La programmation Windows, le retour Partie 2/5 : Utiliser WRL pour faire des composants WinRT

Nous vous proposons une série d'articles sur la pratique de la programmation Windows en C++. Ce nouvel article nous emmène sur le développement autour du Windows Runtime alias WinRT.

Christophe PICHAUD | .NET Rangers by Sogeti
Consultant sur les technologies Microsoft
christophepichaud@hotmail.com | www.windowscpp.net

Depuis la sortie de Windows 8 et des applications Metro-style Apps devenues Modern UI, il est possible de faire du C++/CX pour attaquer WinRT. Il est aussi possible de faire des composants WinRT. A partir de là, reste à répondre à la question suivante : doit-je faire du C++/CX ou bien du C++ ISO à la Bjarne Stroustrup. Moi, je dis qu'il faut mieux « builder on the metal ! » donc pas de surcouche qui génère du code pour nous comme C++/CX ; C++/CX est une macro C++ avec la syntaxique du C++ CLI. Attention, ce n'est pas du CLI et c'est bien du natif... ISO C++ va nous obliger à écrire un peu plus de code mais ce n'est pas bien grave. De toute façon, je n'aime pas la syntaxe des hat (^) donc voilà.

Créer un composant WinRT

Nous allons créer un composant WinRT avec tout ce qu'il faut pour que cela marche (ce n'est pas simple au début), puis nous allons ajouter à ce composant la possibilité de gérer des évènements. Pour commencer, créez une solution dans Visual Studio. Ajoutez un nouveau projet C++ de type « DLL Windows 8.1 » à la solution. Ajoutez un nouveau projet C# de type « Blank App Windows 8.1 ». Ce projet nous servira de client pour tester le composant que nous allons réaliser. Positionnez-vous dans le projet DLL. Ajouter un fichier module.cpp et collez-lui le code suivant dedans :

```
extern "C" HRESULT WINAPI DllGetActivationFactory(_In_ HSTRING activatibleClassId, _Deref
_out_ IActivationFactory** factory)
{
  auto &module = Microsoft::WRL::Module<Microsoft::WRL::InProc>::GetModule();
  return module.GetActivationFactory(activatibleClassId, factory);
}

extern "C" HRESULT WINAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, _Deref_out_
 LPVOID *ppv)
{
  auto &module = Microsoft::WRL::Module<Microsoft::WRL::InProc>::GetModule();
  return module.GetClassObject(rclsid, riid, ppv);
}

extern "C" BOOL WINAPI DllMain(_In_opt_ HINSTANCE, DWORD, _In_opt_ LPVOID)
{
  return TRUE;
}

extern "C" HRESULT WINAPI DllCanUnloadNow()
{
  const auto &module = Microsoft::WRL::Module<Microsoft::WRL::InProc>::GetModule();
  return module.GetObjectCount() == 0 ? S_OK : S_FALSE;
}

#pragma comment(linker, "/EXPORT:DllGetActivationFactory=_DllGetActivationFactory
@8,PRIVATE")
#pragma comment(linker, "/EXPORT:DllGetClassObject=_DllGetClassObject@12,PRIVATE")
#pragma comment(linker, "/EXPORT:DllCanUnloadNow=_DllCanUnloadNow@0,PRIVATE")
```

Il s'agit ici de l'implémentation des fonctions DllMain, DllCanUnloadNow, DllGetClassObject et DllGetActivationFactory. DllMain est le point d'entrée dans la DLL ; on ne fait rien. Les autres fonctions sont la glue nécessaire à l'enregistrement du composant WinRT qui n'est autre qu'un composant COM de nouvelle génération.
Le code de DllGetActivationFactory, DllgetClassObject et DllCanUnloadNow font appel au template Module qui est définit dans <wrl\module.h>.

