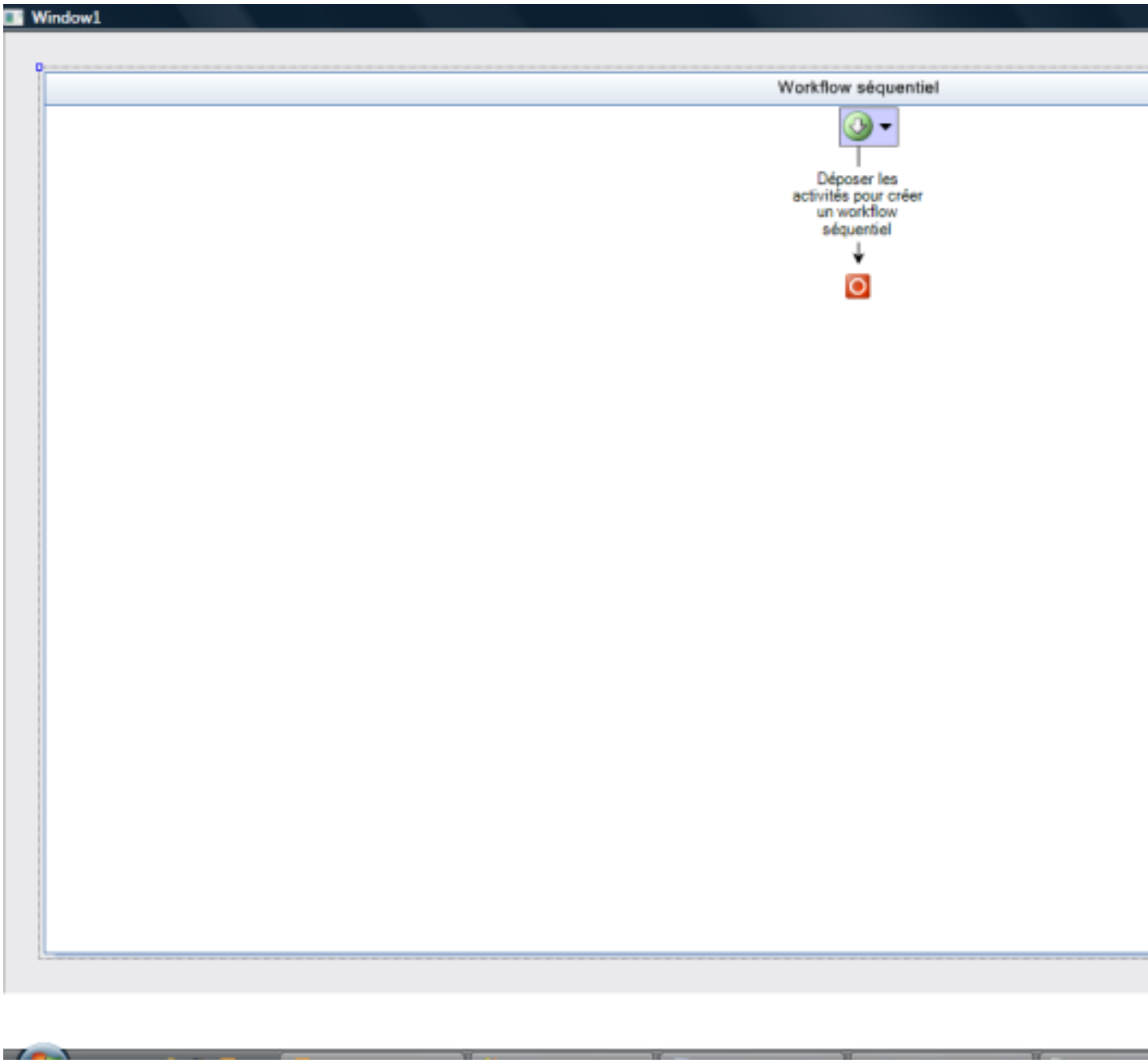
- Dans un premier temps une instance de WorkflowLoader est créée ainsi qu'une instance de DesignSurface.

