# Tutorial 5: Web Service for Device Event-Driven Composition

## 1 Web Service for Device

### 1.1 UPnP Tools Installation

Download and install the open source version of UPnP tools, formerly released as "*Developper Tools for UPnP Technologies*":

http://www.meshcommander.com/upnptools

You can also access useful videos on the use of UPnP.

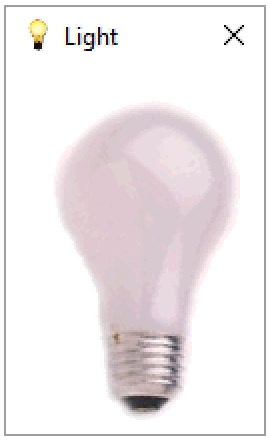
http://trolen.polytech.unice.fr/cours/mit/videos/

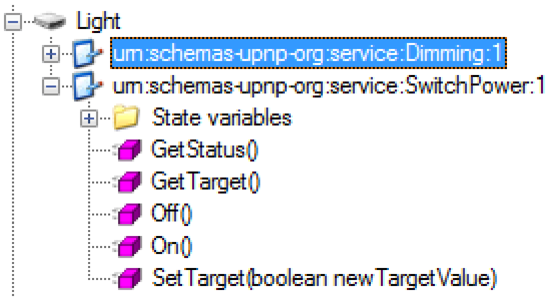
Run the "*Device Spy*" tool, which is a Universal Control Point (UCP). This tool allows discovering UPnP devices, performing action invocations and event subscriptions/notifications on any of them.
You also can download the following package to get some useful tools for testing purposes.

http://trolen.polytech.unice.fr/cours/mit/

### 1.2 UPnP Tools contract exploration

Run the virtual UPnP device (provided in the last downloaded package) called "*Light*", and verify it appears in the UCP. When you expand the device functionalities, you will discover the contract provided by this UPnP Device. This one offers 2 services: a *Dimming* and a *SwitchPower* service. If you expand the *SwitchPower* service, you can watch the offered methods which you can call and, in directory State Variables, the events you can subscribe to. To subscribe to events by right clicking on a *UPnP service* in the UCP.
Try to test a method call on the light to switch it on and off and subscribe to the state variables. When you do a call on the On or Off method, you can watch the message sent to view the new value of the state variables.



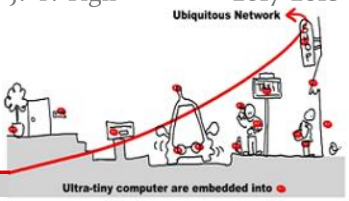

```
Light
  urn:schemas-upnp-org:service:Dimming:1
  urn:schemas-upnp-org:service:SwitchPower:1
    State variables
    GetStatus()
    GetTarget()
    Off()
    On()
    SetTarget(boolean newTargetValue)
```

### 1.3 Dynamic discovery

When you start a new UPnP service on the local network, you can discover it dynamically.

Try to launch the Switch UPnP Device (provided by the test tools package). The Device Spy detects it and you can also interact with it. Stop this program; it disappears from the list of available devices.

# Tutorial 5: Web Service for Device Event-Driven Composition



## 2 Web Service for Device Composition

### 2.1 WComp installation

You need to have a Microsoft Windows® OS (from Windows XP3 to Windows 10). First of all install SharpDevelop (release 3.2.1.6466) and then the WComp AddIn (WCompAA). You can download them from:

```
http://trolen.polytech.unice.fr/cours/mit/
```

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.

