DEVELOPMENT OF DISTRIBUTED AND EXTENSIBLE BUILDING AUTOMATION SOFTWARE SYSTEMS

by

Pham Van Vung

A Tesina submitted in partial fulfillment
of the requirements for the degree of

Maser Degree in Engineering Computer Systems

Politecnico di Milano

2010

Approved by _____
Chairperson of Supervisory Committee

_____

_____

_____

Program Authorized
to Offer Degree _____

Date _____

ABSTRACT

The current requirements for software systems, especially in Building Automation Industry, are first able to extend its functionality to adapt to customers' requirement changes, second accessible over the network, third able to scale well with the varieties of client application needs, and fourth able to scale well with varieties of client devices and their platforms. This writing presents a four step methodology for development of software systems which satisfy these requirements. The four steps include Step 1: Development of a core server with functionalities implemented as add-ins, Step 2: Development of a communication channel as Web Service, Step 3: Development of the client applications and client components, and Step 4: Development of a GUI Designer and Client devices' platforms' simulators.

These steps are realized after a real life experiment which develops a Building Automation System at Omniabit s.r.l. Milano, Italy. This experiment leads to a software system which has extensible functionalities implemented as add-ins. It is accessible over the network using Web Service as its communication endpoint. It scales well with the large varieties of customers' client applications' needs by providing customers a GUI Designer lets users (customers or developers) with little programming experience design the client applications visually. It also scales well with the large varieties of client devices and their platforms in building automation industry with simulators built for them.

This proposed development methodology is applicable not only in the building automation industry software development but in general will help to develop software systems which adapt well with customer requirements change overtime.

TABLE OF CONTENTS

Chapter 1 1
INTRODUCTION 1
Chapter 2 4
REQUIREMENT ANALYSIS AND PROPOSED SOLUTION 4
2.1 Requirement analysis 4
2.2 Proposed solution with technologies and development methodologies 6
Chapter 3 11
LITERATURE STUDIES 11
3.1 Add-Ins Pipeline Architectures 11
3.2 Windows Communication Foundation (WCF) 12
3.3 Windows Presentation Foundation (WPF) 15
3.4 Silverlight 15
3.5 CommandFusion 15
3.6 ProntoPhilips 18
3.7 PRISM - Composite Application 19
3.8 Terminologies definitions 21
Chapter 4 23
DEVELOPMENT METHODOLOGIES 23
4.1 Development of the core server component 23
4.2 Development of the communication channel (Web Service) component 27
4.3 Development of the client components 33
4.4 Development of the GUI Designer and Simulators' components 43
Chapter 5 48
IMPLEMENTATIONS 48
5.1 Implementation of core functionalities server 48
5.2 Implementation of client applications 51
5.3 Implementation of the GUI Designer 54
Chapter 6 58
RESULTS AND DISCUSSIONS 58
6.1 Results and discussions of the core server development 58
6.2 Results and discussions of the communication channel development 59
6.3 Results and discussions of the client development 61
6.4 Results and discussions of the GUI Designer and simulators development 63
Chapter 7 65

CONCLUSIONS 65
Chapter 8 68
FUTURE WORKS 68
BIBLIOGRAPHY 71
APPENDIX 72
A. Summary of ELK Functions 72
B. ProntoScript client applications development tutorial 73

LIST OF FIGURES

Number *Page*
Figure 1 Software Architecture to develop an extensible, and distributed software system 8
Figure 2 Pipeline Architecture 11
Figure 3 Concept of CommandFusion 17
Figure 4 Pronto TSU9800 Device 18
Figure 5 An example of a composite system 21
Figure 6 Core Server Architecture of the Building Automation System 25
Figure 7 Design Architecture of the Communication Channel 31
Figure 8 Varieties of Client Devices in Client Applications Development 34
Figure 9 Client Design using WPF and Silverlight Technologies 36
Figure 10 Architecture implementing ProntoScript Widgets. 42
Figure 11 Generic Design Architecture of GUI Designer 45
Figure 12 Application of GUI Designer Architecture in Building Automation System 46

ACKNOWLEDGMENTS

I would like to express deepest appreciation to my committee chair, Professor Mezzalira Lorenzo. Without his guidance and persistent help this writing would not have been possible.

I would like to thank my supervisor at Omniabit s.r.l., Milano, Italy, Mr. Manuel Elia, who guided me in eliciting the system requirements for software systems in building automation industry. A thank you to my colleague, Domenico Orsini, who gave me useful feedbacks and design guidance during the real life development of the Building Automation System at Omniabit s.r.l.

I thank Omniabit s.r.l. which allows me to have a real life experiment in developing software systems for building automation industry. This helps me to realize this development methodology.

Chapter 1

INTRODUCTION

Currently the developments of industrial and real life software systems are having a challenge which is the system requirements change over time. New business opportunities may come, new technologies may become available, or even ongoing customer feedback during the development cycle which affects the requirements of the software systems. Therefore, it is important to have the methodology to build the software systems so that it is flexible and can be easily modified or extended in the future even when it is already deployed into industry.

Development of such software systems requires an architecture that allows individual parts of the application to be independently developed and tested and that can be modified or updated later, without affecting the rest of the application even if it is already deployed in the industry.