L'instance de DesignSurface est ensuite utilisée pour charger le workflow. L'appel à la méthode BeginLoad avec notre loader en paramètre va déclencher un appel à la méthode void PerformLoad(IDesignerSerializationManager serializationManager)de notre loader.

Ensuite l'instance du designer est obtenue grâce au système de service. Nous vérifions que le rootDesigner est bien instancié et qu'il n'y a pas de problème lors de la création. Il se peut que le rootDesigner soit null si vous faites cet appel trop tôt et que les services ne sont pas en place.

Enfin pour terminer nous obtenons une instance de WorkflowView grâce au designer. C'est cette instance que nous allons afficher dans le WindowsFormHost.

A ce stade vous devez avoir une application qui ressemble à cela :



III-D - Ajouter des activités dans le designer

Maintenant que nous sommes capables de charger un designer, il ne nous reste plus qu'à pouvoir y ajouter des activités. Pour cela nous allons écrire une petite méthode AddActivity nous permettant d'ajouter une activité dans le designer.

Méthode AddActivity

```
public void AddActivity(Activity activity)
{
 if (activity == null)
  throw new ArgumentNullException("activity");

 IDesignerHost designerHost = designSurface.GetService(typeof(IDesignerHost)) as IDesignerHost;
 if (designerHost == null)
  return;
```

Méthode AddActivity

```
  SequentialWorkflowActivity rootActivity = this.workflowView.RootDesigner.Activity as
  SequentialWorkflowActivity;
  if (rootActivity == null)
   return;

  //Si l'activité que l'on ajoute n'a pas de parent on la lie à l'activité root du designer
  if (activity.Parent == null)
   rootActivity.Activities.Add(activity);
  //On ajoute la nouvelle activité dans le designer
  designerHost.Container.Add(activity);
 }
```

Expliquons un peu ce code :

Dans un premier temps nous récupérons l'instance du designer et l'instance de l'activité racine.

Ensuite nous vérifions si la nouvelle activité possède déjà un parent et si elle n'en possède pas nous l'attachons à la racine.

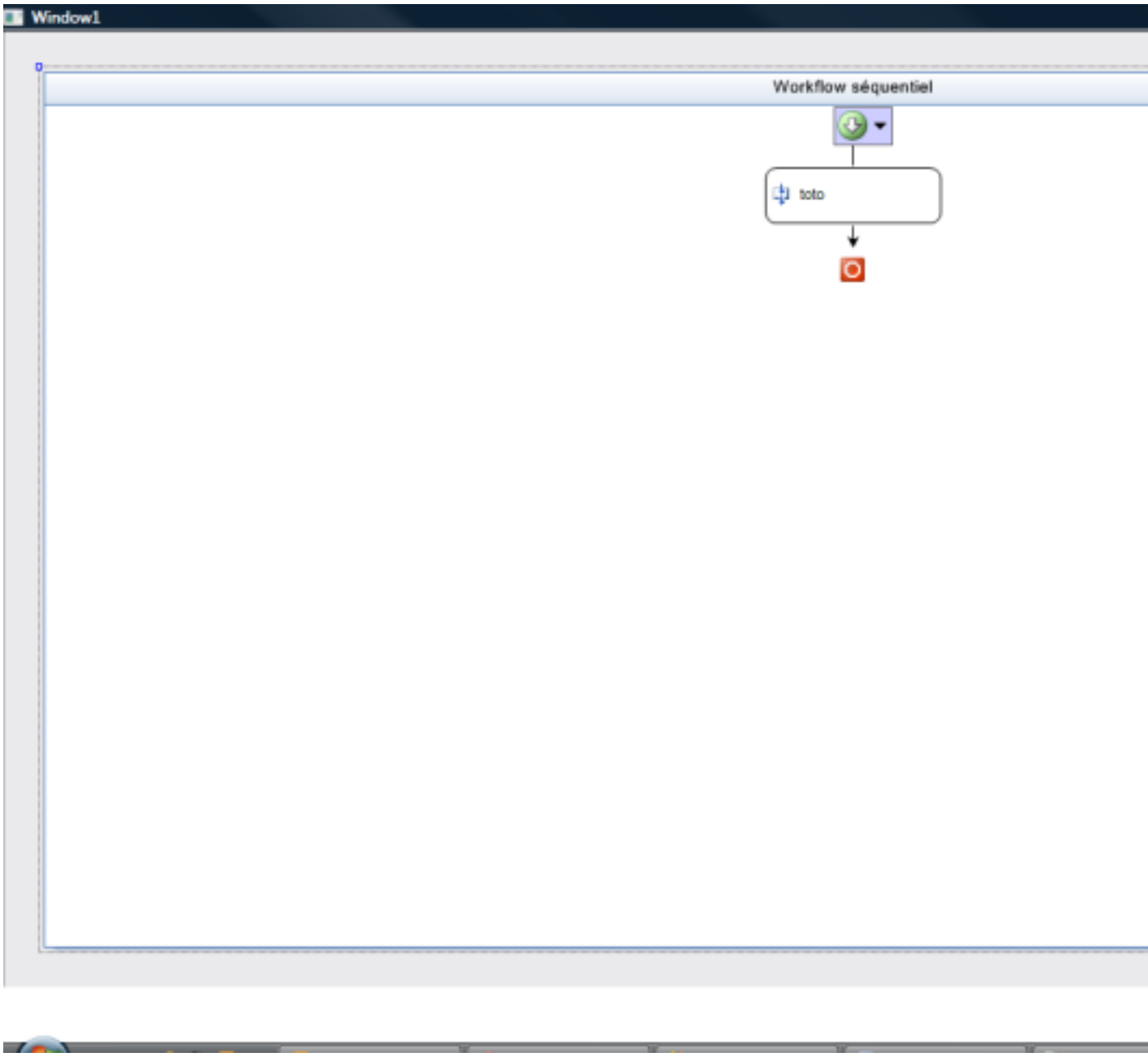
Pour finir il ne nous reste plus qu'à ajouter la nouvelle activité dans le designer.

