國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

基於工作流程與通用隨插即用技術下

之整合性軟體開發架構

A Development Framework for Composable Software Systems

based on Workflow and UPnP Technology

研 究 生：邱博政

指導教授：邵家健 博士

中 華 民 國 九 十 八 年 七 月

基於工作流程與通用隨插即用技術下之整合性軟體開發架構

A Development Framework for Composable Software Systems based on Workflow and UPnP Technology

研 究 生：邱博政　　Student：Po-Cheng Chiu

指導教授：邵家健　博士　　Advisor：Dr. John Kar-Kin Zao

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

July 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年七月

基於工作流程與通用隨插即用技術下

之整合性軟體開發架構

學生：邱博政　　指導教授：邵家健 博士

國立交通大學資訊科學與工程研究所　碩士班

摘要

由中研院主持的 SISARL 計畫—「老年人居家照護」是交通大學遍佈式嵌入系統實驗室(Pervasive Embedded System Lab)近幾年的重點研究計畫之一，其旨在藉由逐步增添智能家電的過程促使老年人所居住的環境有所進化，協助老年人能在自己熟悉的居住環境安享其老年生活。轉化過程中，為了方便家電開發商設計能與環境內裝置立即作溝通反應的系統，此研究訂立一套要求具有下列特性的整合性軟體開發架構：(1) 可整合化開發架構能結合具有相同架構的裝置及服務，形成功能更多的反應系統；(2) 可重複使用的功能函式庫讓已開發完成的裝置服務能重複被建立使用；(3) 人性化的設計介面與工具能讓開發者快速地進行程式開發；(4) 簡易配置環境裝置使得架構能輕易地連結環境裝置。全篇論文內以 Windows Workflow Foundation 技術整合封裝 UPnP 階層裝置，將架構的運作流程與裝置的部分作結合溝通，並設計一整套運作流程的軟體開發方式，使得開發者能在智能環境中開發一套與智能環境中家電裝置溝通的即時反應系統(Real-time Reactive System)，並且能讓多個智能家電裝置之功能得以彼此之間互相溝通、協調各自之運作。最後，此架構整合論文「適用於家庭自動化的通用隨插即用感測與促動器基礎架構」在交通大學電資大樓智能環境實驗室中實作完成室內燈光回饋控制，並封裝階層式燈光控制元件以利於重複使用。另外也完成了智慧型置物櫃。而未來以此軟體開發架構開發出的智能家電會越來越多，逐步走進智能家庭的世代。

A Development Framework for Composable Software Systems based on Workflow and UPnP Technology

Student: Po-Cheng Chiu Advisor: Dr. John Kar-Kin Zao

Institute of Computer Science and Engineering
National Chiao Tung University

ABSTRACT

In the recent years, NCTU Pervasive Embedded System (PES) Lab have focused on an elder-care project named “Kannon”. This project aims at enabling elders to live comfortably in their familiar environments by gradually transforming those environments through acquisition of smart appliances. In this transforming process, the developer of smart appliances plays an important role. In the premature smart environment region, in order to let developers develop a reactive system which can communicate with environment devices easily, we construct a framework with the following four essential properties: (1) Development framework composability; (2) Functional libraries reusability; (3) User-friendly programming interfaces and tools; (4) Simply configure to environment devices. In this thesis, we encapsulate UPnP devices using Windows Workflow Foundation, and design a software development framework through abstracting the operations and devices of the framework. Developers can create a “Real-time Reactive System” and compose more appliances in a smart environment using the Workflow and UPnP technology. This software development framework has integrated the feedback system of indoor luminance from the thesis “UPnP Compatible Sensor/Actuator Infrastructure for Home Automation”, built in NCTU MIRC Smart Environment Lab to demonstrate the effectiveness of the proposed technology, and has encapsulated the libraries of hierarchical luminance control component in order to be reused. Besides from luminance application, smart pantry is another application of this framework. In the future, more and more smart appliances will be developed with this technology, entering into the smart home generation.

誌謝

首先要感謝的是邵家健老師，讓我在修習碩班的這兩年之間，得到許多知識以及做人處事的態度，並且在論文上指導方向、討論方法，順利的完成本論文。另外也特別感謝張韻詩教授及施吉昇教授擔任學生的畢業口試委員，給予了許多寶貴的意見，使得本篇論文更加完備。

在研究中，特別感謝勝焜以及梅瑛這兩位一起進行相同計畫的夥伴們，一起參與大大小小的會議，勝焜對於UPnP技術相當的瞭解，給予我很大的幫助。梅瑛，我的大學同學、研究所同學兼好朋友，時常聆聽我訴說研究上的煩惱並給我鼓勵及支持，感謝你們。

接著很感謝PES實驗室的同儕們：星閔、嘉錡、彥霖、勝焜、梅瑛，以及學弟妹們：子晉、俊維、志明、鈞凱、嘉瑜，能與你們一起在實驗室裡聊天、奮鬥，是我研究所最難忘的回憶。

此外，也感謝我的室友們、好朋友們以及在研究所無論是一起修課認識、或一起運動、遊戲認識，或是一起出去玩認識的朋友們，沒有你們，我的研究所生活就不會如此多采多姿。

最後要感謝一直在背後默默支持、照顧我的家人們，因為有了你們，我才能順利的完成碩士學位。

目錄

圖目錄

表格目錄

第一章 綜述

1.1 問題陳述

由中研院主持的SISARL[1]計畫－「老年人居家照護」是交通大學遍佈式嵌入系統實驗室(Pervasive Embedded System, PES Lab)近幾年的重點研究計畫之一，其旨在藉由逐步增添智能家電的過程促使老年人所居住的環境有所進化，協助老年人能在自己熟悉的居住環境安享其老年生活。在這樣的過程中，智能家電開發商扮演著很重要的角色。在尚未發展健全的智能環境領域內，由於沒有一套通用的開發智能產品技術，使得家電開發商要設計開發新的智能家電產品及整合各項智能家電所提供的服務具有一定的困難度。為了方便家電開發商設計能與環境內裝置立即作溝通反應的系統，此研究訂立一套要求具有下列特性的整合性軟體開發架購：

1. 可整合化開發架構(Composable Development Framework)－在一個智能環境內，會具有許許多多的即時性反應系統與自動化裝置服務，這些裝置系統彼此之間會互相影響。此開發架構具有可整合多個相同架構的系統與服務的特性，讓開發者在設計開發一個智能環境中的即時性反應系統時，能整合多個系統讓它們彼此可以互相地溝通協調，進而形成一個更大規模、服務功能更多的系統。
2. 可重複使用之功能函式庫(Reusable Functional Libraries)－當任何開發者完成一項即時性反應系統後，若能根據個別的功能將其封裝成一個函式庫，使得原系統內的即時反應裝置及服務輕易地在具有相同架構的新系統上被重複使用，對於未來開發智能環境整合型系統會有相當大的益處。
3. 人性化的程式設計介面與工具(User-friendly Programming Interfaces & Tools)－隨著現今科技日新月異的進步，程式設計開發不再完全是傳統一行一行的編寫程式碼，而是將會變得更加簡單，甚至是僅僅以拖曳的方式就能完成一個複雜的程式設計。這樣簡易快速且人性化的開發方式，在智能環境發展的領域裡，對於智能家電開發

[1] SISARL 為「健康銀髮族用的感測資訊系統及服務(Sensor Information Systems (Services) for Active Retirees and Assisted Living)」之簡稱。

商會是一項利多。此架構具有一個能以拖曳方式進行程式開發的設計使用介面，且能以圖形化的方式呈現，讓開發者更輕易且快速的設計一個智能家電系統，建立即時反應的自動化機能。

4. 可簡易配置的架構 (Simply Configuration Framework) －智能家電開發商在開發家電的過程中，並不著重於如何設定裝置溝通的協定及建立智能環境的橋樑。因此本軟體開發架構提供簡易的開發方式，使得此架構可以快速的與智能環境溝通，自動偵測找尋環境中的裝置進行配置並加以控制。

1.2 研究方法

為了在開發系統時能具備上述的特性，我們以微軟(Microsoft)的工作流程技術(Windows Workflow Foundation)整合封裝通用隨插即用(Universal Plug and Play，UPnP)抽象分層的架構。在智慧家庭中的自動化裝置與服務，是屬於一種如同Ptolemy Project[1] 中的即時性反應系統(Real-Time Reactive Systems)，而本研究即是為了方便開發者設計開發此系統，以下列三種軟體開發方式完成的開發架構：

- 軟體開發過程中，我們將構成此架構的兩大基礎元素—運作流程(Operation)及裝置(Device)以適當地抽象化方式來表達，分為運作抽象化(Operation Abstraction)與裝置抽象化(Device Abstraction)。前者將架構內的運作流程抽象化以循序(Sequential)及狀態機(State Machine)這兩種流程方式來表達；後者則是將架構內的裝置抽象化，用物件模組(Object Model)技術表達在 UPnP 網路上的裝置以及軟體開發架構內各種功能的元件模組。
- 軟體開發方法除了上述的抽象分類外，在 Workflow 技術中我們也依功能性的不同將其內部的模組階層化封裝(Hierarchical Encapsulation)，可分為低層直接控制實體裝置的模組，高層具有邏輯及自動化功能的模組，及系統與系統之間的互動模組。清楚且完整的階層化封裝，不但在開發過程中可以任意地整合且置換運作模組或演算法模組，而且封裝後的邏輯設計還可透過圖形化介面重複運用，與其他的邏輯設計整合。這樣階層式模組化的設計對於整體智能家庭系統的開發、整合及維護都顯

得更有效率。

- ❖ 在架構中的運作流程及裝置之間的互動，裝置會以事件(Event)的方式通知讓運作流程接收，進行下一步的運作步驟；運作流程則會以透過控制動作(Control Action)的方式讓裝置進行動作，事件及動作的機制是此軟體開發架構內運作流程與裝置之間最重要的溝通方法。

此軟體開發架構最後將實現在具有工作流程(Workflow)與通用隨插即用(UPnP)技術的平台上，以工作流程技術實現運作流程的抽象化，在工作流程技術的圖形化介面平台上進行整合性運作流程的設計與開發，並利用封裝功能，快速的拖曳設計且重複運用；通用隨插即用技術則會實現裝置的抽象化部份，能讓架構內的裝置快速地與環境中實體裝置進行配置。

1.3 論文大綱

本論文接下來的部份，將在第二章介紹一些背景知識以及相關論文的研究。第三章則是本研究的重點部份，歸納出此軟體開發架構在開發設計一個即時反應系統時的運作流程抽象化的選擇法則，以及裝置抽象化的將架構分為五個功能性元件模組並分別描述其設計時的思考理念，此章節也會詳細介紹在運作流程與裝置之間的傳遞機制，及階層式封裝過程中的選擇及分類。第四章介紹 UPnP 裝置與 Workflow 模組彼此之間的互動關係，以及如何透過 UPnP 的發現機制動態的在工作流程技術中配置運作機制的工作流程。第五章以室內光控系統以及簡單的智慧型置物櫃作為實例來證明此軟體開發架構的可行性。第六章總結本研究的成果，並提出未來仍可研究的課題。

第二章 技術背景

2.1 通用隨插即用 (Universal Plug and Play™ , UPnP)

通用隨插即用 (Universal Plug and Play, UPnP) [2] 是一種遍佈式的點對點網路架構，任何裝置如智能家電、行動無線裝置或個人電腦等等，都可以藉由UPnP這個網路架構彼此互相連結通訊。以TCP/IP與Web技術組成的開放網路，讓連入UPnP網路中的裝置可以被控制並傳送資料，其設計理念提供人們方便使用且有彈性，可以自動偵測網路中各種具有UPnP的裝置且相戶連線，不需要作任何的配置設定即可立即使用裝置所提供的服務。

UPnP論壇[3] 是工業界提倡UPnP技術的一個組織，目的在於整合眾多不同廠商的產品定義，制定簡易且健全的標準協定，且為了讓裝置與裝置能有意義的溝通，發展標準的UPnP相關協定與定義標準裝置XML描述，同時也審核欲發展成標準裝置定義的協定，並發放符合UPnP標準的商標。

2.1.1 **UPnP 網路基本組件**

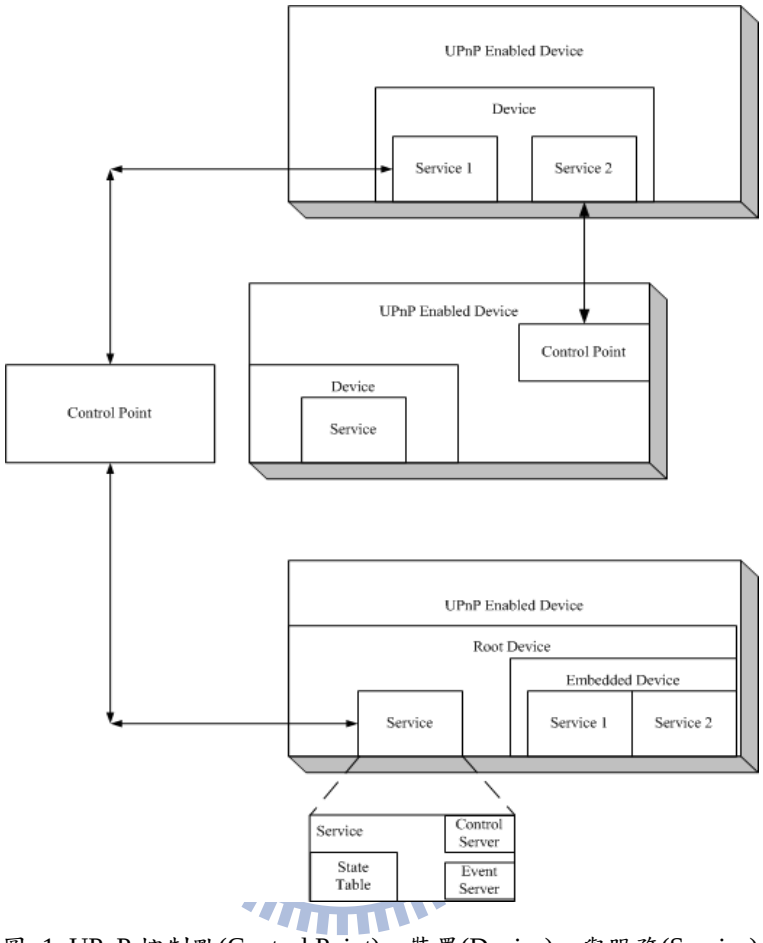

圖 1: UPnP 控制點(Control Point)，裝置(Device)，與服務(Service)

本段的部份圖文是取自Understanding Universal Plug and Play[4]。UPnP由三個基本組件所組成，分別是裝置(Device)、服務(Service)與控制點(Control Point)，圖 1是這三個組件的彼此關係，可以看出彼此之間會是巢狀關係。

❖ 服務(Service)

服務是 UPnP 中最小的控制單位，它提供了功能(Action)以及一組狀態變數(State Variable)來紀錄目前此服務的情況，功能(Action)的設計可以讓服務完成一個特定任務並傳遞參數。狀態變數(State Variable)具有事件(Evented)功能，可以指定當變數數值改變時會觸發事件功能。

❖ 裝置(Device)

裝置(Device)是包含了服務的設備，也可以包含嵌入式裝置(Embedded Device)。

❖ 控制點(Control Point)

控制點可以偵測及控制 UPnP 上的裝置，當取得 UPnP 裝置後，控制點主要可以做的事情有：

- 取得屬於 XML 的裝置描述文件(Device Description)，從而取得裝置中相關服務的簡易列表。
- 對控制點有興趣的服務去取得其服務描述文件(Service Description)。
- 傳送功能(Action)來控制服務。
- 對有興趣的服務進行訂閱動作，當訂閱服務的變數狀態(State Variable)有所改變時，會送回一個事件訊息(Event)。

在「基於工作流程與通用隨插即用技術下之整合性軟體開發架構」中會將裝置控制點封裝成模組，且在流程中進行訂閱事件及傳送控制服務。

2.1.2 **Intel UPnP 軟體開發套件**

在本研究中所實作的UPnP裝置與控制點都是使用Intel UPnP SDK[5] 來完成的，其使用的軟體及功能如下：

❖ 首先使用 Intel Device Author 來設計服務內容，包括內含的功能(Action)及狀態變數(State Variable)，得到為 XML 的服務描述文件(Service Description)。

❖ 接著使用 Intel Device Builder 加入服務描述文件來設計 UPnP 裝置，以及裝置中的控制點。

❖ Device Spy是微軟的標準控制點，可以偵測所有在UPnP網路上的UPnP裝置，並將裝置所有的詳細資訊陳列在Device Spy上，包含嵌入式裝置(Embedded Device)與服務(Service)及每一個服務上的功能(Action)與狀態變數(State Variable)，供使用者直接操作這些功能。圖 2是Device Spy中各個元件的圖示以及說明。

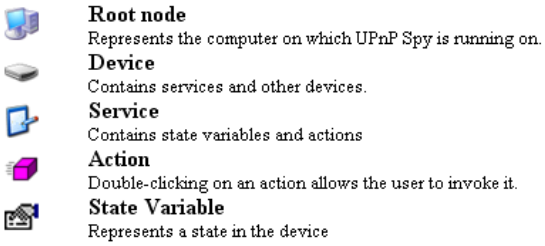

圖 2: Intel Device Spy

2.2 工作流程基礎架構(Windows Workflow Foundation, WF)