### 2.2 WComp discovery

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/
```

To create a WComp Container (a container contains a components assembly):

- File / New → File…
- WComp.NET tab / C# Container item: creates a new file "Container1.cs" (tab at the top of the workspace). We won't need this tab, but when you modify the graphical representation of the components assembly, the corresponding C# code is generated.
- 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 components assemblies using **Export…** in the WComp.NET menu. If you use the regular save functionality, it will save the C# source code of your assembly and it's often not what you want to achieve.

### 2.3 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 component** 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.

### 2.4 Dynamic discovery of devices in WComp

You can also take advantage of the UPnP discovery mechanism in WComp. To achieve this, you have to start a dedicated device spy program that connects to a WComp container. Each time a service if discovered, a proxy component is generated on the fly and instantiated in the WComp container. The following actions allow you to have a proxy component corresponding to each available UPnP device in WComp container:

# Tutorial 5: Web Service for Device Event-Driven Composition

- Start the UPnP Wizard Designer tool
- Activate the UPnP Interface of WComp (menu WComp.NET / Bind to UPnP Device)
- In UPnP Wizard Designer menu Connect, select the detected WComp container to connect to.

Each time a UPnP device appear or disappear, the corresponding proxy component is instantiated of destroyed. This process can be achieved with real devices of course:

http://www.dailymotion.com/video/x33g61m_ubicomp-2015_tech

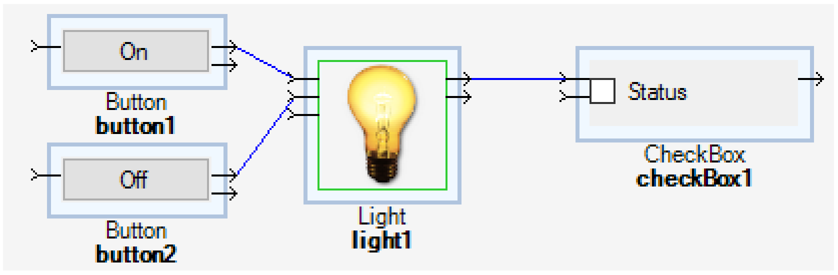
## 3 LCA Model

### 3.1 Application Designing Using Components Assemblies

#### 3.1.1 Simple events and method call

We will control the state of the virtual light in WComp by proceeding to the invocation of the "On" or "Off" 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: 2 winforms Button. They are available in the "Windows Forms" category. Connect the button "Click" event to the "On()" (and with the second button to the "Off()") input port of the *Light* proxy component. The Click event do not emit any value but the "On" and "Off value do not attempt any parameters.

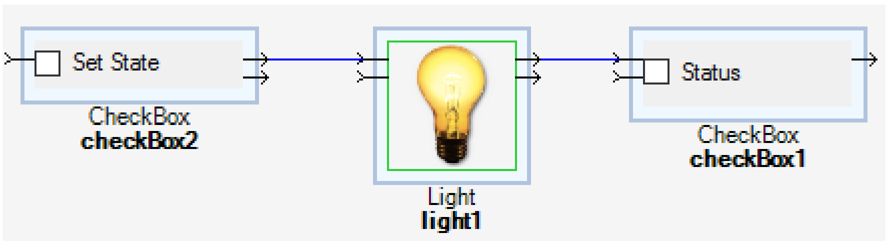
Then instantiate a checkbox Windows Forms component to display the current status of the Light. Connect the emitted event "Status_Event" to the "set_Checked(Boolean)" event. The "Status_Event" event emits the value of a Boolean corresponding to the light state.



#### 3.1.2 Complex Events

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

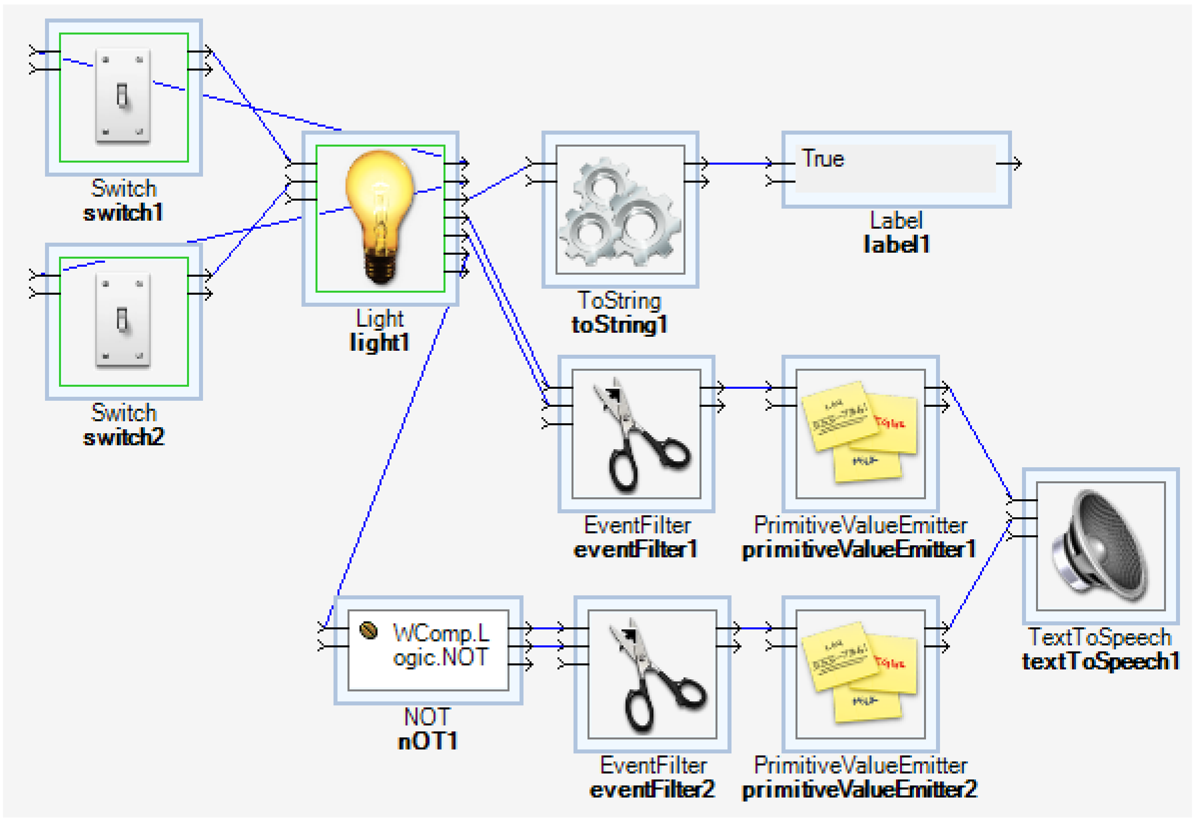
Create a checkbox and connect its "CheckedChanged" (which do not emit any value) event to the "SetTarget(Boolean)" **incompatible** method. Since the "SetTarget" method takes a boolean parameter and that the "CheckedChanged" do not emit any value, they have a different signature, they are not naturally compatible. The call has to be completed by providing a boolean information from the caller. We have to call a getter method to give that parameter: "get_Checked()".