Au niveau de l'utilisation, rien de particulier :

Ajout d'une activité dans le designer

```
private void button1_Click(object sender, RoutedEventArgs e)
{
 this.workflowDesigner.LoadWorflowDesigner();
 Activity activity = new System.Workflow.ComponentModel.Activity("toto");
 this.workflowDesigner.AddActivity(activity);
}
```

Et nous obtenons cela :



Si vous essayez d'ajouter une activité contenant d'autres activités vous remarquerez que les activités enfants ne sont pas visible dans le designer. En effet notre code ne fait pas automatiquement l'ajout des activités enfants dans le designer et par défaut le designer ne le fait pas pour vous.

Voici donc la méthode AddActivity corrigé :

Méthode AddActivity récursive

```
public void AddActivity(Activity activity)
{
 ......
   AddActivityToDesigner(designerHost, activity);
}

private void AddActivityToDesigner(IDesignerHost designerHost, Activity activity)
{
 if (activity == null)
 return;
```

Méthode AddActivity récursive

```
designerHost.Container.Add(activity);
if (activity is CompositeActivity)
{
 foreach (Activity child in ((CompositeActivity)activity).Activities)
  AddActivityToDesigner(designerHost, child);
}
}
```

III-E - Activer le Drag&Drop d'activités dans notre designer

Si vous avez essayé de faire du drag&drop d'une activité ajoutée par code dans le designer, vous avez sûrement remarqué qu'elle disparaît purement et simplement. En effet à ce stade de l'application il manque un service pour permettre la prise en charge du drag&drop. Si vous configurez Visual Studio pour intercepter les exceptions du type System.Exception vous aurez le droit à un message vous indiquant qu'il manque un service de type ITypeProvider pour permettre le Drag&Drop. En effet la gestion du Drag&Drop passe par une sérialisation/déserialisation des activités « Drag&Dropé ».

Ajoutons ce service à notre designer sans plus tarder. Pour cela nous allons surcharger la méthode Initialize de la classe WorkflowLoader et nous en servir pour ajouter le service TypeProvider dans le designer :

Ajout de service au loader

```
protected override void Initialize()
{
 base.Initialize();

 IDesignerLoaderHost host = LoaderHost;
 if (host != null)
 {
  host.AddService(typeof(ITypeProvider), new TypeProvider(host));
 }
}
```

Maintenant que le service ITypeProvider est en place, nous pouvons drag&droper les activités contenues dans le designer.