Un composant COM de nouvelle génération

La différence entre les anciens composants COM et les nouveaux réside dans le fait que les nouveaux s'enregistrent dans un package. Il semble qu'il n'y ait plus de traces dans la base de registres pour les composants. À suivre. Une fois que vous avez mis la glue COM dans votre DLL, on va pouvoir s'attaquer à la conception du composant. Nous savons qu'un composant WinRT est un composant COM donc il nous faut fournir un fichier IDL qui décrit son interface et sa classe d'implémentation.

```
// Library3.IDL
import "inspectable.idl";
import "Windows.Foundation.idl";

#define COMPONENT_VERSION 1.0

namespace Library3
{
  interface IGame;
  runtimeclass Game;

  [uuid(1FCD374B-2C3C-49E3-93A7-6FB801080D45), version(COMPONENT_VERSION)]
  delegate HRESULT GameEventHandler([in] HSTRING e);

  [uuid(3EC4B4D6-14A6-4D0D-BB96-31DA25224A15), version(COMPONENT_VERSION)]
  interface IGame : IInspectable
  {
      [propget] HRESULT RegisterUser([out, retval] HSTRING* value);
      [propput] HRESULT RegisterUser([in] HSTRING value);
      HRESULT StartGame([in] HSTRING value);
      [eventadd] HRESULT UserLogged([in] GameEventHandler* handler, [out][retval] Event
RegistrationToken* token);
      [eventremove] HRESULT UserLogged([in] EventRegistrationToken token);
```

```
    }

    [version(COMPONENT_VERSION), activatable(COMPONENT_VERSION)]
    runtimeclass Game
    {
        [default] interface IGame;
    }
}
```

Ce composant Game fournit au travers de sa déclaration IDL et son interface les éléments suivants :

Composant Game	
GameEventHandler	le delegate pour gérer les évènements
RegisterUser	propriété Get
RegisterUser	propriété Put
StartGame	Méthode qui utilise le delegate
UserLogged	Méthode pour s'abonner aux évènements
UserLogged	Méthode pour ne plus s'abonner aux évènements

Le fichier IDL contient la définition de l'interface IGame et du composant qui va implémenter cette interface : Game. Game est décoré en tant que runtimeclass. Anciennement, c'était coclass. Les propriétés sont décorées propget et propput. Vous remarquerez que la définition d'un évènement est spéciale. Il faut ajouter [eventadd] et [eventremove] et surtout gérer le « token » de type EventRegistrationToken.
Maintenant que la glue est prête et que les déclarations IDL sont faites..., la compilation du fichier Library3.idl génère les fichiers Library3_h.h, Library3_i.c et Library3_p.c. Nous allons utiliser le premier fichier (header) et le reste c'est le code du proxy stub. C'est du COM classique. A noter que l'interface IGame doit hériter de IInspectable qui est la nouvelle interface COM apparue avec Windows 8.

Implémentation de l'interface

```
#pragma once

#include "Library3_h.h"

namespace ABI
{
  namespace Library3
  {
    class Game : public RuntimeClass<IGame>
    {
      InspectableClass(L"Library3.Game", BaseTrust)

    public:
      Game();
      STDMETHOD(get_RegisterUser)(HSTRING* value);
      STDMETHOD(put_RegisterUser)(HSTRING value);
      STDMETHOD(StartGame)(HSTRING value);
      STDMETHOD(add_UserLogged)(IGameEventHandler* handler, EventRegistration
Token* token);
      STDMETHOD(remove_UserLogged)(EventRegistrationToken token);

    private:
      void Notify(HSTRING value);

    private:
      std::wstring name;
      bool m_observed;
      EventSource<IGameEventHandler> m_events;
    };

    ActivatableClass(Game);
  }
}
```

Première remarque, il faut que le composant soit défini dans le namespace ABI. C'est Microsoft qui impose cette règle. Le composant Game hérite du template RuntimeClass qui prend en paramètre l'interface à implémenter. Il y a ensuite deux macros à utiliser : InspectableClass et ActivatableClass. Le code de ces deux macros est disponible dans wrl\Implements.h et wrl\module.h. Le composant Game est un composant COM classique. Les propriétés sont préfixées de get_ et put_.

```
    STDMETHODIMP Game::get_RegisterUser(HSTRING* value)
    {
      HString str;
      str.Set(name.c_str());
      *value = str.Detach();
      return S_OK;
    }

    STDMETHODIMP Game::put_RegisterUser(HSTRING value)
    {
      HString str;
      str.Set(value);
      name = str.GetRawBuffer(nullptr);
      return S_OK;
    }
```