Windows Workflow Foundation[6] 是微軟在 2007 年初推出可在.NET Framework上執行工作流程的軟體開發架構。WF[2]架構不但整合了Visual Studio的圖形化設計開發介面，也包含Framework、.NET API、以及一些可延伸的工具服務等等的，WF並非是個能獨立執行的產品，而是一個單純的開發技術、一個基礎，可以藉由其API和工具讓開發者將工作流程的功能嵌入到一個主控程式內，使得在主控程式中的流程時能以圖形化設計輕易的開發。程式開發人員藉由WF來完成一套工作流程系統，並將功能性的流程活動元件客製化，這樣的技術還可以整合其他完成的Web Service或外部程式，在不同的平台上解決多種人與系統之間複雜的工作流程問題。

2.2.1 WF的基本組件

WF的基本組件有工作流程(Workflow)、活動(Activity)、服務(Service)，以及執行引擎(WF Runtime Engine)[圖 3]。工作流程與活動為組成一個工作流程個體(Workflow Instance)的元件，也就是開發圖形化流程設計所使用的主要元件。執行引擎主要就是用來執行工作流程個體的，而服務則是附加在執行引擎上的功能。接下來我們即介紹這些基本組件。

[2]在此論文中 WF 將泛指工作流程技術，而 Workflow 則是指 WF 中的工作流程元件。

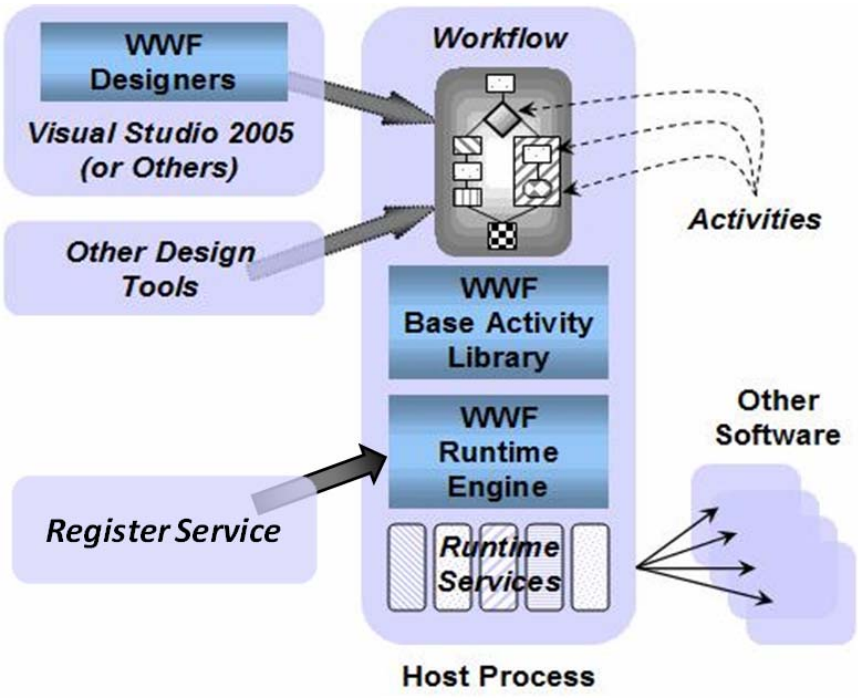

圖 3: WF基本元件[3]

❖ 活動(Activity)

活動(Activity)是 WF 中最基本的單元，所有工作流程是以具備各種功能的活動來組成的。在 WF 中，已經內建了許多標準活動來讓開發者自行使用，這些標準活動根據複雜性以及功能性來分類約可以分成：基本類、通訊事件類、錯誤處理類、異動和補償類、條件和規則類、網路服務類及狀態機工作流程類。我們在此會介紹四種較重要的活動：

1. 程式碼活動(CodeActivity)—如同名稱所述，當開發者希望工作流程在運行時，能執行某些程式碼，則可以使用程式碼活動，將程式碼置於活動的事件中。此活動是程式開發人員最常使用到的，可以藉由程式碼活動，將程式撰寫封裝於活動內，達到重複利用的價值。
2. 狀態傾聽活動(ListenActivity)—當運用到此活動的時候，可以在工作流程運行中，等待 Workflow 的內部事件發生，可以同時等待多個事件發生，但只有一個事件將會被執行，通常也會運用一個類似 Timer 的活動機制—延遲活動(DelayActivity)來將狀態傾聽活動繼續執行下去。
3. 呼叫流程活動(InvokeWorkflowActivity)—這是一個以非同步的方式，呼叫外部的工作流程，並且開啟一個新的執行緒各自運行。由於是非同步的呼叫，所以多個工作流

[3] WWF 為微軟官方之前稱 Windows Workflow Foundation 的縮寫，現在都稱為 WF

程彼此的並沒有一個固定的執行順序，而呼叫工作流程之後，可以用 Binding 的方式傳遞設定初始參數值，但僅能由原工作流程傳出參數值給新的工作流程，並不能從新的工作流程傳參數值回來。

4. 處理外部事件活動(HandleExternalEventActivity)—此活動會等待外部程式引發的特定事件，當事件引發後，在活動內部可以得到事件所帶來的參數。而事件的定義需要透過介面(Class interface)來完成，並根據介面來完成外部程式與活動之間的處理關係。

標準活動除了上述這四種以外，還有提供許多支援開發流程活動如：條件判斷活動(IfElseActivity)、迴圈活動(WhileActivity)等等的。

除了標準活動外，WF 也提供了客製化活動(Custom Activity)，這並不是一個已經設計好的活動，而是提供一個方法來讓開發者可以根據需求來自行設計一個或多個屬於自己且可重複利用的活動，可由多個標準活動或其他客製化活動共同組合完成，促使 WF 的設計功能更強大、更具有彈性。

❖ 工作流程(Workflow)

工作流程(Workflow)為開發一個設計流程的最主要元件，它就像是一個模板框架，可以讓開發者在設計時期加入上述的活動(Activity)來完成整個流程。在啟動一個 WF 技術時，必定會先執行一個主要的工作流程，當然在過程中也可以加入或呼叫其他的 Workflow，彼此各自或互動的進行流程。對於工作流程的分類，我們根據流程性分為循序工作流程(Sequential Workflow)以及狀態機工作流程(State Machine Workflow)，而在之後章節我們將會詳細介紹其區別以及在本研究的應用上所做的選擇。

❖ 工作流程執行引擎(WF Runtime Engine)

工作流程執行引擎(WF Runtime Engine)是一個執行工作流程的函式庫，提供必要的基礎執行組件。非同步的起始創建、啟動工作流程都是在其引擎中，管理工作流程個體的狀態，或可以加入事件處理常式來管理執行狀態改變的事件等。執行引擎還可以載入不同的服務(Services)，彈性的增加功能。

❖ 流程服務(Workflow Service)

由於工作流程引擎(WF Runtime Engine)只提供執行基本的工作流程，可附加在執行引擎的流程服務(Workflow Service)於是就出現了。當在主控應用程式開始一個執行引擎之前，可以根據工作流程內部所需，來加入各種不同的流程服務，使得在規劃設計工作流程時，可以使用這些服務來達到更高的目的，進行更複雜的事情，就像是加入程式設計中加入函式庫一樣，執行引擎提供了一個擴充的架構，透過內部的函式加入新的服務到 WF 環境內。除了微軟所提供的流程服務外，也可能取用其他開發者所提供，或自行開發新的流程服務。流程服務在 WF 內分為兩類：

1. 核心服務(Core Service)—核心服務其實就是微軟提供的 WF 內建服務，共有四種，所提供的功能有像是可以保存當前工作流程的執行狀態到永久性的儲存環境中(如資料庫)、管理排程工作流程執行時所需的執行緒、監控執行時期想要的流程資料，及確保工作流程執行時相關成員得到相同的狀態。
2. 合約服務(Local Service)—WF具有一個可以使工作流程個體與外部程式溝通的服務。透過合約服務，外部程式可以呼叫合約服務內的函式觸發事件產生，讓工作流程在等待事件的地方可以正確接收，達到溝通的效果，除此之外，合約服務也能反向的讓工作流程呼叫函式，讓外部程式得到回應加以互動[圖 4]。

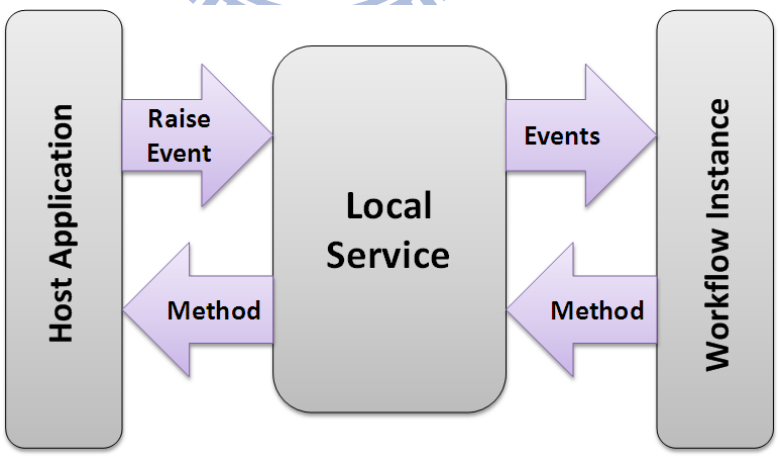

圖 4: 主程式與流程個體透過合約服務的互動[4]

[4]此段部份圖文及中文名詞翻譯參考：MSDN—Introducing Microsoft Windows Workflow Foundation：An Early look 及書目—《Pro WF - Windows Workflow in .NET 3.0》、《Windows Workflow Foundation 新一代工作流程開發實務》。

2.3 相關研究

2.3.1 UPnP階層式架構及室內燈控系統

國立交通大學碩士—劉育志的研究「適用於家庭自動化的通用隨插即用感測與促動器基礎架構」[7] 裡曾經提及以通用隨插即用(UPnP)的技術建立一套家庭自動化網路裝置，這篇論文主要貢獻在於將UPnP的架構階層化，其功能模塊如同圖 5。圖中將高、中、低三層裝置，低層介面直接與廠商自訂裝置介面作互動，中層為通用裝置介面與低層聯繫，高層為即時反應服務介面控制中層的裝置。數位家庭伺服器(eHome Server)這個高層系統的控制中心，與階層化架構這兩部份將會與我們的研究有密切的相關性，接下來的部份將會詳細說明。

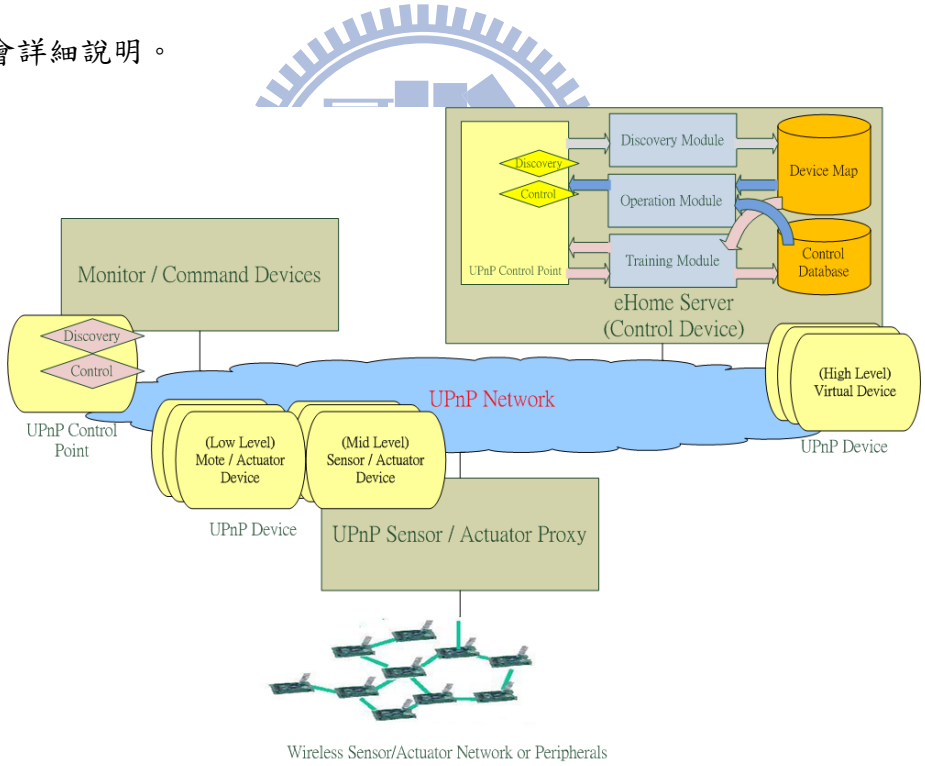

圖 5: 感測與促動器基礎架構功能模塊

❖ 低層：廠商自訂裝置與服務介面

實作 UPnP 低層裝置主要是做為一個與實體裝置溝通的介面，描述的實體裝置可為無線感測網路的微型監測器(Node)、感測裝置(Sensor)或促動器裝置(Actuator)等。由於是與實體裝置接觸，所以 UPnP 裝置介面必須描述製造實際硬體的廠商所提供的所有功能及

服務。除此之外，低層裝置也提供與環境相關的資訊，例如放置地點、裝置狀態、及管理功能等。而每一個微型監測器(Node)、感測裝置與促動器裝置都會具有一個對應的UPnP 裝置。

❖ 中層：通用裝置與服務介面

中層裝置是一個標準化介面，目的是提供各種控制系統一套標準的功能介面，如燈光控制系統具有一套標準的燈光控制功能，具備調整光亮值功能、燈光開關控制；空調控制系統標準功能，具備偵測調控溫度，調整風速，開關控制等等。由於是一個標準的裝置介面，UPnP 論壇對於某些控制裝置以提供了標準描述文件(Standardized Description)來使用，如促動器裝置；而 UPnP 論壇無提供標準裝置的部份，則可以自行開發標準定義。中層的介面所提供的服務，都會與低層的 UPnP 裝置作對應，藉由低層裝置服務真正的來控制實體裝置。

❖ 高層：虛擬裝置與服務介面

高層裝置為一個虛擬裝置，可以作為與實際使用者溝通的介面，此層裝置是發展即時反應系統所需要具備。當開發者知道有哪些中層裝置時，則可以挑選欲控制的裝置系統加以開發高層系統。開發高層裝置也提供了模組化的概念，使系統易於開發、維護及增加可利用性。

❖ 數位家庭伺服器(eHome Server)

數位家庭伺服器(eHome Server)是利用高層裝置介面做自動化系統的流程與控制，而藉由標準化的中層裝置來實際調整裝置參數。此伺服器中為了控制中層的裝置服務所以具有 UPnP 控制點(Control Point)，以及啟動高層虛擬裝置自動化的控制系統。自動化控制流程也在伺服器中的操作模組(Operation Module)中來實作。

❖ 室內燈控系統(Indoor Luminance System)

劉育志的研究中，以他自己的架構完成了裝置在UPnP架構上的室內燈控系統，此室內燈控系統可以自動化的調控室內中各不同區域的亮度，藉由遍佈在室內各個區域的感測器所感測到的外界光亮值，來與區域的燈光控制器作一個協調的自動控制。此外界光亮

值包括了燈光所給予的光亮度及外界環境(陽光)所給予的光亮度，透過演算法計算後得到對應的自動化控制 [圖 6]。

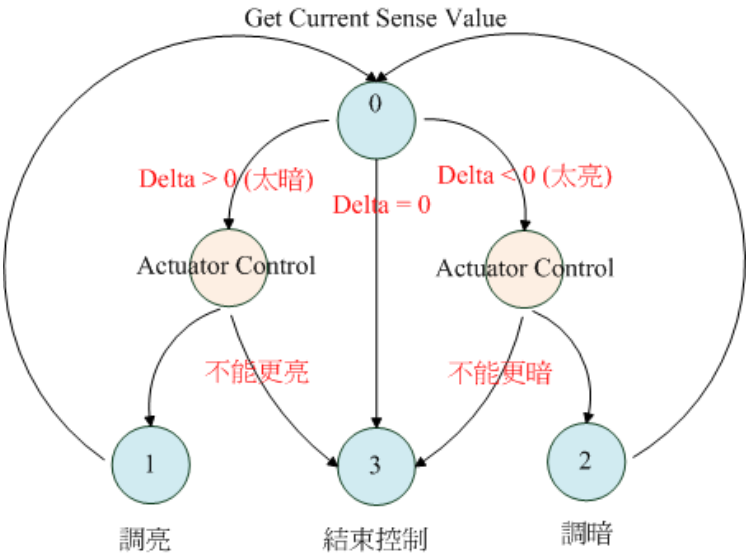

圖 6: 室內燈光控制方法

階層化的 UPnP 裝置讓開發及維護一個智能系統變得更容易且便利。對於實體裝置而言，不同的廠商所提供的功能以及與實體裝置的溝通協定會有所不同，當智能系統中的實體裝置有所改變，也就是可能換了一個不同廠商的實體裝置時，此階層化使得智能系統只需要調整低層與實體裝置溝通的介面，中高層完全不需要重新修改；而當開發者重新定義高層自動化控制流程的時候，也只需抽離高層裝置重新改過即可。此階層化架構讓智能系統具有更大的彈性，也因此讓我們的研究能更好的在其架構上進行擴充。