Le but d'un designer est de permettre une conception visuelle. Il serait donc bien que notre designer possède une boite à outil permettant de drag&droper des activités externe.

III-F - Création de la toolbox

Une fois n'est pas coutume, notre toolbox ne va pas s'appuyer sur le mécanisme proposé par le framework pour gérer ces cas-là. En effet le mécanisme de toolbox du framework est un brin trop complexe et il existe une méthode plus simple pour arriver au résultat que nous souhaitons.

III-F-1 - Création du UserControl de toolbox

Notre UserControl représentant la toolbox sera simplement composé d'une listview. Les activités ajoutées à la toolbox seront stockées dans une ObservableCollection<Activity> qui elle-même sera bindée à la listView. Tout ceci n'est que la tuyauterie nécessaire à WPF.

Ce qui, dans un premier temps, nous donne cela :

UserControl de toolbox

```
public partial class WorkflowToolbox : UserControl
```

UserControl de toolbox

```
{
 private const string CF_DESIGNER = "CF_WINOEDESIGNERCOMPONENTS";
 private ObservableCollection<Activity> activities;
 private WorkflowDesigner designer;

 public WorkflowToolbox()
 {
     InitializeComponent();

     this.activities = new ObservableCollection<Activity>();
     this.listView.ItemsSource = this.activities;
     this.listView.MouseMove += new MouseEventHandler(listView_MouseMove);
 }

 public void AddActivity(Activity activity)
 {
     if (!this.activities.Contains(activity))
         this.activities.Add(activity);
 }
}
```

Maintenant il nous reste à activer le drag&drop de la listview vers le designer. Cela se fait en deux étapes. Dans un premier temps nous allons gérer l'évènement MouseMove de la listview.

Gestion de l'initialisation du Drag Drop

```
private void listView_MouseMove(object sender, MouseEventArgs e)
{
 if (e.LeftButton == MouseButtonState.Pressed)
 {
  Activity activity = this.listView.SelectedItem as Activity;
  if (activity == null)
   return;

  IDataObject dataObject = SerializeActivitiesToDataObject(designer, new Activity[] { activity });
  DragDrop.DoDragDrop(this.listView, dataObject, DragDropEffects.Copy | DragDropEffects.Move);
 }
}
```

Les deux principaux points d'intérêts de ce code son au niveau de la sérialisation de l'activité et au niveau de l'initialisation du Drag&Drop. La sérialisation de l'activité se fait grâce à une méthode spécifique que nous allons voir juste après. Elle renvoie un IDataObject qui est l'élément standard de transport d'information pour les opérations de Drag&Drop. L'initialisation de l'opération de Drag&Drop se fait grâce à la nouvelle classe DragDrop de WPF qui expose la méthode DoDragDrop bien connu des développeurs Winform. Celle-ci prend en paramètres la source du drag&drop, les données (notre fameux objet IDataObject) et enfin le mode possible pour le Drag&Drop. Dans le cas du designer de workflow il faut que le mode de Drag&Drop soit positionné à Copy et Move même si vous ne souhaitez pas gérer le déplacement. Sans cela vos opérations de Drag&Drop échoueront obligatoirement. Voyons maintenant le contenu de la mystérieuse fonction SerializeActivitiesToDataObject.

Fonction de sérialisation des activités

```
private static IDataObject SerializeActivitiesToDataObject(IServiceProvider serviceProvider,
 IEnumerable<Activity> activities)
{
    // get component serialization service
    ComponentSerializationService css =
 (ComponentSerializationService)serviceProvider.GetService(typeof(ComponentSerializationService));
    if (css == null)
        throw new InvalidOperationException("Component Serialization Service is missing.");


    // serialize all activities to the store
    SerializationStore store = css.CreateStore();
    using (store)
    {
        foreach (Activity activity in activities)
```

