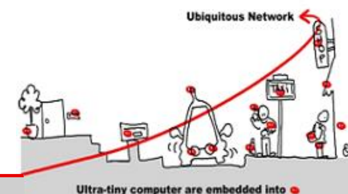
Tutorial 3: Event driven service Composition (LCA and SLCA Models)



1 Introduction to WComp

You can refer to the documentation available online as well as videos of demonstrations available for the installation and to use WComp plateform:

https://www.wcomp.fr/

WComp can be installed on various OS. Here we need any Windows® OS to install SharpDevelop and the WComp AddIn. You have to install SharpDevelop release 3.2.1.6466 in order to use SharpWComp 3.x.

See http://www.wcomp.fr/sharpwcomp3.

For the first use of SharpDevelop, you can choose your language in the menu “Outils / Options / Options de SharpDelevop/ Langue de l’utilisateur” in french, or “Tools / Options/ General / UI Language” in english.

The release to install is SharpWComp 3 for .NET 3.5. To install it, get the latest Release: 3.2.1.1390 - 15/11/2014 at the previous link.

Then, to test it, create a WComp Container:

- File / New → File…
- WComp.NET tab / C# Container item: creates a new file “Container1.cs” (tab at the top of the workspace)
- To manipulate the components, you must activate the graphical representation of the Container (WComp.NET tab at the bottom of the workspace).

Warning: Remember that you can only save your component assemblies using **Export** in the WComp.NET menu.

2 Web Service for Device Composition: Event Driven Orchestration

2.1 UPnP Tools Installation

If this is not done, download the open source version of UPnP tools “Intel® Tools for UPnP Technologies”:

http://opentools.homeip.net/dev-tools-for-upnp

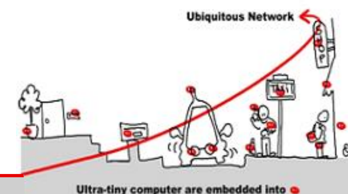
Run the “*Device Spy*” tool, which is a Universal Control Point (UCP). This tool allows discovering UPnP devices, performing action invocations and event subscriptions/notifications on any of them. Then run the virtual UPnP device called “*Network Light*”, verify it appeared in the UCP and test its functionalities. You can subscribe to events by right clicking on an *UPnP service* in the UCP.

2.2 Integration of a UPnP Web Service for Device in WComp

We want to access and manage UPnP devices in WComp. To achieve this, we must generate proxy components for each discovered UPnP device. Let’s generate it for the *Network Light*:

- File / New → File…
- WComp.NET / UPnP Device Web Service Proxy
- Select the Light in the device list and all the methods and state variables that you want to access via the proxy component (generally all of them). Click on Next and then on Finish. You’ve just generated a proxy component for this UPnP device.

Tutorial 3: Event driven service Composition (LCA and SLCA Models)



- Reload the available components list to be able to access this newly generated component (using the menu entry WComp.NET / Reload Beans…)
- Find the component in the category “Beans: UPnP Device” (Tools tab) and instantiate it in the container.

We will now study how this component can be linked to others in order to interact with the UPnP device.

Warning: verify that the IP address and the port number are same for the UPnP device and the proxy.

3 LCA Model for composition

3.1 Application Design using Event Driven Components Assemblies

3.1.1 Simple Events example :

To obtain the status of the virtual light in WComp we call “GetStatus” UPnP action. The proxy component is generated with an input port (a method) and the same name. We will use two components to call this method and display the result: a button and a checkbox. They are available in the “Windows Forms” category.

Exercice: Connect the button “Click” event to the “GetStatus” input port of the *Network Light* proxy component, and the “GetStatus_Return” event of the proxy to the “set_Checked” input port of the checkbox.

3.1.2 Complex Events

We now want to control the status of the *Network Light* with a checkbox inside WComp.

Create a checkbox and connect its “CheckedChanged” event to the “SetTarget” **incompatible** method (the method signature is not the same).

