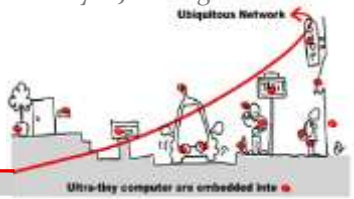
Tutorial 3 : Middleware for Ubiquitous Computing, WComp 2.4



1 Discovery of WComp

You can refer to the documentation available online as well as demonstration videos available for the installation and for taking the WComp middleware in hand:

https://www.wcomp.fr/download

You need to have a Windows® OS. Installing SharpDevelop and the WComp AddIn is mandatory and may be sufficient. At first run 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.

Then, 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).

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

2 Web Service for Device Composition

2.1 UPnP Tools Installation

Download the open source version of UPnP tools, formerly released as "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 a *UPnP service* in the UCP.

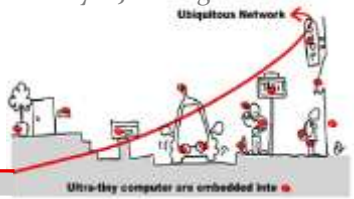
2.2 Integration of a UPnP Web Service for Device in WComp

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 WebService Proxy
- Select the Light in the device list, all 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.
- 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.

Tutorial 3 : Middleware for Ubiquitous Computing, WComp 2.4



3 LCA Model

3.1 Application Designing Using Components Assemblies

3.1.1 Simple Events

We will obtain the status of the virtual light in WComp by proceeding to the invocation of the "GetStatus" UPnP action. The proxy component was generated with an input port (a method) of the same name. We will use two components to proceed to this invocation and display its result: a button and a checkbox. They are available in the "Windows Forms" category. 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. Since the "SetTarget" method takes a boolean parameter and that the "CheckedChanged" has a different signature, they are not naturally compatible. The call has to be completed by providing a boolean information from the caller.

3.1.3 A more complete application

We will complete this application to become 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 unless you click on the button to get its status.

- Find a way to reactively update the state of a new checkbox when the state of the UPnP device changes,
- We want to show the status of the *Network Light* in text in a label or textbox instead of in a checkbox (consider 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).

3.2 Creating a Component

You may need to create your own components if existing components do not meet your needs. The SharpDevelope environment offers the opportunity to create a component from a skeleton.

3.2.1 Create and use a simple component

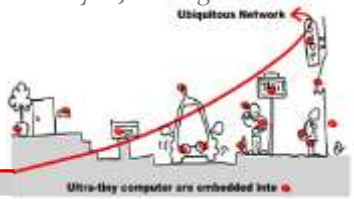
After quitting and restarting SharpWComp, you can create a new solution for creating the desired component:

- 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.

Tutorial 3 : Middleware for Ubiquitous Computing, WComp 2.4



We will make a component that adds two integers. This component will have two methods in entries: *intVal* and *intAdd* which allow you to specify an initial value and a second to add a new value. This component will emit 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\SharpDevelop\2.2\Beans*. Reload beans repository. Your new component is now available in the specificed category, the default being "Beans: Basic".

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 feature 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 a simple loop, called "`ThreadLoop`" in the example code below, that periodically sends a *true* then a *false* boolean event.

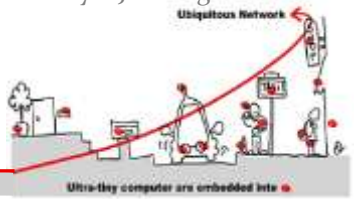
```
private Thread myThread;
private bool run = false;

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

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 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.

Tutorial 3 : Middleware for Ubiquitous Computing, WComp 2.4



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;

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.

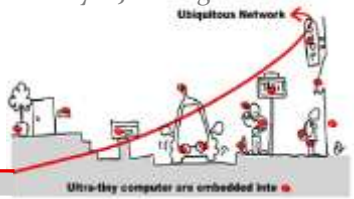
5 Build Components Using Unsafe Code

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.

Tutorial 3 : Middleware for Ubiquitous Computing, WComp 2.4



Below is a sample code of a method applying a simple XOR operation on a character string, using byte table pointers to do it faster than with 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) {
            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.