Fonction de sérialisation des activités

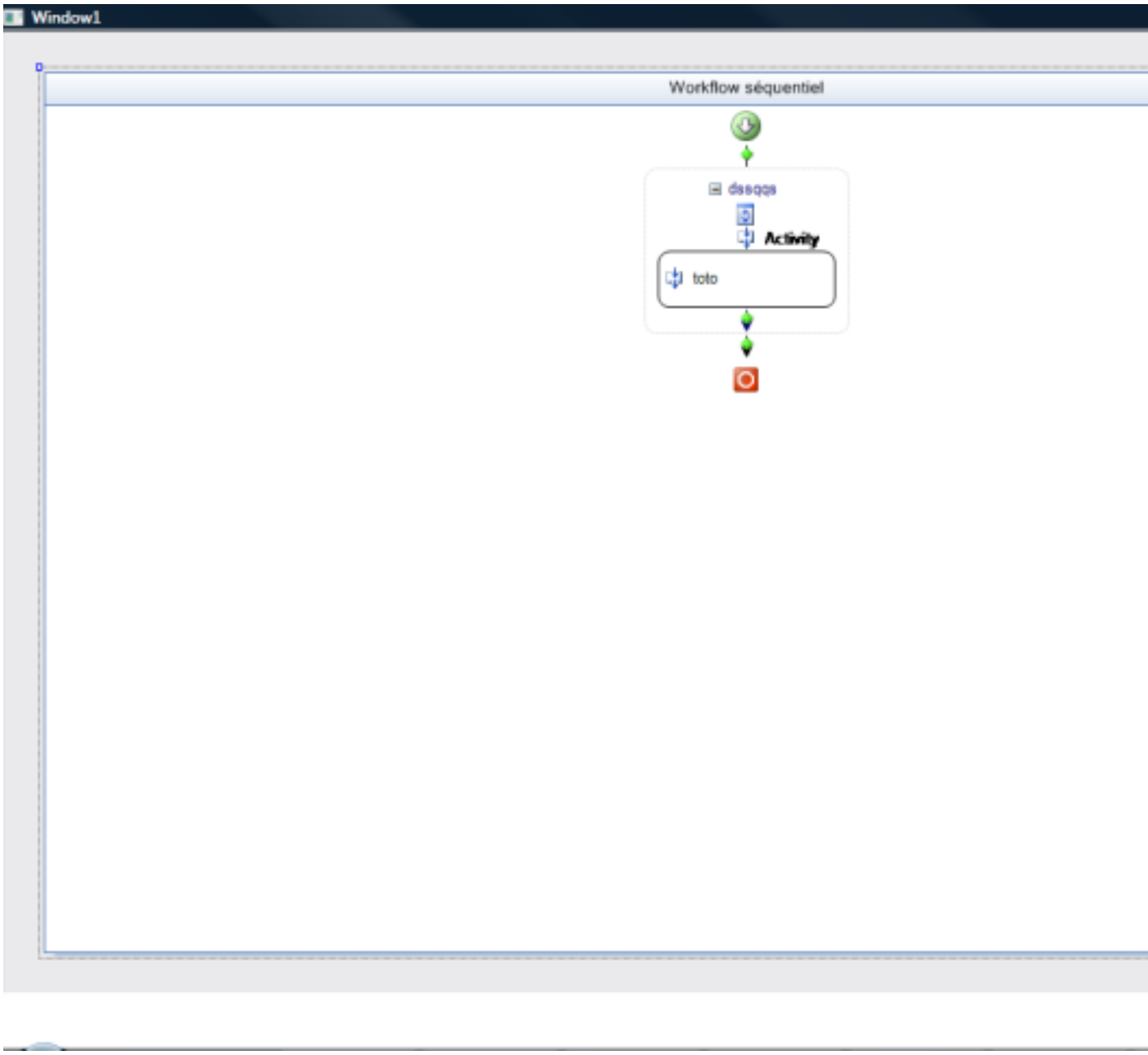
```
            css.Serialize(store, activity);
        }

        // wrap it with clipboard style object
        Stream stream = new MemoryStream();
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(stream, store);
        stream.Seek(0, SeekOrigin.Begin);


        return new DataObject(CF_DESIGNER, stream);
    }
```

Dans un premier temps nous obtenons le service de sérialisation utilisé par les designers du framework. Ensuite nous sérialisons les activités sélectionnés à l'aide de ce service dans un conteneur créé spécialement. Une fois les activités sérialisées dans le conteneur ce dernier est lui-même sérialisé dans un flux binaire. Le tout est renvoyé dans un DataObject indiquant que son format est celui des designers. Toute cette mécanique est nécessaire non pas parce que nous sérialisons une activité mais pour que le designer sache comme extraire l'élément que nous lui transmettons.