### 3.1.3 A more complete application

You can of course connect different UPnP Devices together. For example, try to interconnect the Switch device to the Light device to control the light by the switch. Try to add a second switch to control the light. Modify your assembly in order than when you activate a switch, the other one will receive the light status to modify its position. On the other hand, if you double click on your virtual light, it turns it on or off (simulating a third party interaction) and the switch state should be also modified if you correctly did it previously.
We will complete this application to become familiar with the components.

- We want to show the status of the *Light* in text in a label or textbox instead of in a checkbox (consider using the "ToString" component in the "Beans: Basic" category),
- Connect the components needed to vocalize the state of the light (using several component in Basic tab like "EventFilter", "PrimitiveValueEmitter", "Not" and of course the "TextToSpeech" component in the "Beans: Services" category).



## 3.2 Creating a Component

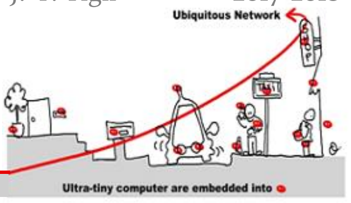
You may need to create your own components if existing components do not meet your needs. The SharpDevelop environment offers the opportunity to create a component from a skeleton.

### 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):



# Tutorial 5: Web Service for Device Event-Driven Composition

- 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 (*C:\Program Files (x86)\SharpDevelop\3.0\AddIns\WCompAddIn\Beans.dll*), 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 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 (x86)\SharpDevelop\3.0\Beans\*. Reload beans repository. Your new component is now available in the specified 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 threaded component to blink a light

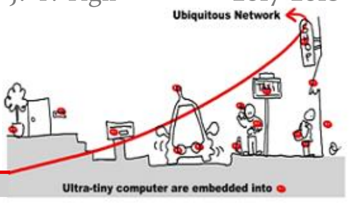
We will now create a component enabling a light to blink.
It will feature an integer property allowing to control the blinking period time in milliseconds. The blinking has to be made by a thread created in the component. The thread cannot be launched when the component is created, and you have to add 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.

```
public class Blink : IThreadCreator {
    private Thread t;      // Private attributes of the class
    private int sleepVal = 500;
    private volatile bool run = false;

    public void Start() {  // Starting the thread
        if (!run) {
            run = true;
            t = new Thread(new ThreadStart(ThreadLoopMethod));
            t.Start();
        }
    }
    public void Stop() {   // IThreadCreator defines the Stop() method
        run = false;
    }

    // Loop sample
    public void ThreadLoopMethod() {
        while(run) {
            // TODO: emit events
            Thread.Sleep(sleepVal);
        }
    }
}
```

# Tutorial 5: Web Service for Device Event-Driven Composition



Finish the bean code, compile it and load it in the container. Instantiate it and connect it to a *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/outputs.

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.

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) {
             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.