在自動化控制的開發中，劉育志的研究以大量程式碼開發出控制流程，對於要去維護此自動化控制流程的其他開發者來說，並不容易被瞭解，甚至對於僅僅只是想利用舊有自動化控制流程來開發新流程的智能家電開發商來說，還需要去瞭解舊有程式碼內容，重新編寫其程式碼。而在我們的研究中，將會把數位家庭伺服器(eHome Server)中的自動化流程部份以 Workflow 的技術來取代大量程式碼編寫的流程，圖形化介面不但讓開發者更快瞭解整體的流程，且將操作中、高層 UPnP 裝置的部份用 Workflow 封裝，讓未來的開發者透過 Workflow 技術的好處快速維護及使用自動化流程。

2.3.2 智慧型置物櫃(Smart Pantry)

智慧型置物櫃出自於「Smart Pantries for Homes」[8]這篇論文，目的在於增進居家環境生活的便利性，讓智能環境走入一般家庭，提昇生活品質。此論文中的智慧型置物櫃分為兩種，其中一種稱為BAC Pantry，是因為此置物櫃將會以物品的bar code作為判別物品的依據。BAC Pantry的每個櫃子都佈有感測器，是為了偵測櫃子是否有物品放置在其中，當放入新物品到置物櫃時，會啟動bar code掃描器，掃描物品的bar code，並提示哪些櫃子是空的。接著將物品放入空置物櫃中，感測器則會收到物品放入的訊息，儲存物品資訊(包括bar code、物品名稱、放入日期等)[圖 7]；而當物品從置物櫃中被取走後，感測器即會反應告知使用者物品已用完，是否需要重新填補等等的訊息[圖 8]；當然因為由於牽扯到人類行為，所以還會有很多例外狀況的處理，在這篇論文皆有討論到。

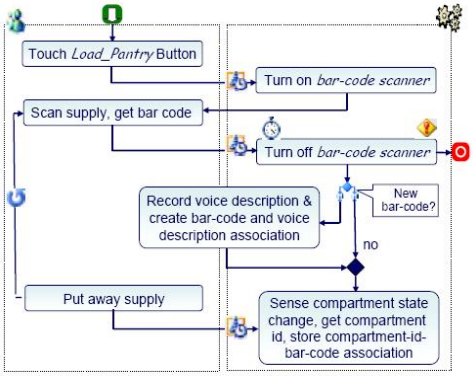

圖 7: Load Pantry 流程

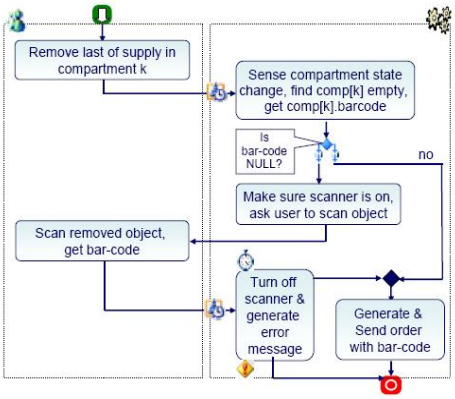

圖 8: Remove Pantry 流程

第三章 設計理念

圖 9為本軟體開發架構的各元件模組，紅色箭頭表示一個實際的事件(Event)發生且在模組之間傳遞，藍色箭頭表示觸發事件的函式，紫色的箭頭則是代表模組間的呼叫函式。而環境與使用者是影響此架構運行的兩個外在條件。

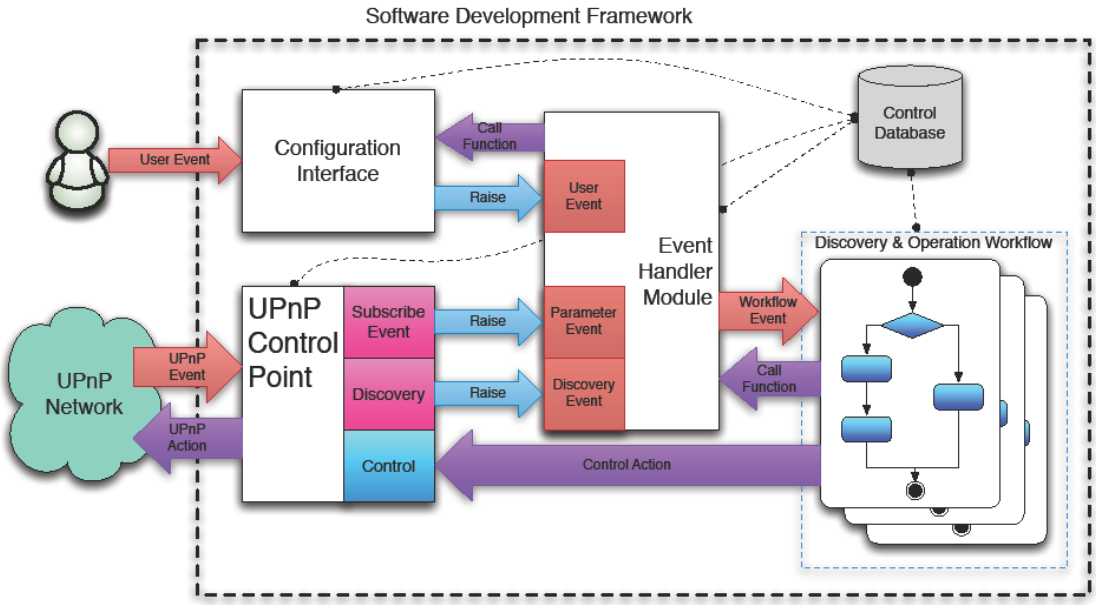

圖 9: 軟體開發架構之元件模組與關係

通用隨插即用網路(UPnP Network)與環境是密不可分的，透過無線感測網路(Wireless Sensor/Actuator Network)或週邊感測器與促動器裝置(Peripheral)來接收環境內的數值，並直接將數值反應在 UPnP 裝置上並加入到通用隨插即用網路上(UPnP Network)，讓此架構可以透過 UPnP 控制點(Control Point)從網路上得到環境的數值，也可以藉由網路來控制環境中的家電裝置。而使用者部份則可以透過調控介面(Configuration Interface)直接操控架構內運作流程的進行。

執行工作流程需要透過工作流程執行引擎(Workflow Runtime Engine)，所以無論是UPnP裝置事件或使用者觸發的事件，外界環境要與內部的工作流程溝通，都需要有一個事件處理模組(Event Handler Module)作為溝通的橋樑。圖 9中右下角的Workflow為工作流程個體(Workflow Instance)，即時反應系統的運作流程開發即是在此。

控制資料庫(Control Database)是這個架構內不可或缺的一環，對於開發 Workflow 以及

UPnP 裝置加入時的各種選擇，都將會透過控制資料庫來決定。上述各元件模組的功能以及模組封裝的分類與優點都將在稍後的章節內有詳細的介紹。

第三章首先我們會介紹運作流程(Operations)的抽象化表示它的分類選擇法則及實現執行法則，接著介紹系統內裝置(Devices)的各分類元件模組，以及運作流程與裝置這兩者之間互動的方式。最後在 3.4 節則介紹這些運作流程及裝置元件如何以階層化的方式進行封裝。

3.1 運作流程建模之法則

WF 是本架構的重點技術之一，在開發一個系統的運作流程過程中會使用到大量的 WF 元件，要如何使用及選擇這些元件來達到最好最有效率的開發，我們將在此節中作一個探討。

3.1.1 循序工作流程(Sequential) vs. 狀態機工作流程(State Machine)

Workflow 為開發一個系統運作流程最主要的元件，完整的描述了服務中的執行流程。而抽象化的運作流程將在 WF 中分為兩種不同形式的作法：

- 循序工作流程(Sequential Workflow)—這是一個以 Flow Chart 方式實作的工作流程。當完成一個循序工作流程後，執行時會根據在設計時期即規範好的流程步驟進行，任何條件式、迴圈式的分支，在執行之前都清清楚楚的被定義下來。由於有明確的起始與結束，循序工作流程對於開發者開發自動化流程來說更加的清楚且快速。
- 狀態機工作流程(State Machine Workflow)—即為一個有限狀態機的工作流程，在流程內的運行會隨著不同事件的進入而有著不同的流程順序，且所有狀態之間的轉換都是由外部事件(Event)或是內部計時器(Timer)所觸發控制的。由於流程為一個一個的狀態所連接而成，所以任何狀態都可以成為一個起始或結束的狀態。

決策依據：任何工作流程都可以以循序工作流程與狀態機工作流程來完成，只是複雜性與適合性的區別。對於如何決策使用這兩種不同的工作流程，我們以實作流程時最自然

的模式來決定。對於一個即時反應系統的整體運作流程而言，我們不難發現系統內需要等待接收許多外界發出的事件後立即作反應，這些事件可能是使用者的動作、可能從環境中的裝置發出，我們很難預測在哪些時間點上會有哪些事件需要去接收，所以使用狀態機工作流程是最自然的模式以表現系統完整的流程；另一方面，每當系統接收到事件後，會馬上進行反應的動作，反應動作可以是在控制環境裝置，也可以是回應使用者訊息，反應的動作最自然的模式會以連續性的動作來完成，會選擇循序工作流程。

3.1.2 封裝法則(Encapsulation) vs. 呼叫法則(Invocation)

此節為實現系統內流程的方法，各工作流程根據不同的執行方式，歸納出兩種的法則作為使用的依據，並具有重複利用性的封裝效果。

- 封裝法則(Encapsulation)—此法則在程式設計上的表現，相當於是一個函式呼叫(Function Call)，它會在同一執行緒(Thread)下完成所有流程，對於具有循序性執行的工作流程來說，每個元件的執行都需要等待前一個元件執行結束後才能進行，完全按照流程的順序。在 WF 中，內建的客製化活動(Custom Activity)讓我們可以直接開發出具有此封裝法則的元件，並且將元件進行模組化來重複利用，在工作流程內重複利用這些開發出的客製化活動，會按照流程順序來同步的執行。
- 呼叫法則(Invocation)—為了解決在同時間內處理多個不同的 Workflow，我們必須呼叫多個不同的執行緒(Thread)個別進行。使用呼叫法則(Invocation)即是在控制執行流程時，以 Fork Thread 的方式來讓多個 Workflow 具有各自的執行緒執行，也因此我們無法預期各 Workflow 彼此之間的執行順序。在 WF 中有一個標準活動—呼叫流程活動(InvokeWorkflowActivity)是可以來完成此呼叫法則，當開發者將工作流程設計完成後，使用此活動來呼叫工作流程則會非同步執行當前的 Workflow 與被呼叫的 Workflow。

決策依據：同步(Synchronism)與非同步(Asynchronism)是設計封裝工作流程時最重要的選擇依據。對於同步(Synchronism)而言，每當一個步驟被要求執行的時候，下面的步驟都需等待執行的完成，這段期間整個 Workflow 都無法進行其他動作；非同步(Asynchronism)則是一次可以進行多項事件。由於呼叫法則(Invocation)會牽扯到新執行

緒的產生，對於機器的負擔相較於封裝法則(Encapsulation)是較重，所以選擇適合的法則運用對於開發設計時是有其重要性的。在設計一個即時性反應系統時，若是希望系統內的多個流程皆能同時保持執行狀態，一直接收事件進行處理而不因為其他工作流程(Workflow)或活動(Activity)就被限制住，呼叫法則(Invocation)是最適合的運用方式。相反地，若流程是屬於循序性的執行，封裝法則(Encapsulation)的運用相對來說是負擔較輕的使用方法。

3.2 元件模組及設計依據

在此軟體開發架構內，裝置抽象化是以物件模組(Object Model)的方式完成，為了讓 UPnP 裝置能有效的整合在 WF 中，物件模組在架構內區分為幾個不同功能的元件模組，分別著重在控制接收 UPnP 裝置元件、設計 Workflow 自動化流程元件、接收使用者要求之元件、溝通 UPnP, Workflow, User 三方的元件，以及提供運作選擇的控制資料庫。這些元件模組依功能性共分為五種，將在之後的部份明白定義每個元件模組的結構與功能，以及其設計時的依據。

3.2.1 UPnP控制點模組(UPnP Control Point Module)

UPnP 技術中，隨插即用的優點讓本架構輕易地與環境中的裝置作溝通。此模組即是本架構用來偵測及操控 UPnP 網路上裝置的動作而產生的，我們依功能性將模組共分為三個機制：

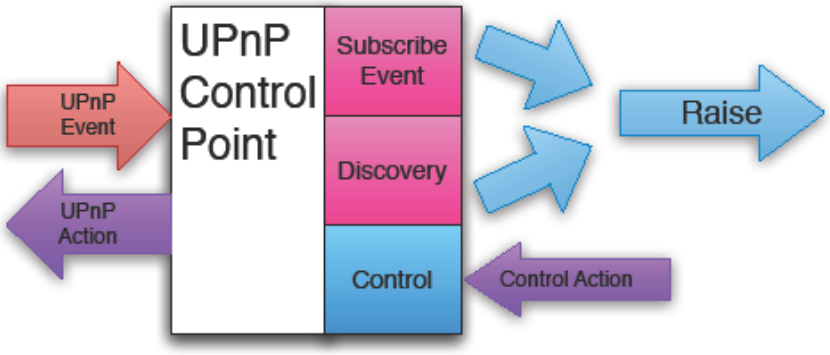

圖 10: UPnP 控制點模組區塊

❖ 發現機制 (Discovery Mechanism)

在控制UPnP裝置前，能找尋到欲控制的UPnP裝置是一件很重要的事情。UPnP控制點的發現機制(Discovery Mechanism)即是用來偵測到UPnP網路內裝置的加入以及移除。透過標準通訊協定—簡單服務發現協定(Simple Service Discovery Protocol：SSDP)，發現機制可以在UPnP網路上發現有哪些裝置(UPnP Device)，哪些服務(UPnP Service)，以及取得裝置上所提供的服務資訊。此機制在指定的裝置加入或從網路中移除時能取得這些指定裝置的相關資訊，也因此當加入或移除的事件[圖 10的UPnP Event]產生時，我們會將擷取到的裝置資訊加入到控制資料庫(Control Database)中，提供其他模組或機制的使用，同時也會觸發一個Workflow的事件[圖 10的Raise]，讓Workflow元件得知裝置加入或移除的訊息。

❖ 控制機制 (Control Mechanism)

UPnP控制點的另外一項重要任務就是利用控制機制(Control Mechanism)去控制裝置的動作。透過前述的發現機制中取得的裝置資訊後，控制機制可以向裝置發出取得裝置上的服務描述(Service Description)的請求，並透過服務描述向網路中的UPnP裝置發出動作訊息 [圖 10的UPnP Actions]，這樣的遠端程序呼叫在UPnP技術中是透過簡易物件存取協定(Simple Object Access Protocol：SOAP)來完成。而呼叫動作的內容，可以是輸入參數要求裝置調整數值，也可以是要求回傳裝置上的內容，而主要控制的呼叫動作會在運作工作流程(Operation Workflow) 的模組那邊執行[圖 10的Control Actions]。

❖ 訂閱事件機制 (Subscribe Event Mechanism)

除了偵測在 UPnP 網路中裝置加入及移除之外，控制點還可以訂閱監視 UPnP 裝置中的數值，每當這些數值有所改變時，控制點則會收到事件的發生以及得到改變的數值，同時觸發 Workflow 事件。對於開發一個與環境相關的自動化系統而言，通常最需要去偵測的就是環境中不停在改變的數值因子(如：光度、濕度、溫度等等)，雖然說大部分的時間這些因子的改變都是緩慢地。因為需要隨時監視這些因子的變動並做相對應的調整，環境中各種不同的即時反應系統必須進行非同步的執行。此機制的動作將觸發下一節的

事件處理模組中的事件。

設計依據:UPnP 控制點是本架構與環境中 UPnP 裝置溝通的唯一管道,把欲監控的 UPnP 裝置加入到 UPnP 控制點後，無論是控制點加入到 UPnP 網路上時可以立即取得控制 UPnP 裝置的權力，或是 UPnP 裝置進入網路中控制點的即時偵測反應，都將使得架構能輕易地與智能環境建立溝通的橋樑，是本架構絕對不可或缺的重要模組。由於 UPnP 控制點是架構內唯一能從 UPnP 網路上取得 UPnP 裝置的地方，擷取到的資訊勢必要放入資料庫以供其他模組來使用，資料庫的內容將在模組—控制資料庫(Control Database)的地方再來更仔細的探討。

3.2.2 事件處理模組(Event Handler Module)