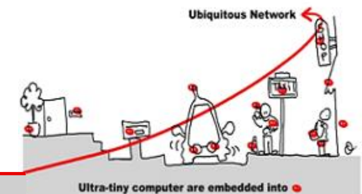
Exercice: From the call of the “SetTarget” method takes a boolean parameter “CheckedChanged” to provide the expected parameter to the method.

3.1.3 A more complete application

We will complete this application to become more familiar with the components.
If you double click on your virtual light, it turns it on or off. However, your checkbox does not reflect these changes until you click on the button to get its status.
Exercice :

- Find a way to reactively update the state of a new checkbox when the state of the UPnP device changes,
- Show the status of the *Network Light* in a text of a label or a textbox instead of the checkbox (using the “ValueFormatter” component in the “Beans: Basic” category),
- Connect the components needed to vocalize the state of the light (using components “BoolFilter”, “PrimitiveValueEmitter” and of course “TextToSpeech” in the “Beans: Services” category).

Tutorial 3: Event driven service Composition (LCA and SLCA Models)



3.2 Creating a Component

You may have to create your own components if existing components are not satisfying your needs. The SharpDevelopp allows to create a component from a termplate.

3.2.1 Create and use a simple component

After quitting and restarting SharpWComp, you can create a new solution for creating the desired component:

- File / New → Solution...
- WComp.NET / Wcomp Bean Solution.

Choose a name and path for your solution. Then add a new file to it (using the Projects tab):

- Right click on the project in the solution, and select Add → New Item...
- WComp.NET / C# Bean: creates a new file "Bean1.cs" or any name you chose,
- Update the "Beans" reference in the project, pointing it to the Beans.dll located in the AddIn installation directory, in SharpDevelop's files. You can now compile the bean template.

We will make a component that adds two integers. This component will have two methods: *intVal* and *intAdd* which allow you to specify an initial value and a second to add a new value. The last component emits an event which is the sum of both. We call this component *AddInt*.

Write the code, compile it and copy the created library to the component repository, located where you installed SharpDevelop, usually *C:\Program Files (x86)\SharpDevelop\3.X\Beans*. Reload the beans repository. The new component is now available in the specified category. The default category is "Beans: Basic".

Exercice : Make an assembly to test your component with two textboxes, using components named "StringToInt" and "ValueFormatter". Display the result in a label.

3.2.2 Create a component to blink a light

We will now create a component enabling a light to blink.

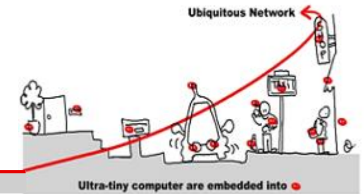
It will contain an integer property allowing to control the blinking period time in milliseconds. The blinking has to be made by a thread created in the component. The thread cannot be launched when the component is created, and you have to add an input port to the component to start it, and possibly stop it. The thread will be executing in a simple loop, called "`ThreadLoop`" in the example code below. It periodically sends a *true* then a *false* boolean event.

```
private Thread myThread;
private bool run = false;

public void SetBlinking(bool on){              // activating method
   if (on && !run) {
      run = true;
      myThread = new Thread(new ThreadStart(ThreadLoop));
      myThread.Start();
   } else if (!on && run) {
      run = false;
      myThread.Join();
   }
}

private void ThreadLoop(){                     // thread execution
```

Tutorial 3: Event driven service Composition (LCA and SLCA Models)




```
        while (run) {
            // TODO: emit true and false events
            Thread.Sleep(period);
        }
    }
```

Exercice : Finish the code of the bean, compile it and load it in the container. Instantiate it and connect it to a *Network Light*, and verify that it is blinking at the expected rate.