A partir de là nous avons un mécanisme de Drag&Drop qui fonctionne entre la listview et notre designer de workflow :



Nous avons maintenant un designer qui possède les principales fonctionnalités que l'on peut attendre de lui : Ajout d'éléments par code et par Drag&Drop.

Toutefois il reste quelque éléments assez intéressant que nous allons voir maintenant.

III-G - Gestion des commandes du designer

Notre designer est opérationnel mais il serait sûrement intéressant de pouvoir le manipuler autrement que juste avec la souris et le drag&drop. Par exemple un menu contextuel serait le bienvenu.

III-G-1 - WorkflowMenuCommandService : notre service de gestion des commandes du designer

La première des choses à faire, c'est de créer un service qui va gérer le menu contextuel. Pour cela nous allons créer une classe WorkflowMenuCommandService qui va hériter de MenuCommandService. Cela vous nous permettre de nous concentrer sur le code important.

Classe de gestion des commandes du workflow

```
public class WorkflowMenuCommandService : MenuCommandService
{
 public WorkflowMenuCommandService(IServiceProvider serviceProvider)
 : base(serviceProvider)
 {
 }
}
```

Nous sommes maintenant en capacité d'intercepter les demandes d'exécution de commande et les demandes d'affichage pour le menu contextuel. Toutefois à l'heure actuelle nous n'affichons rien. Nous allons remédier à cela maintenant.

La première chose à faire est de surcharger la méthode ShowContextMenu afin d'y ajouter notre propre logique.

Fonction d'affichage du menu contextuel

```
public override void ShowContextMenu(CommandID menuID, int x, int y)
{
    base.ShowContextMenu(menuID, x, y);

    MenuItem[] menuItems = null;
    if (menuID == WorkflowMenuCommands.SelectionMenu)
    {
        menuItems = GetSelectionMenuItems();
    }

    if (menuItems != null)
    {
        ContextMenu contextMenu = new ContextMenu();
        contextMenu.MenuItems.AddRange(menuItems);

        WorkflowView workflowView = GetService(typeof(WorkflowView)) as WorkflowView;
        if (workflowView != null)
            contextMenu.Show(workflowView, workflowView.PointToClient(new Point(x, y)));
    }
}

private MenuItem[] GetSelectionMenuItems()
{
    Dictionary<CommandID, string> selectionCommands = new Dictionary<CommandID, string>();
    selectionCommands.Add(WorkflowMenuCommands.Cut, "Cut");
    selectionCommands.Add(WorkflowMenuCommands.Copy, "Copy");
    selectionCommands.Add(WorkflowMenuCommands.Paste, "Paste");
    selectionCommands.Add(WorkflowMenuCommands.Delete, "Delete");
    selectionCommands.Add(WorkflowMenuCommands.Expand, "Développer");
    selectionCommands.Add(WorkflowMenuCommands.Collapse, "Réduire");

    List<MenuItem> menuItems = new List<MenuItem>();

    foreach (CommandID id in selectionCommands.Keys)
    {
        MenuCommand command = FindCommand(id);

        if (command != null)
        {
            MenuItem menuItem = new MenuItem(selectionCommands[id], new EventHandler(OnMenuClicked));
            menuItem.Tag = command;
```

Fonction d'affichage du menu contextuel

```
            menuItems.Add(menuItem);
        }
    }

    return menuItems.Count != 0 ? menuItems.ToArray() : null;
}

private void OnMenuClicked(object sender, EventArgs e)
{
    MenuItem menuItem = sender as MenuItem;
    if (menuItem != null)
    {
        MenuCommand command = menuItem.Tag as MenuCommand;
        command.Invoke();
    }
}
```

La fonction GetSelectionMenu permet de créer un ensemble d'entrées standard pour gérer les éléments courant. Pour cela on utilise la classe WorkflowMenuCommands définie dans le framework et qui est expose les ID des commandes standard. La fonction importante suivante est FindCommand. Elle permet d'obtenir une instance de la classe MenuCommand qui permet ensuite d'invoquer la fonction de la commande.