圖 11為事件處理模組的功能區塊圖。由於WF是此架構的中心技術，一個系統運作流程的處理即是使用WF中的工作流程執行引擎(WF Runtime Engine)來完成。工作流程個體(Workflow Instance)內可以傳遞與接收事件的發生,但因為流程個體是由工作流程執行引擎(WF Runtime Engine)所啟動及執行，只有屬於執行引擎內的事件，流程個體才能接收的到。此研究以執行引擎的內外部作為基準，將事件分為內部事件(Internal Event)以及外部事件(External Event),外部事件的觸發需透過此事件處理模組(Event Handler Module)轉換成對應的內部事件來與流程個體進行溝通，流程個體也能經由模組來發出訊息讓外部程式接收到。

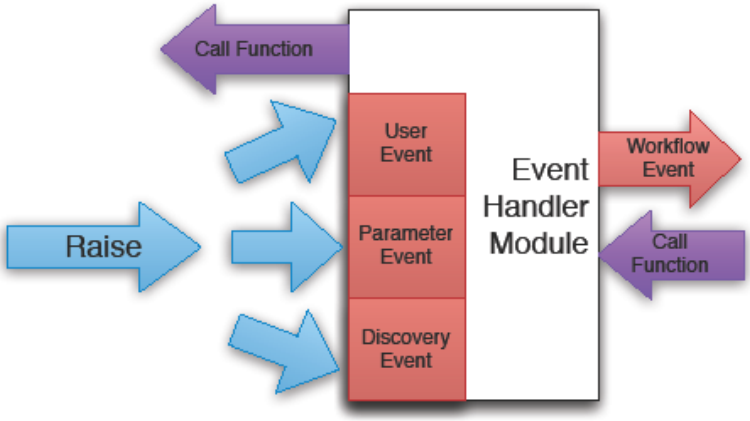

圖 11: 事件處理模組區塊

3.2.2.1 事件種類

事件處理模組是由三種事件架構而成：

- UPnP 事件(UPnP Events)

  屬於外部事件(External Event)，通常是環境中的裝置有所動作而觸發，經由 UPnP 控制點得到，並在控制點上進行事件處理。UPnP 事件又可分為三種類型：

  1. 發現事件(Discovery Events)是環境中的 UPnP 裝置加入到網路時所被立即反應出的事件，是 UPnP 內不可或缺的事件。
  2. 移除事件(Removal Events)是環境中的 UPnP 裝置從網路上被移除時反應出的事件，是 UPnP 內不可或缺的事件。
  3. 變數事件(Parameter Events)由 UPnP 裝置上的變數改變而觸發，這些變數可以為環境上的變數因子(如光度、溫度等)，也可以為人類所決定的變數(如光度設定、物品名稱等)，變數事件是對應這些變數改變而產生的。變數事件由控制點中的事件機制(Event Mechanism)來取得。

- 使用者事件(User Events)

  使用者事件(User Events)也是屬於外部事件的一種，由使用者的行為所觸發，通常都是經由調控介面模組(Configuration Interface Module)來接收。使用者事件會與系統中需要操控的變數做配合。

- 工作流程事件(Workflow Events)

  流程個體可接收唯一的內部事件。流程個體使用標準活動的處理外部事件活動(HandleExternalEventActivity)來接收宣告定義在介面(Interface)中的工作流程事件。除了流程個體外，也可以在其他地方接收及處理工作流程事件，用來更新及顯示自動化流程的資訊(如在調控介面模組上顯示)。此外，流程個體中的狀態機工作流程(State Machine Workflow)內所有的狀態流程轉換，也是透過工作流程事件的觸發來進行的。

3.2.2.2 處理模組功能

事件處理模組主要有以下的功能機制：

- ❖ 事件引數機制(Workflow Event Argument)

  流程個體ID (Workflow Instance ID)是傳遞工作流程事件(Workflow Event)時最重要的引數。由於流程個體可以同時有多個工作流程執行，具有這個引數才能將工作流程事件傳遞到正確的工作流程來接收。此研究的事件引數機制繼承WF原先的引數機制，會傳遞Guid型態的流程個體ID，並加入.NET中標準函式庫—Dictionary實作出傳遞多個引數的機制[表 1]。事件觸發時，此通用機制就能彈性地傳遞任何種類且任意數目的事件引數。

表 1: 程式碼—事件引數機制

```
[Serializable]
public class WorkflowEventArgs : ExternalDataEventArgs
{
        private Dictionary<String,object> _args;
        public WorkflowEventArgs(Guid instanceId, Dictionary<String,object> a): base(instanceId)
        {
            _args = a;
        }
        public Dictionary<String, object> args
        {
            get { return _args; }
            set { _args = value; }
        }
}
```

- ❖ 事件列舉機制(Enumeration Event Type)

  如前面所提到，為了讓 Workflow 能取得事件的觸發，事件處理模組將所有的外部事件(UPnP、User Event)都轉換成為內部事件(Workflow Event)，模組內使用了事件列舉機制，將所有的事件列舉成不同的事件類型且一一做對應及轉換，其他的模組可以透過列舉機制取得事件處理模組內的所有事件，正確地加以運用。

- ❖ 觸發事件機制(Raise Event Function)

  如圖 11中Raise所示，事件處理模組提供一個通用的機制讓各模組可以觸發任何的工作流程事件[圖 11 Workflow Event]，且配合列舉機制簡化了原先每個事件都要各

自的觸發事件機制這樣的詬病，利用此機制所取得的事件種類以及事件引數，做為欲觸發事件的選擇以及傳遞的引數。

設計依據：在第二章的WF技術中，我們曾經提到合約服務(Local Service)，是一條讓外部程式與Workflow進行溝通的管道，也可以讓Workflow透過函式呼叫來聯絡外部程式[圖 11 Call Function]。事件處理模組透過合約服務的方式與WF技術結合，讓開發者可以將所有欲和Workflow接觸的事件都放入此模組內，模組內通用的方式使得開發者能輕易地使用及擴展原先的架構，加入新的事件也變得更輕鬆更快速，是我們設計這個模組的重要依據。

3.2.3 工作流程模組(Workflow Module)

3.2.3.1 服務工作流程(Service Workflow)

服務工作流程(Service Workflow)利用 WF 技術中的工作流程(Workflow)元件來完成，在即時反應系統中會第一個被呼叫出來執行。此模組目的是為了接收 UPnP 裝置加入的發現事件(Discovery Event)，根據特定的變數讓每個加入的裝置動態配置到一個對應的運作工作流程(Operation Workflow)來進行內部運作的工作，動態配置的部份將在第四章詳細的說明。簡而言之，服務工作流程是一個即時反應系統中最主要的工作流程，有以下三個功能：

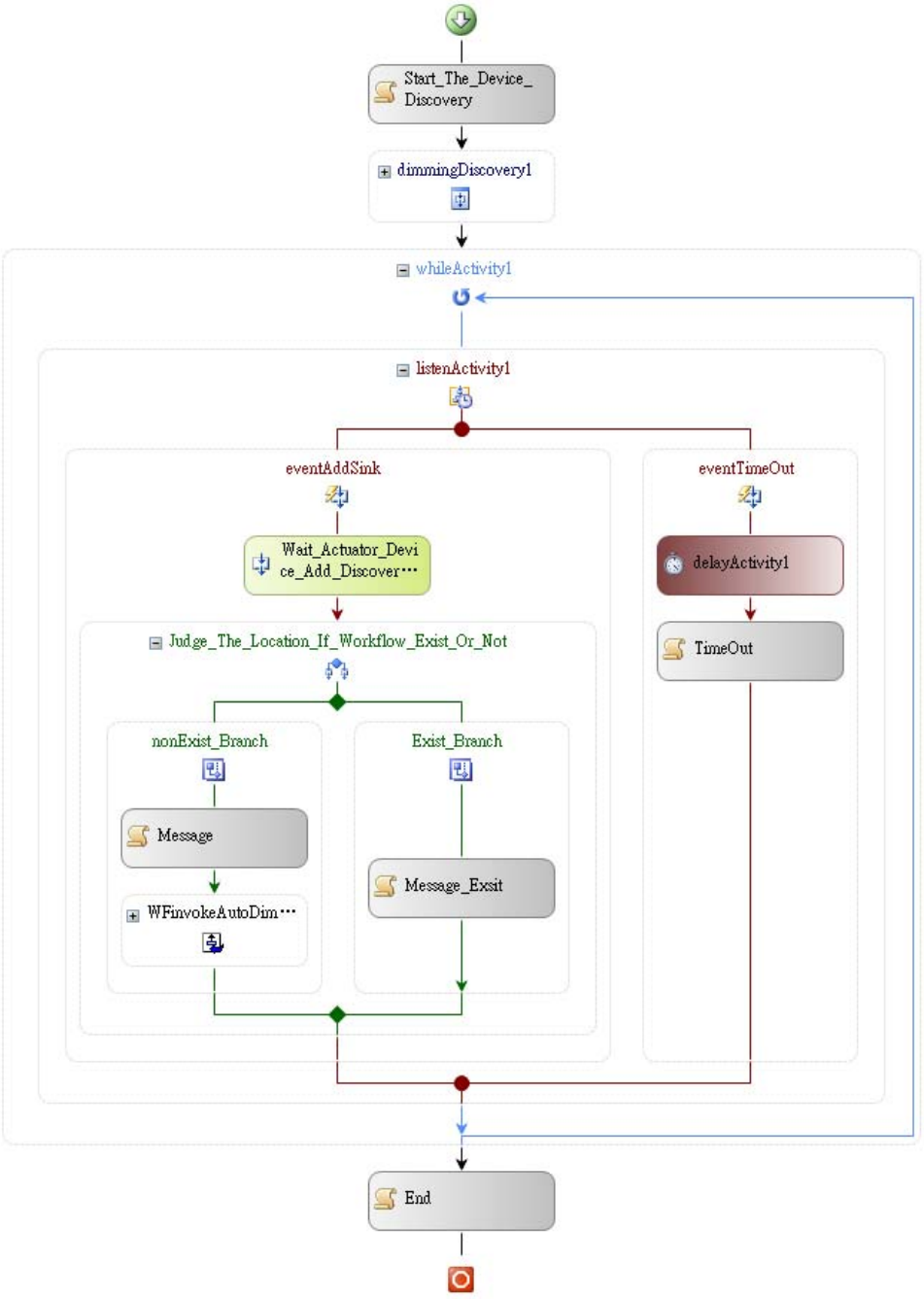

圖 12: 工作流程—服務工作流程

- 啟動UPnP控制點：由於服務工作流程是一個架構運作流程開始執行的第一支工作流程，一進入到此工作流程時，需要馬上啟動UPnP控制點(Control Point)開始偵測網路上的UPnP裝置，並接收UPnP事件[圖 12, 活動：DimmingDiscovery]。
- 接收發現事件：如圖 12, 活動：Wait_Actuator...所示，服務工作流程(Service Workflow)使用標準活動的處理外部事件活動(HandleExternalEventActivity)來等待接收內部事件的到來，內部事件是經由UPnP裝置加入到網路中所觸發的發現事件

(Discovery Event)所轉換過來。

- ❖ 呼叫運作工作流程：此工作流程中還有一個重要的功能，就是呼叫運作工作流程[圖 12, 活動: WFInvokeAutoDim...]。利用各UPnP裝置中的獨特ID或Location作為依據，呼叫並配置新的運作工作流程給新加入的裝置，讓每個裝置都有各自的流程，使得運作不會互相衝突。而服務工作流程執行完呼叫運作流程的功能後，則是會繼續的等待新裝置加入。

設計依據：3.1.1的循序工作流程(Sequential Workflow)及3.1.2的呼叫法則(Invocation)是我們設計此服務工作流程(Service Workflow)實作技術的選擇。前者在圖 12中我們可以很明顯的看到此流程只等待接收Discovery Event，當接收到事件後也會很快的跳回等待狀態，且流程大都已經規範好，整個流程以循序工作流程來完成是最為合適。後者由於智能環境中會有多數的系統及裝置進入，系統運作的即時性成為最基本且重要的要求，所以對於系統的啟動流程—服務工作流程，必須使用呼叫法則來執行。

3.2.3.2 運作工作流程(Operation Workflow)

各種控制UPnP裝置的活動元件和工作流程完成內部運作的流程[圖 13]。在本架構中，每個UPnP智能裝置都會擁有一個運作工作流程來完成其即時反應的運作流程，這些運作流程會根據不同需求而被開發者各自設計出，雖然流程不盡相同，但仍有幾個通用的事件在此流程中會被接收，我們歸納出以下幾種事件：

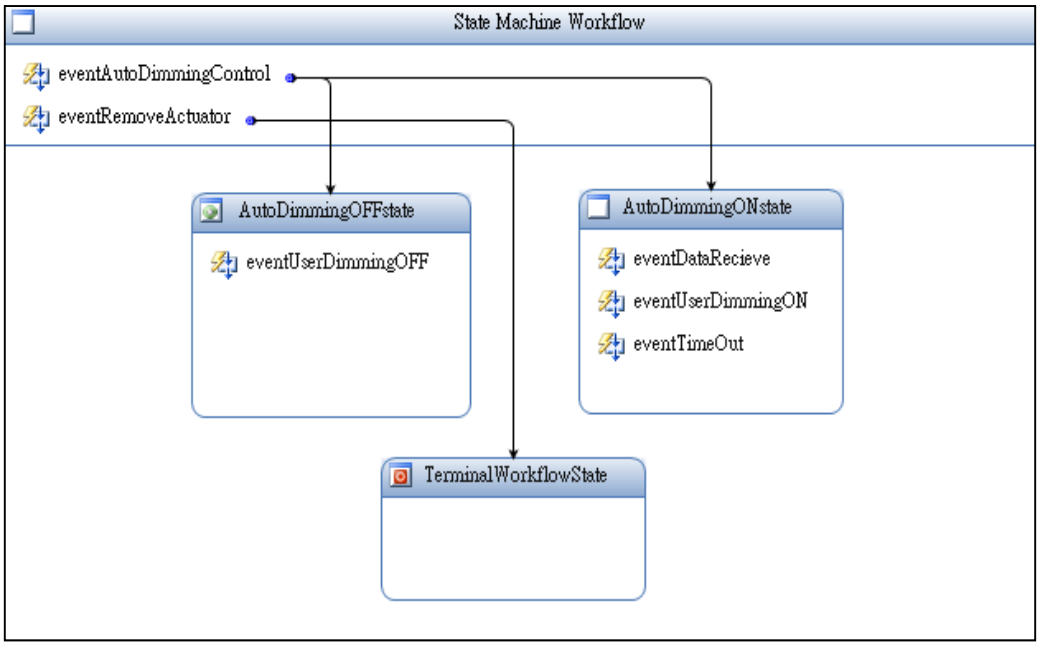

圖 13: 工作流程—運作工作流程

- 自動化事件(Automation Event)：讓系統運作流程在自動與手動狀態之間切換，通常是使用者透過介面觸發產生，當觸發為自動狀態時，即時反應服務即被啟動；而處於手動狀態時，則系統只會具有基本的手動功能。
- 移除事件(Removal Event)：此事件是當 UPnP 裝置離開網路後所觸發的事件。由於運作工作流程一對一產生來對應新加入網路中的 UPnP 裝置，反觀當裝置離開網路後，此工作流程就會結束運作。
- 變數事件(Parameter Event)：此事件其實就是內部運作所接收的環境參數改變時，運作工作流程得到的事件。對於要與環境進行即時性反應互動的運作工作流程而言，此變數事件是必須被接收的。

設計依據：3.1.1狀態機工作流程(State Machine Workflow)和3.1.2呼叫法則(Invocation)是運作工作流程實作法則的兩種選擇。運作工作流程即是一個即時反應系統的內部運作，最自然的運作機制即是使用狀態機工作流程；而在一個即時反應系統中，通常會有一個以上的運作工作流程在執行，而這些工作流程會一直等待各自的事件到來，所以對於此流程的執行需要使用呼叫法則來實作。

在此工作流程模組內，我們分別完成3.2.3.1服務工作流程以及3.2.3.2運作工作流程這兩種不同的流程，目的是為了讓開發者在運作流程的開發過程中可以依據不同的功能性更容易使用，服務工作流程主要是針對一個系統內的所有裝置及流程在做管理控制，而運作工作流程則是對系統內單一裝置來進行處理的流程。分開這兩種工作流程是本架構的特點之一，如此一來開發者不但可以操縱整個系統的裝置及流程，也可以對單一裝置流程進行控制。

3.2.4 調控介面模組(Configuration Interface Module)

如圖 14所示，調控介面模組(Configuration Interface Module)以介面控制項的方式呈現給使用者，提供給使用者一個操控系統內參數的橋樑。此模組在本架構中提供兩種主要的功能：