4 Dynamic composition at runtime

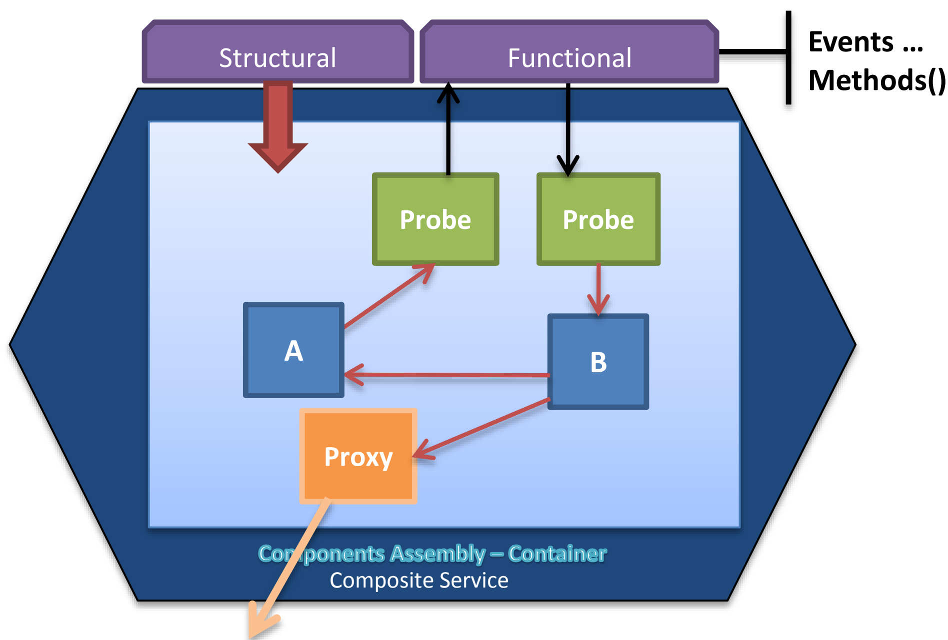
The aim of this tutorial is to create a simple adaptation mechanism for the WComp dynamic composition platform. You will first learn how to interact with containers in order to modify their current composition at runtime, using the Web Service for Device interfaces.

4.1 Using the Control Interface of a Composite UPnP Device

To familiarize with the control interface of the container, you will use the "Tools for UPnP Technologies", that you have to install from this website:

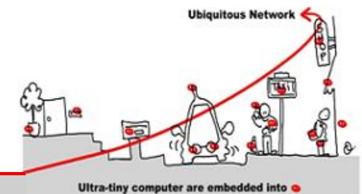
http://opentools.homeip.net/dev-tools-for-upnp

Launch the UPnP "*Device Spy*" tool, a WComp container, and activate the UPnP interfaces of your container using the "WComp.NET / Bind to UPnP Device" menu entry. Two new devices will appear in the "*Device Spy*": the control interface containing one service (`Container1_Structural_0` for example) and another one (`Container1_Functional_0`) containing no service.



V. Hourdin – S. Lavirotte – G. Rey – J.-Y. Tigli
Université de Nice – Sophia Antipolis
930, Route des Colles – B.P. 145 - 06903 Sophia Antipolis Cedex – France

Tutorial 3: Event driven service Composition (LCA and SLCA Models)



The control interface allows you to remotely modify the structure of the component assembly of the container. You can instantiate new beans, create new links, or destroy both. The control interface also allows the assembly of components to be displayed and manipulated by other tools than the built-in graphical designer of the SharpDevelop AddIn.

Exercice 1 : Create some beans (like an UPnP proxy on the Light UPnP service) in the container (the application), and invoke the "GetBeans" method on the control interface with the "*Device Spy*". That's a simple way of getting the list of the beans instantiated in the container and their type. You can also find the type of a bean when you select it, at the top of SharpDevelop's property tab ("GetBeansNames" only allow to get the bean names without their type).

Exercice 2 : Subscribe to container structural events (right click on the service in the "*Device Spy*") and try to create some components and links in the assembly using the control interface.

Exercice 3 : Try to create a button bean with the structural service of your container. Create a link to connect the light and the checkbox to allow switch on the light.