Notez également que le simple fait d'avoir ajouté un service dédié à la gestion des commandes permet de rendre opérationnel le repli des activités. Tant que ce service n'est pas en place cela ne fonctionnait pas.

III-H - Imprimer le workflow

Avoir un joli workflow dans le designer c'est bien mais permettre à vos utilisateurs de l'imprimer c'est encore mieux.

Encore une fois c'est dans ces « petits » détails que l'on constate que le framework est bien fait. La classe WorkflowView expose une propriété de type PrintDocument nous permettant d'utiliser directement les contrôles de prévisualisation et d'impression sans passer par un long et fastidieux développement. Ce qui, couplé à notre gestion des commandes, nous donne :

Aperçu avant impression

```
WorkflowView workflowView = GetService(typeof(WorkflowView)) as WorkflowView;
if (workflowView == null)
    return;

PrintPreviewDialog ppd = new PrintPreviewDialog();
ppd.Document = workflowView.PrintDocument;

ppd.ShowDialog();
return;
```

III-I - Générer une assembly contenant le workflow designé

La dernière chose à faire afin de rendre notre projet complètement fonctionnel c'est de permettre la génération d'une assembly permettant d'utiliser le workflow designé.
Tout cela est possible grâce au mécanisme de CodeDOM du framework .NET. Le plus simple pour mettre en oeuvre la génération de code et/ou d'une assembly est de se servir de la méthode PerformFlush du loader.
Coté génération de code xaml tout est pris en charge par le framework grâce au service de serialisation **WorkflowMarkupSerializer**.
Le seul problème du générateur de code XAML est qu'il ne crée pas l'attribut x:Class nécessaire à la compilation avec le code C#. C'est pour cela que le fichier Xoml est ouvert à nouveau et que l'attribut x:Class est ajouté à la main. Ensuite il reste le code C# (ou VB.NET, ou même Delphi .NET) qui est généré également (quasiment) automatiquement.
La dernière étape est d'appeler MSBuild à l'aide du fichier TestActivityLibrary.csproj afin de générer l'assembly.

```
protected override void PerformFlush(IDesignerSerializationManager serializationManager)
{
 //base.PerformFlush(serializationManager);
 IDesignerHost host = (IDesignerHost)GetService(typeof(IDesignerHost));
    Activity rootActivity = host.RootComponent as Activity;

    if (host != null && host.RootComponent != null)
    {

        if (rootActivity != null)
        {
            XmlTextWriter xmlWriter = new XmlTextWriter(this.Xoml, Encoding.Default);
            try
            {
                WorkflowMarkupSerializer xomlSerializer = new WorkflowMarkupSerializer();
                xomlSerializer.Serialize(xmlWriter, rootActivity);
            }
            finally
            {
                xmlWriter.Close();
            }

            XmlDocument doc = new XmlDocument();
            doc.Load(this.Xoml);
            XmlAttribute att = doc.CreateAttribute("Class", "http://schemas.microsoft.com/winfx/2006/
xaml");
            att.Value = "foo.Workflow1";
            XmlNode node = doc.FirstChild;
            node.Attributes.Append(att);

            doc.Save(this.Xoml);
        }
    }

    string codeBesideFile = Path.Combine(Path.GetDirectoryName(this.Xoml),
 Path.GetFileNameWithoutExtension(this.Xoml) + ".xoml.cs");
    string libFile = Path.Combine(Path.GetDirectoryName(this.Xoml),
 Path.GetFileNameWithoutExtension(this.Xoml) + ".dll");

    //Création du générateur de code C#
    CSharpCodeProvider provider = new CSharpCodeProvider();

    //Création des options du générateur de code C#
    CodeGeneratorOptions options = new CodeGeneratorOptions();