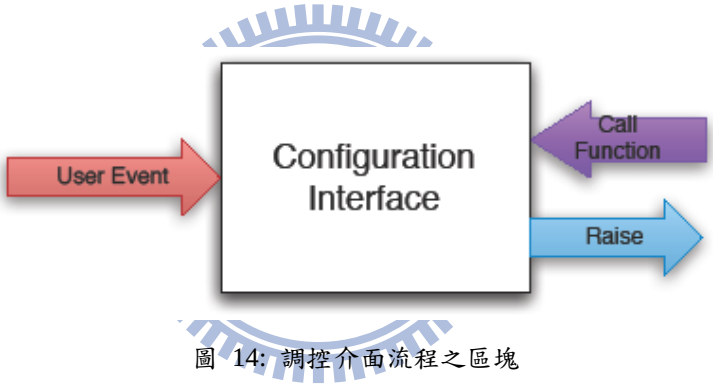

圖 14: 調控介面流程之區塊

- 接收使用者事件及調控流程：圖 14明顯看出此模組可以接收使用者所發出的行為事件，並對使用者所改變的控制項做出反應，這些反應會透過事件處理模組(Event Handler Module)進入到Workflow中調控自動化流程。
- 接收呼叫以更新顯示資訊：調控介面模組除了接收使用者事件外，還可以進行函式呼叫[圖 14, Call Function]，這些呼叫可以用以更新顯示工作流程中的狀態、變數，或UPnP裝置的加入移除等等資訊。

設計依據：調控介面模組的設計，目的在於將即時反應系統中所有可調控的事件、變數等等都以控制項的方式呈現在模組上，讓使用者在操控即時反應系統的過程中更加方便，如前段提到切換系統自動或手動化的事件—Automation Event，在此模組中就會被實作成一個可操縱的控制項。此模組所開發出的介面，可以是圖形使用者介面(Graphic User Interface, GUI)、網路操控介面(Web GUI)、甚至是最簡單的控制台(Console) [圖 15]。

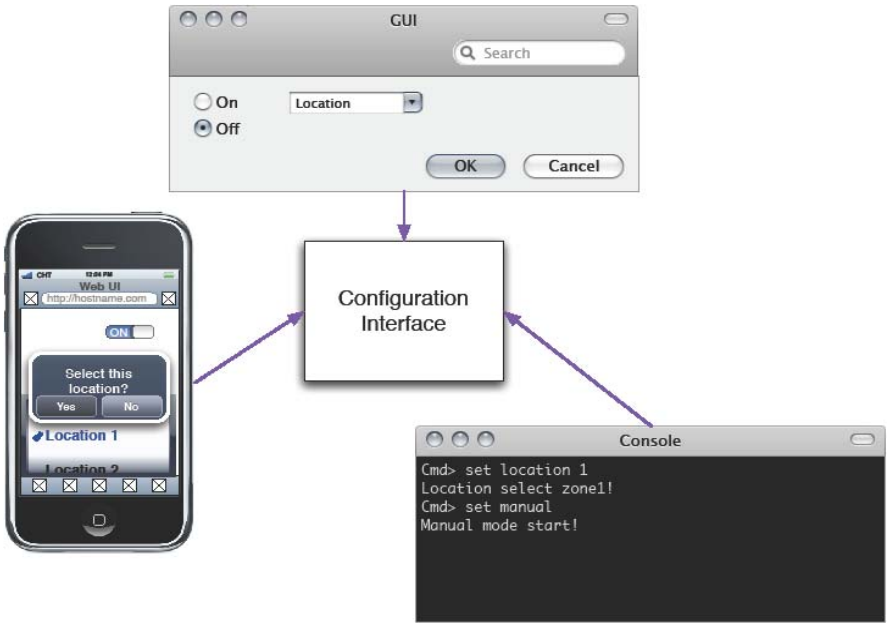

圖 15: 調控介面模組之應用

3.2.5 控制資料庫(Control Database Module)

控制資料庫(Control Database)其實是儲存機制的統稱，儲存了架構中控制流程時所需要的資訊，包括了 UPnP 裝置、服務、工作流程的名稱和獨特的 ID 等等，這些資訊會影響運作執行的方向，或決定事件傳遞工作流程個體。控制資料庫主要具有三種功能：

1. UPnP裝置加入網路內時，會將屬於此裝置的物件存入到裝置庫(DeviceMap)，移除時，物件也會從裝置庫中移除。除此之外，裝置庫還提供可以快速獲得裝置物件的資訊如裝置名稱、所在區域或直接取得所有UPnP裝置(Device)的服務(Service)，也提供一個偵測裝置是否仍存在的方法[表 2]。

表 2: 程式碼—簡易裝置庫類別(DeviceMap Class)

```
public class DeviceMap
{
      private Hashtable SensorMap = new Hashtable();
      private Hashtable ActuatorMap = new Hashtable();

      public Hashtable Get_SensorMap()
      public Hashtable Get_ActuatorMap()
      public UInt16 Get_ML_ID(UPnPDevice d)
      public String Get_Location(UPnPDevice d)
      public bool Exist_Sensor_Location(String
Location)
      //-----------
      public void Remove_Sensor(UPnPDevice d)
      public void Add_Sensor(UPnPDevice d)
      public void Remove_Actuator(UPnPDevice d)
      public void Add_Actuator(UPnPDevice d)
}
```

2. 圖 16可以看出當流程具有裝置服務(Service)、裝置本體(Device)、或流程個體ID (Workflow Instance ID)其中一個的資訊，就可以取得其他資訊，這是由於控制資料庫裡面實作了一個對應機制(Mapping)。對應機制利用.NET中標準函式庫—Dictionary來實作兩項對應：

- 服務對應裝置(Services to Devices)：UPnP 技術中，每個服務會屬於某一個裝置，每個裝置可以找到多個服務。由於在 UPnP 裝置的變數事件(Variable Event)發生時，僅是回傳引起此事件的服務，此時就要去對應找尋服務所在的裝置。
- 裝置對應工作流程 ID (Devices to Workflow IDs)：先前我們提到過，裝置加入到網路中會啟動一個對應的運作工作流程(Operation Workflow)，我們可以透過指定裝置得到欲控制的工作流程 ID。

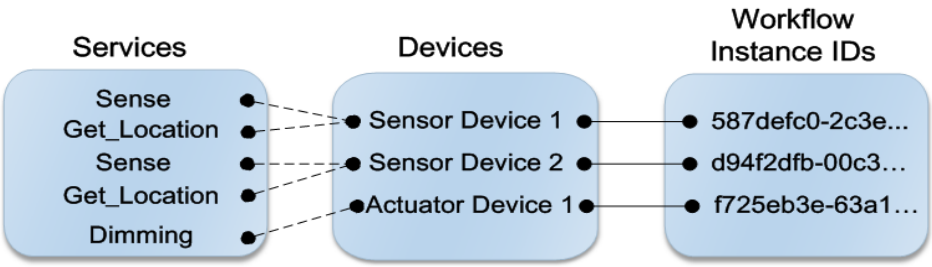

圖 16: Service、Device、Workflow ID 對應圖

3. 控制資料庫內的資料元件是以Public Static的方式實作[表 3]，開發者在使用整合封裝的模組時，控制資料庫提供獨一的靜態(Static)資料，使得模組即使整合到新的環境時，流程仍然可以正確無誤的執行。模組中使用Abstract Class來存放這些靜態物件(Static Object)，讓這個class成為獨一的物件。

表 3: 程式碼—靜態物件類別(Static Object Class)

```
public abstract class StaticObject
{
    public static Dictionary<String, Guid> Location_To_WorkflowID_Dict
                                = new Dictionary<string,Guid>();
    public static Dictionary<CpStandardizedLightSensor, UPnPDevice> CpStand_To_UPnPDevice_Dict
                                = new Dictionary<CpStandardizedLightSensor, UPnPDevice>();
    public static DeviceMap map = new DeviceMap();
    public static Guid DiscoveryInstanceId;
    public static LoSrvAutoDimming SLocalService = new LoSrvAutoDimming();
    public static SampleDevice AutoHighLevel;
  }
}
```

設計依據：控制資料庫內的資料決定了架構內一些重大的流程，工作流程個體 ID (Workflow Instance ID)是其中重要的資料元件，傳遞工作流程事件(Workflow Event)時需

要提供這個元件好讓事件處理模組可以傳遞到正確工作流程。本架構中的服務工作流程(Service Workflow)和運作工作流程(Operation Workflow)都會接收工作流程事件，所以儲存這兩種工作流程的實體 ID 是不可缺少的動作。動態配置工作流程也會根據資料庫內的裝置庫來決定。

3.3 運作流程及裝置的互動訊息

運作流程(Operations)和裝置(Devices)是構成此軟體開發架構兩個最大的基礎元素，而這兩者之間彼此的互動方式及關係是完成一個系統重要的部份。運作流程與裝置之間由事件(Events)及動作(Actions)來互動[圖 17]。而這邊的裝置通常是指透過架構內的UPnP控制點所偵測到智能環境裡的UPnP裝置。

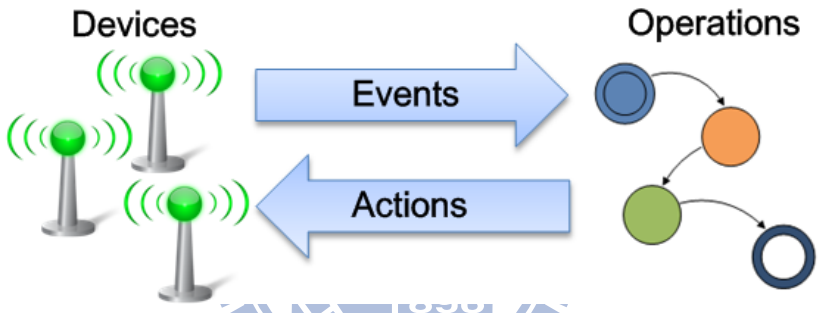

圖 17: 運作流程與裝置互動之傳遞機制

- 此架構內，裝置會以事件(Events)來觸發運作流程進行，當環境中的數值因子(如光度或濕度)有所改變的時候，偵測到的裝置會發出事件(Events)訊息讓運作流程收到，由於運作流程是以 WF 技術來呈現，外界裝置會將事件先交付給事件處理模組(Event Handler Module)，透過此模組將事件由外部事件處理轉換成屬於 WF 技術的內部事件再傳遞給運作流程，流程進行時，就可以順利接收裝置不停傳入的訊息。
- 運作流程則會以控制動作(Control Actions)的方式控制裝置，當即時性反應系統的運作進行到控制裝置的部份時，運作流程會以動作(Actions)的方式由 UPnP 控制點傳遞到智能環境內的裝置。

3.4 階層式封裝元件

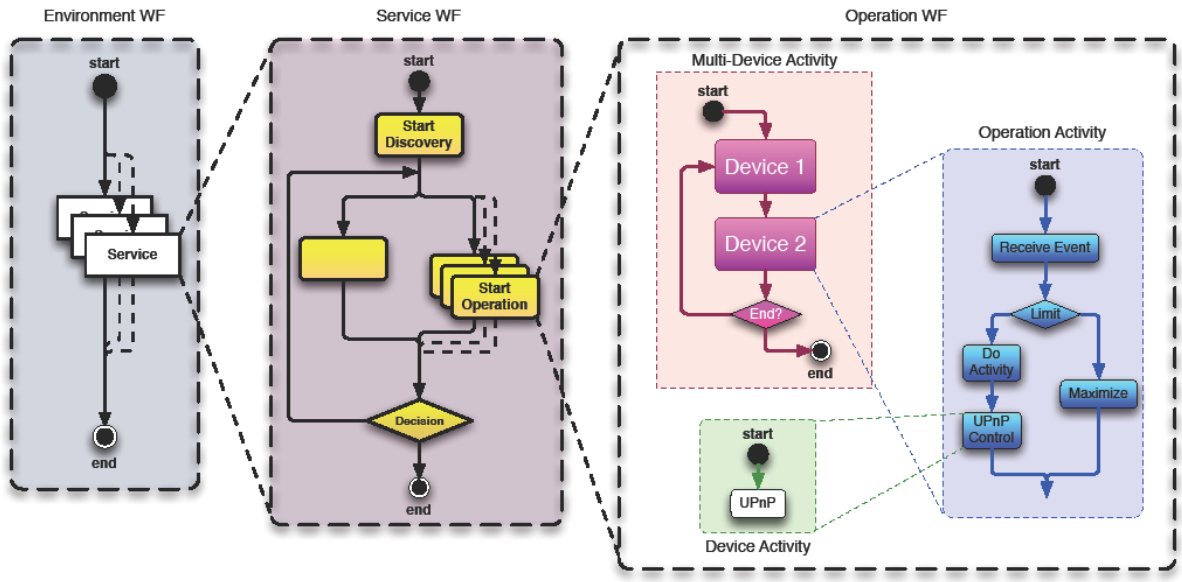

圖 18: 階層式軟體開發架構

在未來的環境裡，智能家電裝置的發展會隨著人們環境的需求而日漸成長，配合著家電裝置所產生出的即時反應系統也會越來越被重視，這些由開發者設計出的即時反應系統，預期將不單單只使用在單一家電裝置內，而是可以與多項家電裝置整合，形成更多功能的系統。在設計或整合一套系統的運作流程時，所有的設計都是在具有人性化拖曳及圖形化介面的WF技術平台上開發，若是這些設計開發的過程中，配合著軟體開發方式一起進行，將使得開發過程更加的快速，階層式封裝即是為此目的而使用在本架構中。圖18可以看出在一個系統的運作流程內，對於環境、即時反應系統、裝置與流程活動之間的階層關係，環境(Environment)內擁有且整合多個即時反應系統，而每個系統可以整合多個裝置(Device)或服務，裝置內的工作流程再以階層式活動(Activity)完成。本架構封裝共可分為活動元件階層、工作流程元件階層以及元件模組封裝：

3.4.1 活動元件階層封裝

活動元件(Activity)是WF技術中的最小元件，由多個功能性的活動元件組成一個系統的流程，這些活動元件在WF技術中可透過客製化活動(Custom Activity)的功能來完成，我們以一個系統運作流程所需要的功能來將這些活動做階層式的分類，每個活動都有其特定的功能，從最低層直接控制裝置的活動，中層具有即時反應功能的活動，到高層描述多個裝置彼此互相溝通的活動元件，這些活動元件的發展使得設計運作流程更具有彈性

[圖 19]。接下來將介紹這些活動元件的內容：

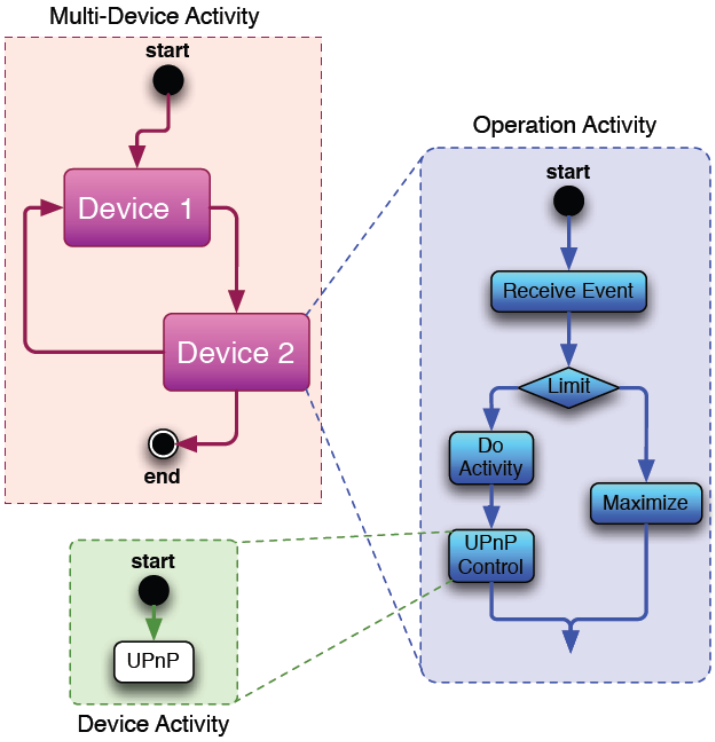

圖 19: 活動元件階層式封裝架構

- 裝置功能活動(Device Activity)：裝置功能活動在一個運作流程中屬於最低層的活動。活動中的標準程式碼活動(Code Activity)，將 UPnP 控制點(Control Point)操控裝置所提供功能的程式碼封裝在其中，對於運作流程要操縱家電裝置的功能，是絕對不可少的活動。這樣的活動並沒有包含任何的自動化功能，使用者將直接給予動作和變數來操控 UPnP 裝置。
- 運作流程活動(Operation Activity)：是操控一個智能家電裝置的主要運作流程。此活動為了與 UPnP 裝置做溝通，需加入欲操控的裝置功能活動。也會實作運作流程，根據從外部環境接收到的數值判斷接下來欲進行的活動，在將判斷的數值交給已封裝完成的裝置功能活動，來完成運作流程活動。
- 多裝置整合活動(Multi-Device Activity)：智能家電裝置在智能環境中，其實並不是以一個獨立的個體出現，環境中的各項裝置對於彼此是會互相影響的，如對於空調裝置來說，燈光裝置的亮度會影響到室內溫度，而再影響到一個家中的睡眠系統。將多個已封裝的運作流程活動加入在同一活動內，再透過控制流程將它們整合後封裝，最高層的多裝置整合活動即完成。圖 19為感測器裝置和燈光控制器裝置形成