Les chaînes de caractères sont des HSTRING. La classe HString les encapsule. Dans cet exemple, je stocke la HSTRING dans un std::wstring.
Les évènements sont préfixés par add_ et remove_. La gestion des évènements se fait au travers de l'interface du delegate (IGameEventHandler) et un token de type EventRegistrationToken. Les événements sont stockés via le template EventSource<T> là où T est une interface du delegate définie dans le fichier IDL.

```
    STDMETHODIMP Game::add_UserLogged(IGameEventHandler* handler, EventRegistration
Token* token)
    {
      m_observed = true;
      m_events.Add(handler, token);
      return S_OK;
    }

    STDMETHODIMP Game::remove_UserLogged(EventRegistrationToken token)
    {
      m_events.Remove(token);
      return S_OK;
    }
```

Comment ça marche ?
Le membre m_events est utilisé avec ses méthodes Add et Remove. Dans le add_, on ajoute le handler en lui passant le token aussi. Dans le remove_, on supprime le token. C'est tout ! Il nous reste à voir comment déclencher

l'évènement. C'est la méthode StartGame qui va s'en charger en faisant appel à la méthode Notify.

```
STDMETHODIMP Game::StartGame(HSTRING value)
{
    HString str;
    str.Set(value);
    std::wstring ws = str.GetRawBuffer(nullptr);
    Notify(value);
    return S_OK;
}

void Game::Notify(HSTRING value)
{
    if (m_observed)
    {
        HString str;
        str.Set(value);
        m_events.InvokeAll(str.Detach());
    }
}
```

Notify fait appel à la méthode InvokeAll en lui passant l'argument HSTRING désiré.

Le client C#

Le client est une C# App Blank XAML qui contient juste un bouton et une zone de texte pour le debug. Voici à quoi ressemble le code du client. On fera d'abord un « Add Reference » pour y ajouter la DLL. Il ne faudra pas oublier dans les propriétés du projet C++ de dire dans les paramètres du Linker de produire les Windows Metadata (fichier WINMD) sinon il est impossible de faire « Add References ». Fig.1 et 2.

```
private void btnGo_Click(object sender, RoutedEventArgs e)
{
  LogInfo("Go...");

  Game game = new Game();
  game.RegisterUser = "Christophe";
  string str = String.Format("Register User: {0}", game.RegisterUser);
  LogInfo(str);

  game.UserLogged += Game_UserLogged;

  game.StartGame("BattelField 2 Bad Company");

  game.UserLogged -= Game_UserLogged;
 }

 private void Game_UserLogged(string game)
 {
  string str = String.Format("Event Game_UserLogged {0}", game);
  LogInfo(str);
 }
}
```

La fonction Game_UserLogged est ajoutée dans la liste des handlers et reçoit les évènements. C'est très simple.
Voici la sortie de l'application XAML :
Go...
Register User: Christophe
Event Game_UserLogged BattelField 2 Bad Company

La création directe d'objet avec WRL

Ajoutons une couche sur notre composant Game et codons l'objet Tournament qui aura pour but de créer un objet Game. Voici les lignes à ajouter au fichier IDL :

```
interface ITournament;
runtimeclass Tournament;

[uuid(3EC4B4E7-14A6-4D0D-BB96-31DA25224A15), version(COMPONENT_VERSION)]
interface ITournament : IInspectable
{
    HRESULT StartTournament([out, retval] IGame** game);
```