4.2 From the dynamic variation of the set of the available (ex. UPnP Wizard Designer)

To automatically generate and instantiate a proxy component on appearance of an UPnP device, we need to install a supplementary tools called "UPnP Wizard Designer".

You can find it at http://www.wcomp.fr/telechargement:v_3.2.1.1390.

You will now introduce an example of designer that automatically instantiates UPnP proxy components when UPnP *devices* are discovered on the network: the UPnP Wizard Designer. It should already been installed and available with a shortcut on your desktop.

Launch the designer, and start the UPnP interfaces of a SharpWComp container. At this time, if you launch an UPnP device like the "*Network Light*", the proxy component will be automatically generated, compiled, loaded in the container, and instantiated with the correct URI property. The proxy component will also be removed when the device disappears.

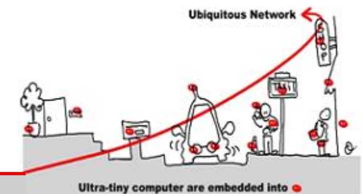
You have to define with which container you want to connect this tool by selecting the right one in the Connect menu. You can also filter some device type to avoid to automatically create beans from every kind of device.

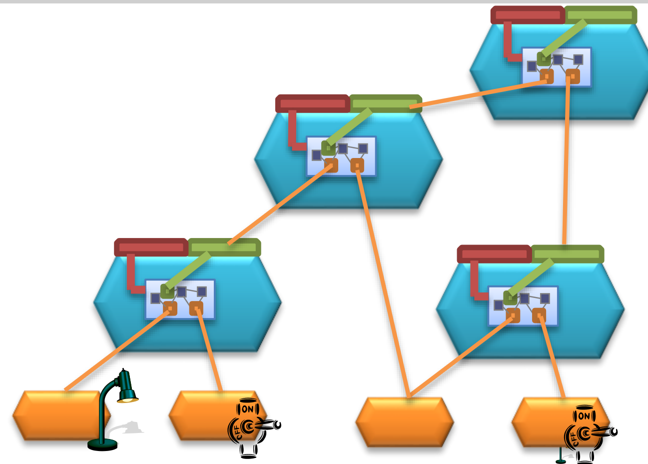
Exercice 4 : Remove all the beans in your container. Start the UPnPWizardDesigner, filter the unwanted type of UPnP device and start you UPnP Light. You should obtain an UPnP proxy bean instantiated in your container, corresponding to this UPnP device.

5 Creating a Composite Service: toward an Event driven Choregraphy

We know now how to use the control interface of WComp containers to remotely manage their internal assembly of components. A second Web service for device interface is provided by containers, the *functional interface*. It allows to export the functionalities created by the component assembly as a new (composite) Web service for device. Functionalities can then be distributed as a net of containers, as you can see below.

Tutorial 3: Event driven service Composition (LCA and SLCA Models)





Two specific types of components are used to manipulate the functional interface, called probe components. The “EmitProbe” component adds an action (method) to the UPnP functional interface of the container, and when invoked, an event with the same argument is emitted in the assembly of components. Conversely, the “SourceProbe” component adds an evented variable to the UPnP functional interface of the container, and this event is emitted when the method in the component assembly is invoked. These two components can be found in the “UPnP Probes” category in the tools tab.

The most basic example using two probes is to create an EmitProbe and a SourceProbe and connecting the event of the first to the method of the second. The resulting application will be a simple UPnP relay: by invoking the “FireEvent(string)” action on the functional interface, an event will be immediately sent on the sourceProbe1 event of the other service of the functional interface. Note that it can also be done with only one EmitProbe, since this probe provides the capability of sending an event corresponding to a return value of the invoked action.

The goal of this interface is to export functionalities created as an assembly of components into a new service, which can be later reused as any service provided by a device. To illustrate that, take the designer and the assembly you created in exercise 1.3, and replace the checkbox by an EmitProbe to provide a UPnP action that will switch on all lights. You should now be able to control the light with the Device Spy and in fact with any other WComp container instantiating a proxy on your first container service.