的一個多裝置整合活動的例子。

- 演算法活動：在運作流程中，會有許多的控制流程的決定會經過複雜的演算法來計算，而這些複雜的演算法可能因為不同的裝置或不同的開發者而會更動，演算法活動即產生。與低層裝置功能活動相同，演算法活動將以程式碼活動(Code Activity)做為核心，將複雜的演算法計算封裝在內，再給開發者在設計運作流程時加入使用，當演算法改變的時候，抽換演算法活動也將會更容易且快速。

3.4.2 工作流程元件階層封裝

工作流程元件也具有階層式封裝，對於架構中所擁有的Workflow，通用可分為三階層[圖20]，在封裝架構中都有各自的功能特色以及彼此之間階層式運用：

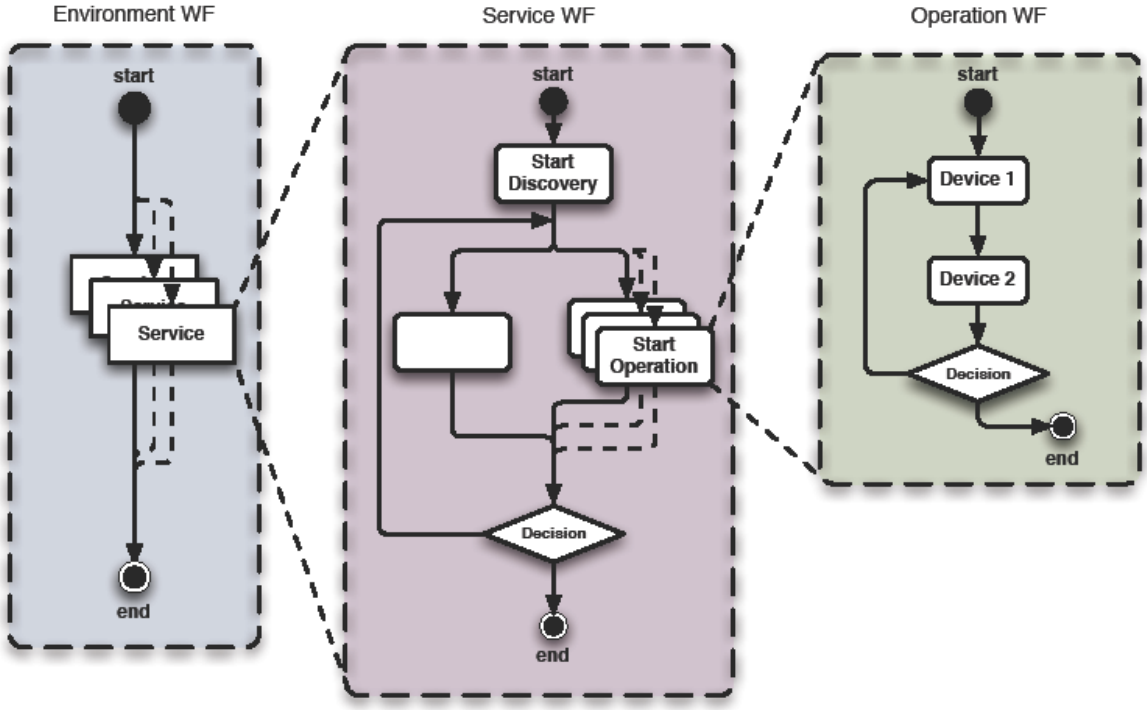

圖 20: 工作流程階層式封裝架構

- 運作工作流程(Operation Workflow)：即為3.2.3.2的元件模組，在階層式封裝過程中，是屬於最低層的元件，其內容即為前一節的整體活動元件，開發者在設計開發系統流程時，可整合多個封裝活動元件完成工作流程。
- 服務工作流程(Service Workflow)：為3.2.3.1的模組，加入了自動偵測裝置進入的功能以及可以呼叫原先低層的運作工作流程。在工作流程元件階層中，服務工作流程是一個最完整系統流程，完整的從自動在網路上取得裝置加入，到開始進行即時反

應流程，所以此部份的封裝是系統中不可或缺的。

- ❖ 環境工作流程(Environment Workflow)：環境工作流程是系統中最上層的工作流程，管理了智能環境裡所有的即時反應系統的啟動與終止。服務工作流程則是在此流程中加入或移除的目標元件。

3.4.3 模組元件封裝

當開發者要整合已開發的系統擴展形成一個新的系統時，模組元件的加入絕對是不可少的，除了前述在 WF 技術中加入的服務工作流程及運作工作流程之外，其他必要的模組元件或其功能也要進行封裝且一同加入，以下則是欲封裝的模組元件：

- ❖ 控制點模組啟動：控制點模組是在WF技術中啟動的，我們將控制點模組的啟動機制封裝成一個活動元件[圖 21]，提供各開發者更快的在Workflow中加入此模組的啟動機制。

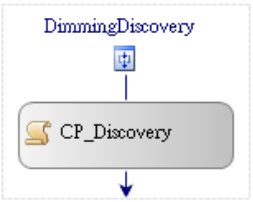

圖 21: 活動—控制點啟動封裝

- ❖ 事件處理模組：在啟動工作流程執行引擎(WF Runtime Engine)之前，就要將欲執行的即時反應系統的事件處理模組加入到引擎內，而事件處理模組需要是有一個完整的封裝才能更方便的使用[表 4]。執行引擎可以加入不只一個事件處理模組。

表 4: 程式碼—加入事件處理模組機制

```
ExternalDataExchangeService exchangeService = new ExternalDataExchangeService();
workflowRuntime.AddService(exchangeService);
exchangeService.AddService(StaticObject.SLocalService);
```

- ❖ 調控介面模組：若是以 Windows Form 產生出來的調控介面模組，其被封裝的控制項元件，可與其他的控制項元件進行擴充與整合，做成更大的使用者介面。

這些模組或變數名稱在各自的即時反應系統中都可以任意命名，也就是說，雖然命名的名稱相同，但是根據命名空間(Namespace)可以分辨出是屬於哪個即時反應系統，封裝成函式庫加入到新的設計時，是不會混淆的。

第四章 運作理念

在這個章節將深入瞭解「基於工作流程與通用隨插即用技術下之整合性軟體開發架構」的內部運作技術，探討的技術包括：

- UPnP 裝置與 Workflow 模組之間的互動機制
- 發現機制動態配置工作流程

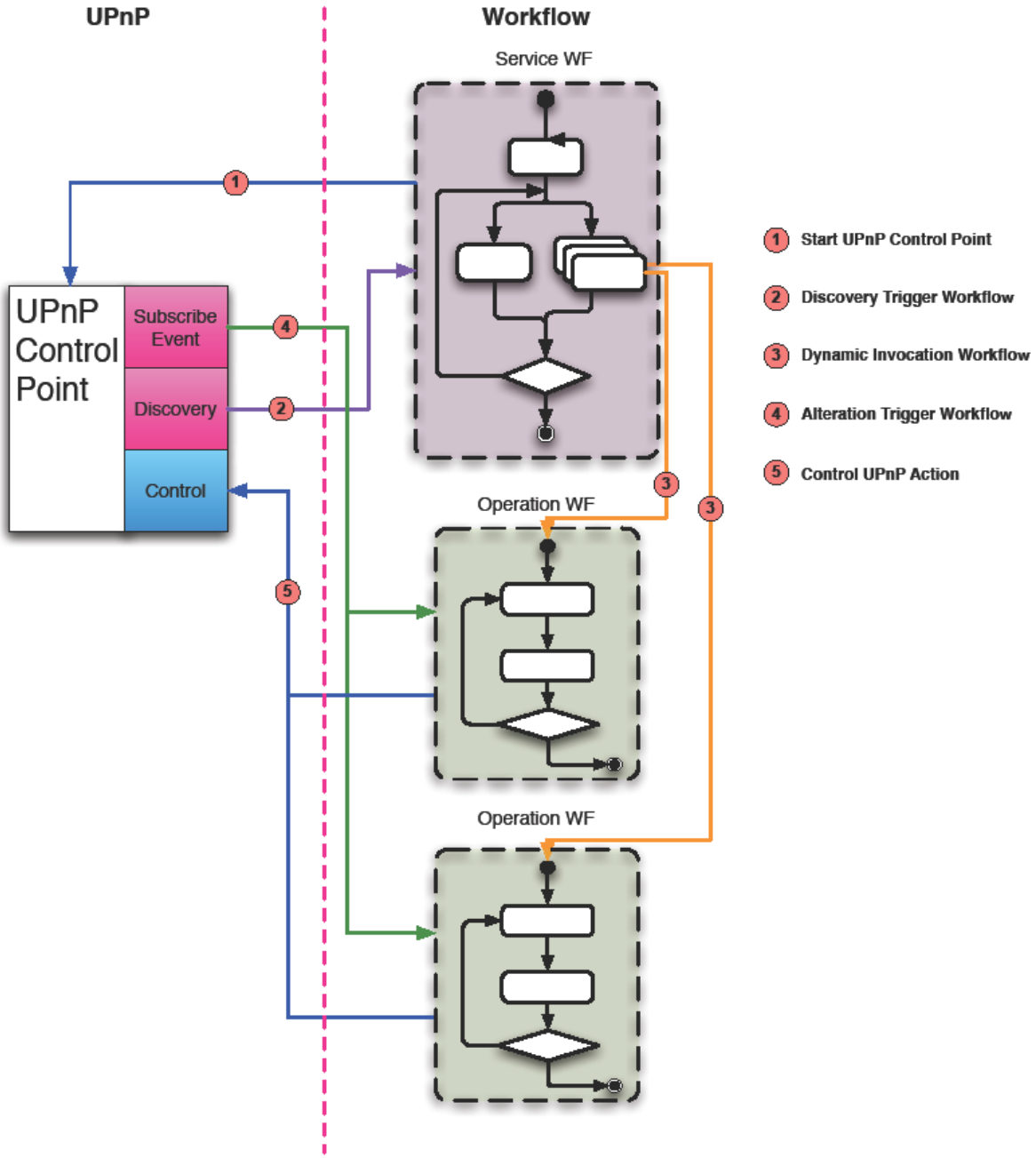

圖 22: UPnP 與 WF 技術互動流程

本架構運用UPnP以及WF兩大技術完成，在架構內最主要的運作流程都將會運用這兩項技術，圖 22即是本章節在這兩大技術間所探討此架構的內部運作技術。除此之外，本章節還會介紹如何在此軟體開發架構上使用這些運作技術來開發完成一項即時反應系統。

4.1 UPnP裝置與Workflow元件的互動機制

在第三章，我們談到本架構的元件模組，瞭解大部分的元件模組都與UPnP和Workflow有關係。接下來的部份將依照此架構的運作順序搭配圖 22來介紹兩者之間的關係與機制：

1. 工作流程(Workflow)啟動 UPnP 控制點

   當即時反應系統啟動後，執行引擎會開始執行的服務工作流程，此系統為了取得或調控環境內裝置的數值，如圖 22線條(1)，服務工作流程會馬上啟動UPnP控制點，儘快偵測UPnP裝置的存在及取得對它的操控，而UPnP控制點啟動則是將Discovery機制開啟即可[表 5]。這個啟動的機制在3.4.3模組元件封裝中的控制點模組啟動有提過將其封裝成活動元件，需啟動時則直接在服務工作流程中加入啟動的活動元件即可。

表 5: 程式碼—啟動 UPnP 控制點

```
//Actuator
Actuator_disco.OnAddedDevice += new DimmableLightDiscovery.DiscoveryHandler(Actuator_AddSink);
Actuator_disco.OnRemovedDevice += new DimmableLightDiscovery.DiscoveryHandler(Actuator_RemoveSink);
Actuator_disco.Start();
//Sensor
Sensor_disco.OnAddedDevice += new StandardizedLightSensorDiscovery.DiscoveryHandler(Sensor_AddSink);
Sensor_disco.OnRemovedDevice +=
      new StandardizedLightSensorDiscovery.DiscoveryHandler(Sensor_RemoveSink);
Sensor_disco.Start();
```

2. UPnP 裝置加入之事件觸發 Workflow

   啟動UPnP控制點後，如圖 22線條(2)，UPnP控制點中的發現機制會偵測到網路上UPnP裝置的加入，當新裝置加入，裝置會先存入控制資料庫的裝置庫(DeviceMap)，接著馬上透過事件處理機制觸發內部事件讓服務工作流程接收[表 6]。

表 6: 程式碼—裝置加入後的處理

```
private static void Actuator_AddSink(DimmableLightDiscovery sender, UPnPDevice d)
{
     StaticObject.map.Add_Actuator(d);
     String _location = StaticObject.map.Get_Location(d);

     Dictionary<String, object> WfArgs = new Dictionary<string, object>();
     WfArgs.Add("Location", _location);
     try
     {
          StaticObject.SLocalService.RaiseEvent(EventTypes.Discovery,
              new WorkflowEventArgs(StaticObject.DiscoveryInstanceId, WfArgs));
     }
     catch { }
}
```

3. UPnP 裝置改變之事件觸發 Workflow

啟動運作工作流程後，即時性反應系統即開始運行。系統最主要的服務是接收環境中的數值來進行裝置自動的調適改變。如圖 22線條(4)，運作工作流程不停地在接收UPnP裝置改變的事件，每當裝置內的數值有改變的時候，UPnP控制點會先從控制資料庫中搜尋改變的事件是由哪個裝置發出的，進而去得到對應的工作流程個體 ID，再透過事件處理模組將事件讓運作工作流程接收[表 7]。

表 7: 程式碼—裝置改變後的處理

```
static void StandardizedLightSensor_OnStateVariable_result
                (CpStandardizedLightSensor sender, byte NewValue)
{
     UPnPDevice d;
     String _location;
     Guid SMInstanceID;

     d = StaticObject.CpStand_To_UPnPDevice_Dict[sender];
     _location = StaticObject.map.Get_Location(d);
     if (StaticObject.Location_To_WorkflowID_Dict.ContainsKey(_location))
     {
          SMInstanceID = StaticObject.Location_To_WorkflowID_Dict[_location];
          Dictionary<String, object> WfArgs = new Dictionary<string, object>();
          WfArgs.Add("ML_ID", StaticObject.map.Get_ML_ID(d));
          WfArgs.Add("SensorValue", NewValue);
          StaticObject.SLocalService.RaiseEvent(EventTypes.Sense,
               new WorkflowEventArgs(SMInstanceID, WfArgs));
     }
}
```

4. Workflow 操控 UPnP 裝置行為

如圖 22線條(5)，運作工作流程接收到事件後，開始做自動化的計算處理，最後在流程中再加入低層的直接控制裝置的活動元件。低層透過UPnP控制點操控裝置的程式碼[表 8]。

表 8: 程式碼—低層控制裝置

```
CpSwitchPower SwitchPower = new CpSwitchPower(d.GetServices(CpSwitchPower.SERVICE_NAME)[0]);
SwitchPower.Sync_SetTarget(true);

CpDimming Dimming = new CpDimming(d.GetServices(CpDimming.SERVICE_NAME)[0]);
Dimming.Sync_SetLoadLevelTarget((byte)DimmingValue);
```

4.2 發現機制動態配置工作流程

運作工作流程是裝置在即時反應系統流程中運作的核心，其每個裝置在各自的工作流程中，有著屬於自己的狀態及變數。為了配合著裝置可隨插即用的 UPnP 技術，動態配置工作流程這樣的自動化偵測且配置流程的功能即產生，這些工作流程的配置，需要有功能性的方法進行管理，如下列功能：

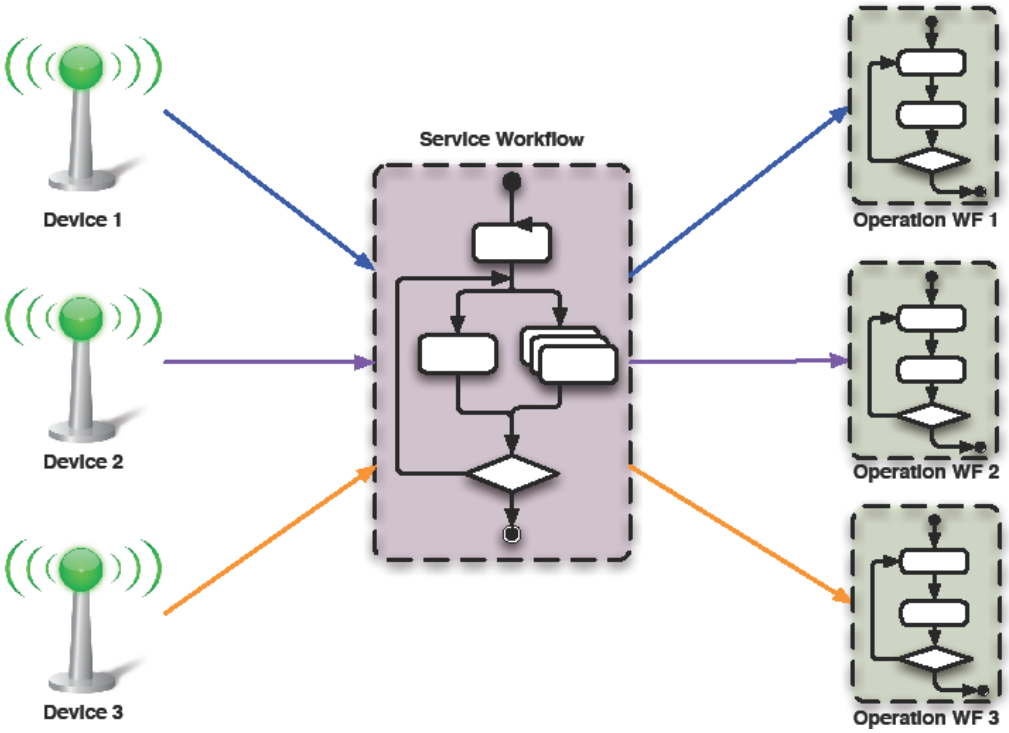

圖 23: 裝置透過服務工作流程動態對應運作工作流程

1. 呼叫流程功能：這是動態啟動一個工作流程的機制，對於每個加入網路的UPnP裝置都有其ID或Location等等的特定變數，在服務工作流程內針對這些裝置的特定變數做判斷處理。如圖 22線條(3)，當加入未出現過的裝置，此機制會使用呼叫法則(Invocation)啟動屬於此裝置的運作工作流程，藉此來完成系統運作流程；反之，若是服務工作流程得到裝置的特定變數已經啟動，則不進行任何處理。透過動態的呼