    options.BlankLinesBetweenMembers = true;
    options.BracingStyle = "C";
    options.ElseOnClosing = false;
    options.IndentString = "    ";

    CodeCompileUnit codeCompileUnit = new CodeCompileUnit();
    codeCompileUnit.ReferencedAssemblies.Add("System.Drawing.dll");
    codeCompileUnit.ReferencedAssemblies.Add("System.dll");

    //Les lib du Fx 3.0 ne sont pas automatiquement reconnues par le framework :-(
    string programFilesPath = Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles);
    codeCompileUnit.ReferencedAssemblies.Add(
        programFilesPath + @"\Reference Assemblies\Microsoft\Framework
\v3.0\System.Workflow.Activities.dll");
    codeCompileUnit.ReferencedAssemblies.Add(
        programFilesPath + @"\Reference Assemblies\Microsoft\Framework
\v3.0\System.Workflow.ComponentModel.dll");
    codeCompileUnit.ReferencedAssemblies.Add(
        programFilesPath + @"\Reference Assemblies\Microsoft\Framework
\v3.0\System.Workflow.Runtime.dll");

    CodeNamespace ns = new CodeNamespace("foo");
    ns.Imports.Add(new CodeNamespaceImport("System"));
    ns.Imports.Add(new CodeNamespaceImport("System.Drawing"));
    ns.Imports.Add(new CodeNamespaceImport("System.Collections"));
    ns.Imports.Add(new CodeNamespaceImport("System.ComponentModel"));
    ns.Imports.Add(new CodeNamespaceImport("System.ComponentModel.Design"));
```

```
    ns.Imports.Add(new CodeNamespaceImport("System.Workflow.ComponentModel.Design"));
    ns.Imports.Add(new CodeNamespaceImport("System.Workflow.ComponentModel"));
    ns.Imports.Add(new CodeNamespaceImport("System.Workflow.ComponentModel.Serialization"));
    ns.Imports.Add(new CodeNamespaceImport("System.Workflow.ComponentModel.Compiler"));
    ns.Imports.Add(new CodeNamespaceImport("System.Workflow.Activities"));
    ns.Imports.Add(new CodeNamespaceImport("System.Workflow.ComponentModel"));
    ns.Imports.Add(new CodeNamespaceImport("System.Workflow.Runtime"));
    codeCompileUnit.Namespaces.Add(ns);

    CodeTypeDeclaration ctd = new CodeTypeDeclaration("Workflow1");
    ctd.BaseTypes.Add("System.Workflow.Activities.SequentialWorkflowActivity");
    ctd.IsPartial = true;
    ctd.Attributes = MemberAttributes.Public;
    ns.Types.Add(ctd);

    //Ecriture du code .NET du workflow
    using (StreamWriter writer = new StreamWriter(codeBesideFile))
    {
        provider.GenerateCodeFromCompileUnit(codeCompileUnit, writer, options);
        writer.Close();
    }

    Process process = new Process();
    process.StartInfo.FileName = @"C:\Windows\Microsoft.NET\Framework\v3.5\MSBuild.exe";
    process.StartInfo.Arguments = string.Format("/tv:3.5 {0}",
 Path.Combine(Path.GetDirectoryName(this.Xoml), "TestActivityLibrary.csproj"));
    process.StartInfo.RedirectStandardOutput = true;
    process.StartInfo.UseShellExecute = false;

    process.Start();
    process.WaitForExit();

    if (process.ExitCode != 0)
    {
        System.Windows.MessageBox.Show(process.StandardOutput.ReadToEnd());
    }
}
```

IV - Conclusion

En conclusion nous avons vu qu'il est relativement simple d'héberger le designer de workflow fourni par le framework .NET et d'accèder aux fonctionnalités utilisées par Visual Studio.
Cela doit vous permettre d'écrire des applications ayant une forte valeur ajoutée dans le sens où, ce n'est plus le développeur qui défini le workflow mais tout utilisateur à la possibilité d'en créer et/ou de modifier un workflow.

Remerciements

Je tiens à remercier toute l'équipe de la rubrique dotnet pour leurs aides.
Je remercie plus particulièrement **dourouc05** pour sa relecture orthographique.

VI - Téléchargements

Les sources de l'article