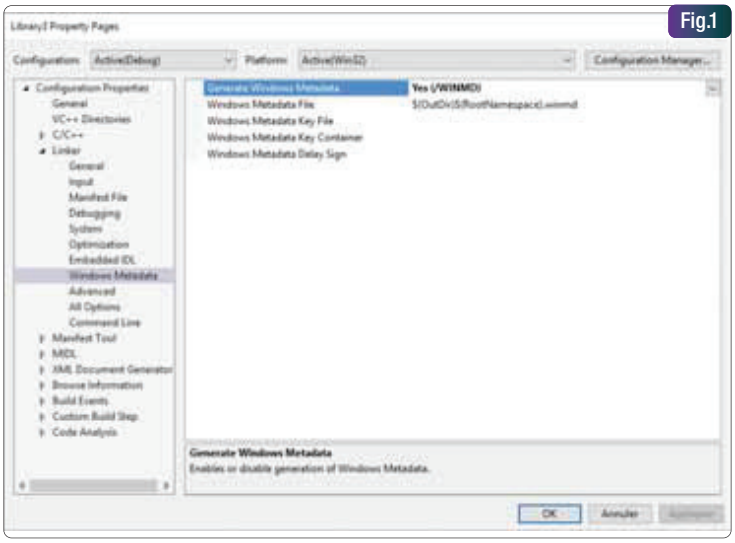
Fig.1

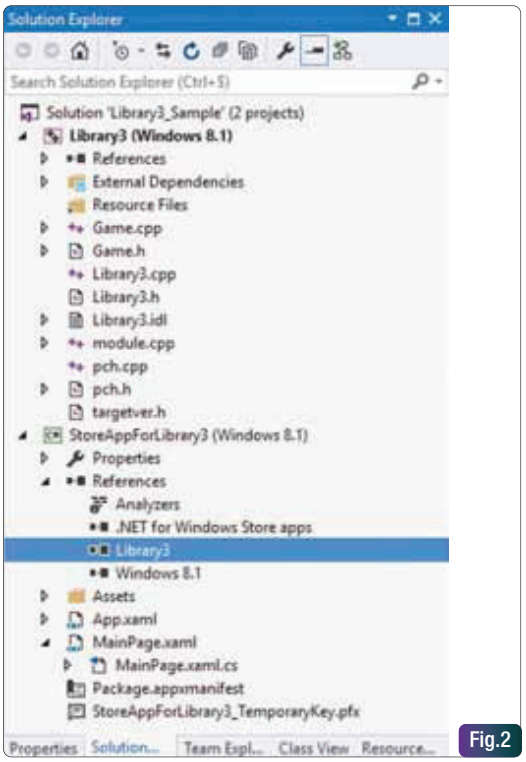

Fig.2

```
}

[version(COMPONENT_VERSION), activatable(COMPONENT_VERSION)]
runtimeclass Tournament
{
    [default] interface ITournament;
}
```

La méthode StartTournament présente dans l'interface ITournament retourne un objet Game au travers de son interface IGame. On échange et retourne des interfaces. Toujours. Jamais des composants. Par contre dans l'implémentation de la méthode, on va construire un composant et retourner l'objet. Voici le code du composant Tournament :

```
#pragma once

#include "Library3_h.h"

namespace ABI
{
  namespace Library3
  {
    class Tournament : public RuntimeClass<ITournament>
    {
      InspectableClass(L"Library3.Tournament", BaseTrust)

    public:
      Tournament();
      STDMETHOD(StartTournament)(IGame** game);

    };

    ActivatableClass(Tournament);
  }
}

#include "pch.h"
#include "Tournament.h"
#include "Game.h"

namespace ABI
{
  namespace Library3
  {

    Tournament::Tournament()
    {
    }

    STDMETHODIMP Tournament::StartTournament(IGame** game)
    {
      ComPtr<Game> p = Make<Game>();
      *game = p.Detach();
      return S_OK;
    }
  }
}
```

La méthode StartTournament crée un objet Game directement avec la fonction template Make<T> qui retourne un ComPtr<T>. Il faut faire un appel à Detach() pour retourner l'objet via son interface. Voici les modifications apportées au client C# pour obtenir un objet Game, ou plutôt une interface IGame :

```
private void btnGo_Click(object sender, RoutedEventArgs e)
{
  LogInfo("Go...");

  Tournament t = new Tournament();
  IGame game = t.StartTournament();

  //Game game = new Game();
  game.RegisterUser = "Christophe";
  string str = String.Format("Register User: {0}", game.RegisterUser);
  LogInfo(str);

  game.UserLogged += Game_UserLogged;

  game.StartGame("BattelField 2 Bad Company");

  game.UserLogged -= Game_UserLogged;
}
```

Conclusion

La création d'un composant WinRT ressemble fortement à la création d'un composant COM classique. Un fichier IDL avec une définition d'interface, une classe qui contient cette interface, et des évènements plus ou moins faciles à développer.

Tout ceci existe depuis Windows 8. WinRT est le nouveau COM et les composants exposent des méta-data sous forme de fichiers winmd au lieu de tlb. COM est de retour. Et ceci est vrai avec Windows 8.1 et Windows 10. WRL est l'équivalent de la nouvelle API ATL.

Le code source est disponible ici :

https://code.msdn.microsoft.com/windowsapps/A-Simple-Windows-Runtime-cd0095c9