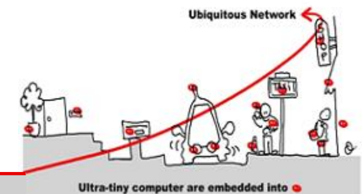
6 Advanced Tutorial: Build Components Using a Native DLL

Middleware for Ubiquitous Computing often need to interact with devices. In some cases, the managed framework (ex.Net Framework) doesn’t provide the corresponding interfaces to the devices, and native libraries are required to interact with the corresponding low level inputs/ouputs.

Create a new component in your project, and complete its code to provide methods that will call a native library. As an example device interaction, you can use the internal speaker of your computer, provided by the kernel32.dll native library. Native libraries can also provide purely software functionalities, like the workstation locking in Windows. Below is the code required to import these two DLLs and functions:

```
using System;
using WComp.Beans;
using System.Runtime.InteropServices;
```

Tutorial 3: Event driven service Composition (LCA and SLCA Models)




```
namespace WComp.Beans
{
  [Bean]
  public class BeanWin32
  {
        // import DLLs and methods
        [DllImport("kernel32.dll")]
        public static extern bool Beep(UInt32 frequency, UInt32 duration);

        [DllImport("user32.dll")]
        public static extern bool LockWorkStation();
        ...
  }
}
```

As can be seen in this code, two things are required to use a native library:

1. import the namespace “System.Runtime.InteropServices” that supports DllImport,
2. use the DllImport attribute to import a method from the native DLL, possibly specifying the path of the DLL if it is not a system DLL.

Methods are then simply called as regular C# methods. Compile and add your bean to the repository, instantiate it and test it. You can for example use a trackbar to change the pitch of the beep.

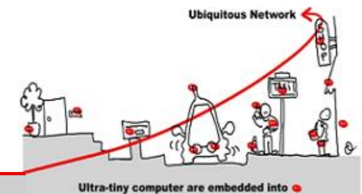
7 Advanced Tutorial: Build Components Using Unsafe Code

C# is not only able to execute managed code and invoke native libraries like we just saw, but can also execute non-managed code, similar to C-style pointer-based code. Benefits are the same than in low-level languages, a higher control over the execution of the code and best performance of execution. Moreover, it allows handling native DLL invocations requiring pointers, which is often the case.
A major problem with the use of pointers in C# is that a background garbage collection (GC) process is operated. When freeing up memory, this GC is liable to change the memory location of a current object without warning. A pointer previously pointing to that object would thus become a dangling reference, and the object will be dereferenced. Such a scenario leads to two potential problems. Firstly, it could compromise the execution of the C# program itself. Secondly, it could affect the integrity of other programs. To address this issue, the `fixed' C# keyword allows to specify that an object will be kept at the same memory address, preventing the GC to move it.
Because of these problems, the use of pointers is restricted to code which is explicitly marked by the programmer as `unsafe', in a block. Because of the potential for malicious use of unsafe code, programs which contain unsafe code will only run if they have been given full trust.

Below is a sample code of a method applying a simple XOR operation on a character string, using byte array of pointers to do it faster than managed code. This method will be seen as an input port of a bean, and its result, the processed string, will be sent as an event.

```
public unsafe void FastXOR(string str)
{
      System.Text.ASCIIEncoding encoding = new System.Text.ASCIIEncoding();
      byte[] managedBuf = encoding.GetBytes(str);
      int size = managedBuf.Length;
      fixed (byte* fixedBuf = managedBuf) {
```

Tutorial 3: Event driven service Composition (LCA and SLCA Models)




```
            byte* buf = fixedBuf;
            while (size >= 4) {
                  *((int*)buf) = *((int*)buf) ^ 123456789;
                  buf +=4;
                  size-=4;
            }
      }
      FireStringEvent(encoding.GetString(managedBuf));
}
```

Modify this code to take the XOR key as a property of the bean, and verify that the string value changes when the key changes. To compile it, you will need to set the “allow unsafe code” option in project’s properties.