叫流程機制，我們可以確保每一個裝置都有一個特定的運作工作流程配合運行[圖23]。

2. 裝置對應流程：動態配置工作流程判斷所需的資料—工作流程個體 ID 以及裝置特定變數，是在執行一個運作工作流程的最一開始即被一起存入的，接著動態配置工作流程時就能根據控制資料庫(Control Database)的裝置是否存在做判斷。

3. 移除流程功能：即時反應系統會隨著裝置從網路中移除後而停止，在 WF 技術中運行的運作工作流程此時也終止移除掉。對於動態地移除正確的運作工作流程，是當移除發生時直接讓裝置對應的運作工作流程接收裝置所發出的移除事件(Removal Event)，進而進入終止狀態來結束掉流程。除了結束掉流程外，控制資料庫也會因為裝置及流程的終止進而移除掉其內部的資料。

第五章 應用

5.1 室內燈光回饋控制

論文「適用於家庭自動化的通用隨插即用感測與促動器基礎架構」[7]在交通大學電資大樓智能環境實驗室(NCTU MIRC Smart Environment Lab)[圖 24]中已實作完成室內燈光回饋控制，整合其中層標準裝置以及低層UPnP燈光控制系統，重新以「基於工作流程與通用隨插即用技術下之整合性軟體開發架構」來完成高層即時反應系統的運作流程部份，並階層式封裝燈光控制元件以利於室內燈光回饋控制系統能重複使用。

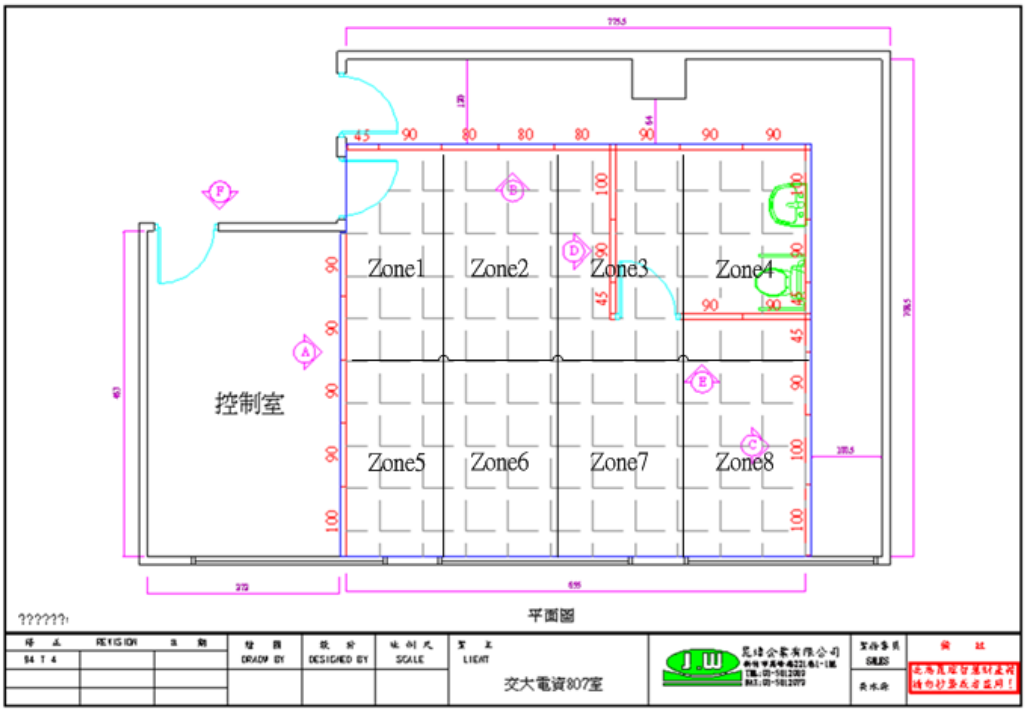

圖 24: 交通大學電資大樓智能環境實驗室

5.1.1 實作開發流程

在實驗室中共分為八塊區域(Zone1~Zone8)，每塊區域都預先安裝了調光器，是以 Lite-Puter 公司開發可以接收 RS485 控制訊號的前端燈光控制系統(Lighting Control System)來控制調光器，並使用 Crossbow 公司的 Micaz 佈置無線感測網路，透過 UPnP 代理伺服器來取得光感測數值。

利用2.1.2的Intel Tool產生三個UPnP裝置：

- 低層裝置—與實際硬體配備溝通
- 中層裝置—控制低層裝置所有功能
- UPnP 控制點—控制所有中層裝置

而UPnP控制點，即是3.2.1的UPnP控制點模組。並將控制點模組的功能及裝置，加入到 3.2.5的控制資料庫中，並在偵測到裝置變化的時候，實作對應的事件觸發機制。

接下來我們加入了WF技術，首先是完成3.2.2的事件處理模組，根據不同的功能加入了五項事件：

- Discovery Event—調光器新加入到網路中
- Removal Event—調光器從網路中移出
- Sense Event—光感測器接收到環境中光亮度改變
- Dimming Event—使用者對於調光器亮度的改變
- Automation Event—使用者決定是否啟動自動反應服務

接著在WF技術中設計3.2.3的服務工作流程，首先開始啟動UPnP控制點來偵測裝置。在調光器裝置進入網路後，服務工作流程使用4.2動態配置運作工作流程的時候，我們根據了圖 24的區域來對每個調光器裝置配置對應的流程，每個區域有其對應的調光器、光感測器以及運作工作流程。

室內燈光回饋控制系統是一個即時反應系統，其運作流程如圖 25。在每個運作工作流程內，系統首先都會接收到由光感測器送出的環境光亮值，其數值只會傳入與光感測器在相同區域的流程。接收到未經處理的光數值後，進行的感測演算法(Sense Algorithm)來取得平衡的區域光亮值，再經由控制演算法(Control Algorithm)決定最後欲控制的調光器數值，交由UPnP控制點傳給調光器進行控制。

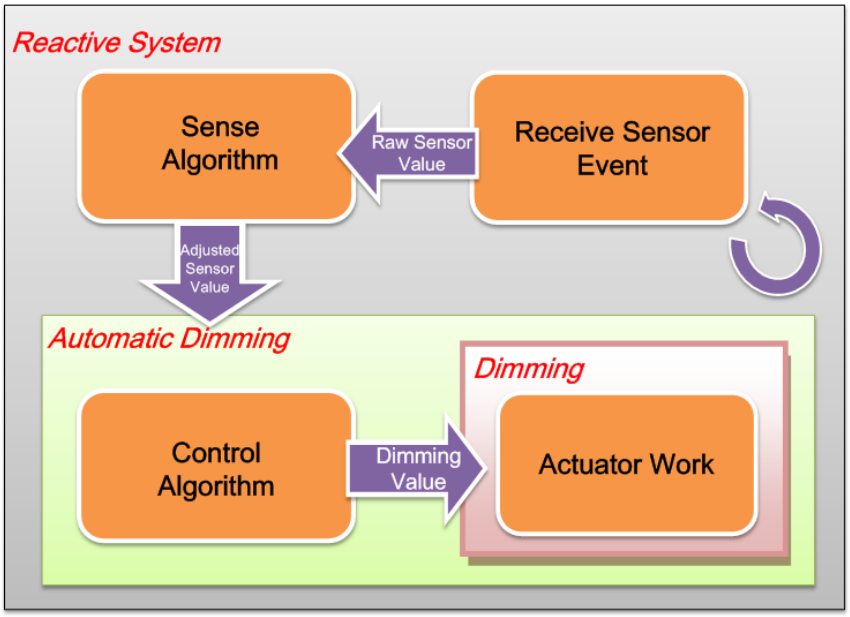

圖 25: Reactive System 運作流程

對於在燈光回饋系統內的演算法說明如下：

- 感測演算法(Sense Algorithm)—由於在佈置無線網路光感測器時，在一個區域內會佈置不只一個的感測器，而這些感測器與燈光的距離並不一定相同，在物理意義上這些感測器不能依照平分的計算來決定最後平衡的光亮度，而必須根據感測演算法來決定。
- 控制演算法(Control Algorithm)—室內燈光回饋控制系統主要是利用外界環境既有的亮度，與燈光給予的亮度兩者進行室內燈光亮度的調整，當外界環境亮度很亮的時候，可以根據控制演算法使得調光器亮度數值不用很高，就可以讓整個室內環境達到使用者所設定的光平衡狀態。

室內燈光回饋控制系統除了使用Intel Tool中的Device Spy進行控制以外，也可以實作3.2.4的調控介面模組(Configuration Interface Module)[圖 26]來完成，其控制模組的內容說明如下：

- Location—當調光器加入到網路中，控制項可以選擇目前有調光器的區域來進行操控
- Automation—針對選擇好的調光器來決定是否開啟自動調整燈光模式
- Level—直接決定環境中的燈光數值

- Color—顯示調光器目前的亮度

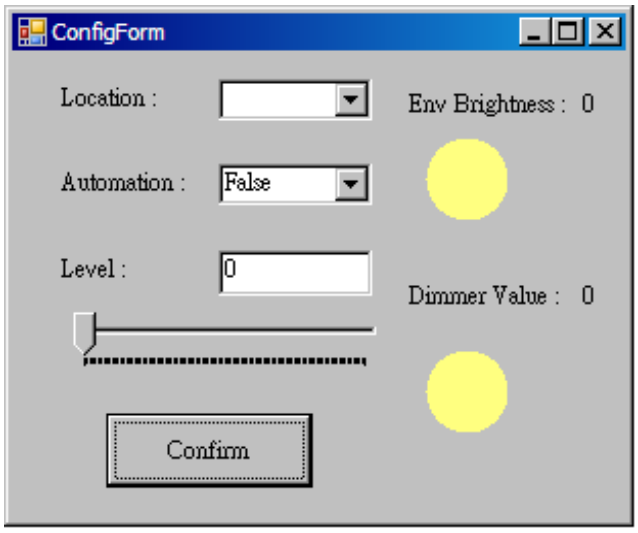

圖 26: 調控介面模組實作

5.1.2 封裝流程

為了讓其他開發者能簡易且快速的重複利用此室內燈光回饋控制系統，我們將此系統的功能包裝起來成為下列的活動：

- HL_ReactiveSystem — 如圖 25的包含接收光感測器裝置數值以及經由演算法決定控制調光器的數值，是一個完整的即時反應系統
- ML_AutoDimming — 對於調光器而言，一個根據外界傳入的數值而自動化計算調光器欲調整的亮度
- LL_Dimming — 單純接收一個數值後，直接對調光器做數值改變
- LL_ReadSensor — 可直接讀取光感測器當下的數值

5.2 智慧型置物櫃(Smart Pantry)

智慧型置物櫃(Smart Pantry)是我們利用此軟體開發架構所完成的另一項應用，我們根據「Smart pantries for homes」[8]論文中所提供的情景模式，簡化成幾種較普遍且為正確使用下的流程狀態[圖 27]。

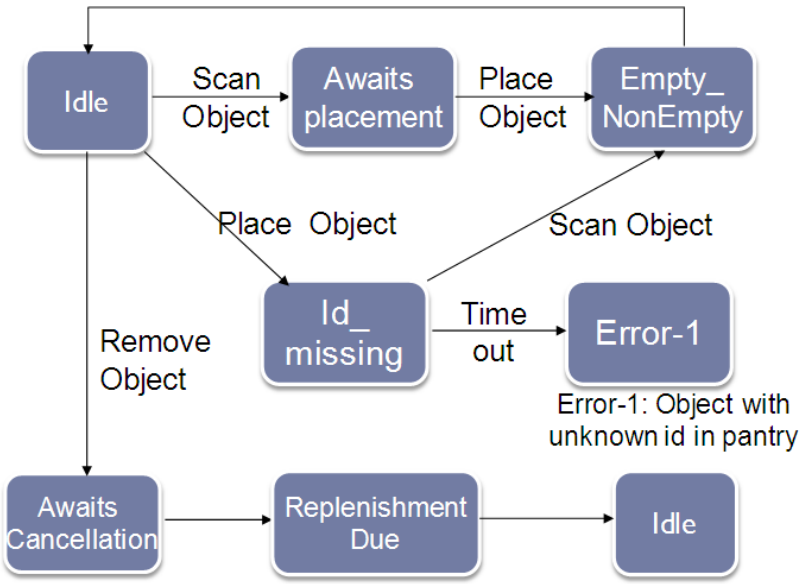

圖 27: Smart Pantry 狀態流程圖

探討三個普遍的情景：

情景 1：物品先經由 Bar Code 掃描器掃描後，再將物品放入置物櫃中。

情景 2：物品先放入置物櫃中，再等待 Bar Code 掃描器取得資訊。

情景 3：物品從置物櫃中取出。

以此架構開發出的智慧型置物櫃系統運作流程中，我們仍然分為服務工作流程[圖 28]以及運作工作流程[圖 29]來完成，服務工作流程除了偵測是否有裝置進入之外，同時也偵測是否有Bar Code掃描器的掃描事件傳入，接著將Bar Code存入Queue中，由於Bar Code的掃描是屬於整個系統流程的動作，所以Bar Code掃描的部份我們會在服務工作流程內來完成。運作工作流程則是會等待接收物品放入置物櫃的事件產生，再去根據是否有被儲存下來的Bar Code來判斷，若是目前Queue有儲存Bar Code，則代表已經有物品被掃描並要放入，則會進行情景 1 的模式，將物品Bar Code、物品名稱等等的資訊存入裝置中；若是Queue中並無Bar Code，則會進行情景 2，等待掃描器掃描Bar Code的事件後，將資訊存入裝置。當運作工作流程接收到物品取出的事件時，則會清除置物櫃中物品資訊。這邊我們在儲存Bar Code時以Queue的方式來實作，主要是可以解決當有多人以上欲掃描Bar Code時，會有一個先後順序，也就是先掃描的人可以完成整個放入物品的動作，之後掃描Bar Code的人必須等待前人完成後，才能放入物品。對於無法預測多人放入物品的先後順序導致的Error，這部份我們暫時不探討。

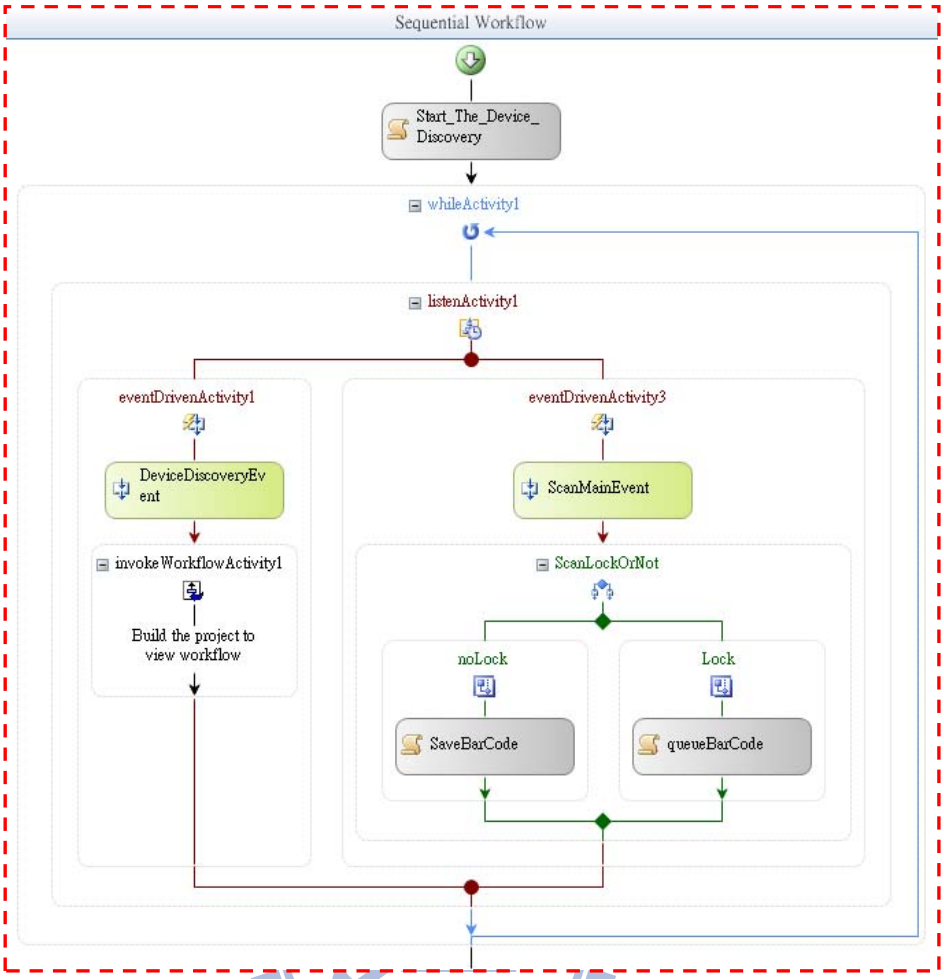

圖 28: Smart Pantry 之服務工作流程

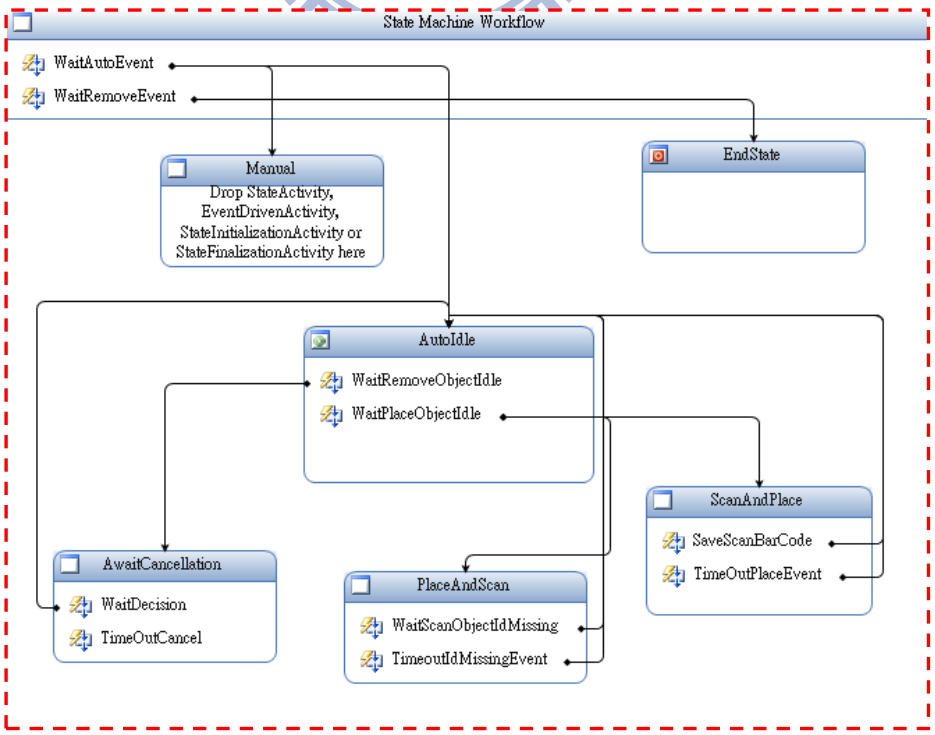

圖 29: Smart Pantry 之運作工作流程

第六章 結論

6.1 研究成果

在這個研究中，我們完成了：

- 定義「基於工作流程與通用隨插即用技術下之整合性軟體開發架構」的五大功能模組彼此之間的關係及功用[圖 9]。
- 建立事件處理模組(Event Handler Module)來協調運作流程以及裝置之間的事件傳遞運作。
- 開發以 Service Workflow 和 Operation Workflow 這兩種工作流程為主軸的運作流程設計方式，並設計其互動及動態配置關係，來讓開發者可以利用這樣的設計方式來完成一套完整的系統運作流程。
- 設計階層式封裝的軟體開發方式，配合上述的工作流程分為工作流程階層式封裝以及活動階層式封裝，讓開發者可以藉以參考完成一套具有高度整合性的運作流程。
- 此架構整合論文「適用於家庭自動化的通用隨插即用感測與促動器基礎架構」中的燈光裝置，在交通大學電資大樓智能環境實驗室中實作完成室內燈光回饋控制，並封裝階層式燈光控制元件以利於重複使用。另外根據「Smart Pantries for Homes」[8]論文，以此架構完成Smart Pantry的UPnP虛擬裝置之開發以及即時反應系統之運作流程發展。

6.2 未來方向

對於「基於工作流程與通用隨插即用技術下之整合性軟體開發架構」來說，仍有一些待解決的問題可以讓此架構變得更加完善，問題如下：

- 事件處理模組部份，由於目前的事件處理模組是根據已知的事件來進行事件的轉換，所以每當具有新的事件欲加入到系統內被接收時，則需要重新編譯這個事件處理模組。所以若是能完成一個具有動態偵測及轉換事件的模組，則會使得設計架構變得

更具有彈性，且大大地提昇開發效率，這樣的模組會是一個可研究的課題。

- ❖ Service Workflow 的功能主要是進行 UPnP 裝置進入的偵測，對於即時反應系統的運作流程來說，可能會具有多個不同的運作模式可以作選擇，如室內燈控系統可能具有白天模式、夜晚模式、派對模式等等。這樣的選擇如何配合著 Service Workflow 來完成，其設計的方式仍待研究。
- ❖ 對於系統中 Operation Workflow 的執行法則，目前我們是採用 Invocation 的執行方式讓每一個裝置具有一個運作工作流程且在各自的執行緒(Thread)內執行，這樣的執行方式是否太耗費資源，以及是否可以運用 Encapsulation 配合輪流(polling)的方式來完成，是可以進行分析及研究。

參考文獻

[1] Edward Lee, “The Ptolemy Project”, Universal of California at Berkeley.

[2] UPnP™ Device Architecture, http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0.pdf

[3] UPnP Forum™, http://www.upnp.org/

[4] Microsoft, “Understanding Universal Plug and Play”, White Paper.

[5] Intel® Software for UPnP Technology, http://software.intel.com/en-us/articles/intel-software-for-upnp-technology-download-tools

[6] Windows Workflow Foundation, http://msdn.microsoft.com/zh-tw/library/ee210343(en-us).aspx

[7] Yu-Chih Liu, “UPnP Compatible Sensor/Actuator Infrastructure for Home Automation”, National Chiao Tung University, Degree of Master Thesis, July 2007

[8] C. F. Hsu, Y. H. Liao, P. C. Hsiu, Y. S. Lin, C. S. Shih, T. W. Kuo, and J. W. S. Liu, “Smart pantries for homes”, in Proceedings of IEEE SMC, October 2006.

附錄一 室內燈控系統程式碼

室內燈控系統(Indoor Luminance System)的 Main Program

```
class Program
{
        [MTAThread]
        static void Main(string[] args)
        {
            using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
            {
                AutoResetEvent waitHandle = new AutoResetEvent(false);
                workflowRuntime.WorkflowCompleted +=
                        delegate(object sender, WorkflowCompletedEventArgs e) {waitHandle.Set();};
                workflowRuntime.WorkflowTerminated +=
                        delegate(object sender, WorkflowTerminatedEventArgs e)
                {
                    Console.WriteLine(e.Exception.Message);
                    waitHandle.Set();
                };
                ExternalDataExchangeService exchangeService = new ExternalDataExchangeService();
                workflowRuntime.AddService(exchangeService);
                exchangeService.AddService(StaticObject.SLocalService);

                workflowRuntime.StartRuntime();
                WorkflowInstance instance =
                        workflowRuntime.CreateWorkflow(typeof(UPnPInWF.SmartEnvWF));
                instance.Start();

                waitHandle.WaitOne();
                Console.WriteLine("End");
                Console.ReadLine();
            }
    }
}
```

室內燈控系統(Indoor Luminance System)的事件處理模組

```
public enum EventTypes { Discovery, Remove, Sense, UserDimming, Automation };

public class LoSrvAutoDimming : ILoSrvAutoDimming
{
    public event EventHandler<WorkflowEventArgs> SenseEvent;
    public event EventHandler<WorkflowEventArgs> DiscoveryEvent;
    public event EventHandler<WorkflowEventArgs> UserDimmingEvent;
    public event EventHandler<WorkflowEventArgs> RemoveEvent;
    public event EventHandler<WorkflowEventArgs> AutoEvent;
    public event EventHandler<WorkflowEventArgs> Events;

    public virtual void RaiseEvent(EventTypes eventType,WorkflowEventArgs args)
    {
        switch (eventType)
        {
            case EventTypes.Discovery:
                Events = DiscoveryEvent;
                break;
            case EventTypes.Remove:
                Events = RemoveEvent;
                break;
            case EventTypes.Sense:
                Events = SenseEvent;
                break;
            case EventTypes.UserDimming:
                Events = UserDimmingEvent;
                break;
            case EventTypes.Automation:
                Events = AutoEvent;
                break;
        }

        if (Events != null)
            Events(null, args);
    }
}
```

室內燈控系統(Indoor Luminance System)的裝置庫(DeviceMap)

```
public class DeviceMap
{
    private Hashtable SensorMap = new Hashtable();
    private Hashtable ActuatorMap = new Hashtable();
    //Get Sensor Device Map
    public Hashtable Get_SensorMap()
    {
        return SensorMap;
    }

    //Get Actuators' Device Map
    public Hashtable Get_ActuatorMap()
    {
        return ActuatorMap;
    }

    //Get Sensors' Device Map
    public UInt16 Get_ML_ID(UPnPDevice d)
    {
        UInt16 ML_ID = 0;
        CpMidLevel_Information MidLevel_Information = new
            CpMidLevel_Information(d.GetServices(CpMidLevel_Information.SERVICE_NAME)[0]);
        MidLevel_Information.Sync_Get_ML_ID(out ML_ID);
        return ML_ID;
    }

    //Get Devices' Location
    public String Get_Location(UPnPDevice d)
    {
        String location_tmp = "";
        CpMidLevel_Information MidLevel_Information = new
            CpMidLevel_Information(d.GetServices(CpMidLevel_Information.SERVICE_NAME)[0]);
        MidLevel_Information.Sync_Get_Location(out location_tmp);
        return location_tmp;
    }

    //Check Sensors Exist
    public bool Exist_Sensor_Location(String Location)
    {
        object[] keys = new object[SensorMap.Count];
        SensorMap.Keys.CopyTo(keys, 0);
        foreach (object key in keys)
        {
            UPnPDevice d = (UPnPDevice)SensorMap[key];
            String Location_tmp = "";
            try
            {
                Location_tmp = this.Get_Location(d);
            }
            catch
            {
                SensorMap.Remove(key);
                Console.WriteLine("Removed Sensor Device: " + d.FriendlyName);
            }
            if (Location_tmp.Equals(Location))
            {
                return true;
            }
        }
        return false;
    }
```

```
        //Remove Sensors from Device Map
        public void Remove_Sensor(UPnPDevice d)
        {
            lock (SensorMap)
            {
                object[] keys = new object[SensorMap.Count];
                SensorMap.Keys.CopyTo(keys, 0);
                foreach (object key in keys)
                {
                    if (SensorMap[key].Equals(d))
                    {
                        SensorMap.Remove(key);
                        Console.WriteLine("Removed Sensor Device: " + d.FriendlyName);
                    }
                }
            }
        }

        //Add Sensors to Device Map
        public void Add_Sensor(UPnPDevice d)
        {
            UInt16 ML_ID = 0;
            ML_ID = this.Get_ML_ID(d);
            SensorMap.Add(ML_ID, d);
            Console.WriteLine("Added Sensor Device: " + d.FriendlyName);
        }

        //Remove Actuators from Device Map
        public void Remove_Actuator(UPnPDevice d)
        {
            lock (ActuatorMap)
            {
                object[] keys = new object[ActuatorMap.Count];
                ActuatorMap.Keys.CopyTo(keys, 0);
                foreach (object key in keys)
                {
                    if (ActuatorMap[key].Equals(d))
                    {
                        ActuatorMap.Remove(key);
                        Console.WriteLine("Removed Actuator Device: " + d.FriendlyName);
                    }
                }
            }
        }

        //Add Actuators to Device Map
        public void Add_Actuator(UPnPDevice d)
        {
            String Location = "";
            Location = this.Get_Location(d);
            ActuatorMap.Add(Location, d);
            Console.WriteLine("Added Actuator Device: " + d.FriendlyName);
        }
    }
```

```
class Discovery
{
    private static DimmableLightDiscovery Actuator_disco = new DimmableLightDiscovery();
    private static StandardizedLightSensorDiscovery Sensor_disco =
        new StandardizedLightSensorDiscovery();

    public Discovery()
    {
        System.Console.WriteLine("Intel's UPnP .NET Framework Stack");
        //Actuator
        Actuator_disco.OnAddedDevice +=
            new DimmableLightDiscovery.DiscoveryHandler(Actuator_AddSink);
        Actuator_disco.OnRemovedDevice +=
            new DimmableLightDiscovery.DiscoveryHandler(Actuator_RemoveSink);
        Actuator_disco.Start();
        //Sensor
        Sensor_disco.OnAddedDevice +=
            new StandardizedLightSensorDiscovery.DiscoveryHandler(Sensor_AddSink);
        Sensor_disco.OnRemovedDevice +=
            new StandardizedLightSensorDiscovery.DiscoveryHandler(Sensor_RemoveSink);
        Sensor_disco.Start();
    }

    private delegate void ActuatorAddDelegate(UPnPDevice d);
    private static void ActuatorAddFunc(UPnPDevice d)
    {
        StaticObject.map.Add_Actuator(d);

        String _location = StaticObject.map.Get_Location(d);
        Dictionary<String, object> WfArgs = new Dictionary<string, object>();
        WfArgs.Add("Location", _location);
        try
        {
            StaticObject.SLocalService.RaiseEvent(EventTypes.Discovery,
                new WorkflowEventArgs(StaticObject.DiscoveryInstanceId, WfArgs));
        }
        catch { }
    }

    private static void Actuator_AddSink(DimmableLightDiscovery sender, UPnPDevice d)
    {
        ActuatorAddDelegate dele = new ActuatorAddDelegate(ActuatorAddFunc);
        dele.BeginInvoke(d, null, null);

    }

    private static void Actuator_RemoveSink(DimmableLightDiscovery sender, UPnPDevice d)
    {
        StaticObject.map.Remove_Actuator(d);

        String _location = StaticObject.map.Get_Location(d);
        Guid SMInstanceID = StaticObject.Location_To_WorkflowID_Dict[_location];

        Dictionary<String, object> WfArgs = new Dictionary<string, object>();
        WfArgs.Add("Location", _location);

        StaticObject.SLocalService.RaiseEvent(EventTypes.Remove,
            new WorkflowEventArgs(SMInstanceID, WfArgs));
    }

    private delegate void SensorAddDelegate(UPnPDevice d);
    private static void SensorAddFunc(UPnPDevice d)
    {
        StaticObject.map.Add_Sensor(d);

        CpStandardizedLightSensor StandardizedLightSensor =
            new CpStandardizedLightSensor(d.GetServices(CpStandardizedLightSensor.SERVICE_NAME)[0]);
```

```
            StandardizedLightSensor._subscribe(300);
            StandardizedLightSensor.OnStateVariable_result += new
                CpStandardizedLightSensor.StateVariableModifiedHandler_result(StandardizedLightSensor_OnStat
                eVariable_result);

            StaticObject.CpStand_To_UPnPDevice_Dict.Add(StandardizedLightSensor, d);

        }

        private static void Sensor_AddSink(StandardizedLightSensorDiscovery sender, UPnPDevice d)
        {
            SensorAddDelegate dele = new SensorAddDelegate(SensorAddFunc);
            dele.BeginInvoke(d, null, null);
        }

        private static void Sensor_RemoveSink(StandardizedLightSensorDiscovery sender, UPnPDevice d)
        {
            StaticObject.map.Remove_Sensor(d);
        }

        static public event EventHandler senseEvent;

        static void StandardizedLightSensor_OnStateVariable_result(CpStandardizedLightSensor sender,
                byte NewValue)
        {
            Dictionary<string, object> args = new Dictionary<string, object>();
            args.Add("SenseValue", NewValue);
            senseEvent(null, new WorkflowEventArgs(StaticObject.DiscoveryInstanceId, args));

            UPnPDevice d;
            String _location;
            Guid SMInstanceID;

            d = StaticObject.CpStand_To_UPnPDevice_Dict[sender];
            _location = StaticObject.map.Get_Location(d);
            if (StaticObject.Location_To_WorkflowID_Dict.ContainsKey(_location))
            {
                SMInstanceID = StaticObject.Location_To_WorkflowID_Dict[_location];
                Dictionary<String, object> WfArgs = new Dictionary<string, object>();
                WfArgs.Add("ML_ID", StaticObject.map.Get_ML_ID(d));
                WfArgs.Add("SensorValue", NewValue);
                StaticObject.SLocalService.RaiseEvent(EventTypes.Sense,
                        new WorkflowEventArgs(SMInstanceID, WfArgs));
            }
        }
}
```