Most software systems' developments need a large team of developers such as graphical user interface designers, system analysts, business logic coders, etc. The difficulty the developers face is to design the software systems in such the way that multiple developers or sub-teams can work effectively on different pieces of the system independently, still making sure that the artifacts coming out from each team are integrated and work together correctly.

This writing will propose a solution for developing distributed systems which have requirements change over time. It was realized after the development of a "Building Automation System" at Omniabit s.r.l. This "Building Automation System" is extensible, scalable and can be deployed not only in individual houses, but also in industries such as Government Offices (for Video Conferences, for instance) or Companies (for Camera Surveillance and lightning systems, for

instance). Sometimes, the word "home" is used instead of "building", it means the devices which the built software applications control are the "home automation" devices such as the Cameras, the TV, the Sound Systems, the ProntoPhilips, etc.

The outcome artifacts of the development of this "Building Automation System" include of a core server which directly controls the home automation devices, the client applications used directly by the clients (customers), and a GUI Designer which the users can use to design the client applications. In this case, "users" may be the clients/customers who use the GUI Designer to design their own client applications or the developers who develop client applications based on the clients/customers' requirements.

This writing starts with **Chapter 1** (this chapter) which is an introduction to the background problem and brief descriptions about the proposed solution. **Chapter 2** presents the requirement analysis of software systems in building automation industry and proposes a methodology to develop such the systems which satisfy the analyzed requirements. **Chapter 3** presents some materials and technologies in order for readers to understand and be able to experience their own development. **Chapter 4** explains the methodologies, reasoning about choices of technologies being used, and how to apply the technologies and the development architectures in development of the software systems. This chapter can be used as guidance to readers to experience by their own the development of extensible, distributed software systems, especially for building automation industry. **Chapter 5** presents some real implementations of some special parts in development of the Building Automation System. Readers who are not interested in detail coding implementation could skip this chapter. **Chapter 6** presents the results of the development methodologies and some further discussions about noticing points. **Chapter 7** presents conclusions of this writing. **Chapter 8**

presents the future works or the directions which should be applied in this development methodology.

Chapter 2

REQUIREMENT ANALYSIS AND PROPOSED SOLUTION

2.1 Requirement analysis

This part will present the requirement analysis of software systems in general and in building automation industry specifically. The current urging challenges in developments of software systems are the functionalities change over time, the accessibility over the network, the varieties of client devices and their operating platforms. To deal with these challenges the to-be-built systems should have following requirements: functionality extensibility, distributed over the network, supporting varieties in types of client applications (desktop based, web based, mobile phone based systems, and other types of client applications for embedded systems), and supporting multiple platforms (Windows, Mac OS, iPhone, Palm Os, and devices with embedded operating systems, etc). These requirements could be described as follows.

The first requirement is the **Extensibility** of the software systems; the critical requirement of current software systems is the ability to extend its functionalities dynamically without having to recompile the code of the systems. I.e., the systems must face challenges such as discovering, activating and managing the lifetime of the functionalities. Therefore, there is a requirement for an infrastructure that is designed to include hooks and mechanisms for expanding/enhancing the system with new capabilities without having to make major changes to the system infrastructure, even if it has been deployed in the industry.

The second requirement of the current software systems is the requirement to be **Distributed Systems**. Many software systems require the capability to be able to communicate with each other over the network. For example, the user of the

Building Automation System may be at work and would like to connect to his/her security system at home via the Internet. The use of a distributed system is beneficial for many further reasons. For instance, it may be more cost-efficient to obtain the desired level of performance by using a cluster of several low-end computers, in comparison with a single high-end computer [2]. A distributed system can be more reliable than a non-distributed system, as there is no single point of failure. Moreover, a distributed system may be easier to expand and manage than a monolithic uni-processor system [3]. There are lots more advantages including the ability to connect remote users with remote resources in an open and scalable way. The system can be easily altered to accommodate changes in the number of users, resources and computing entities.

The third requirement is to support the **varieties in platforms** of operating systems and based programming language running the software systems. In many software systems, there is a variety of clients such as Desktop based clients, Web based clients, Mobile Phone based clients, etc. These clients' interfaces should be unified as closely as possible. For instance, in case of a Building Automation System, the user may have a desktop installed in their house to control the system, s/he may also have an iPhone and would like to run the client application on it, s/he may also like to access to the security system at her house from the internet when she is traveling, etc. And the clients' interfaces should look alike for the convenience of the users.

The fourth requirement is to support **varieties in types of client devices** with different capacities such as different processing powers and memory capacities. In this era of pervasive computing, there are many devices with their own operating systems in the building automation industry. Such as computers with Microsoft Windows, Mac OS, or Linux, mobile phones with Windows Mobile, and other devices with their embedded OS such as ProntoPhilips. Therefore, the

next requirement is the software systems must be able to accommodate such varieties in the client platforms.

The fifth requirement is to support the **varieties in client interfaces** combinations and functionalities. Different clients may need different combination of the functionalities of the software system. Therefore, the system must accommodate a combination of functionalities which meets the clients' needs. For instance, one user of the Building Automation System may need an application to control a security system and a heater; another user may need a different client application to control the lightning system and a video conference system, not the heater or security system, and so on. There should be an easy way, with less programming requirements, which lets the users easily create their own clients' applications as they need.

Besides these mentioned requirements, there are, of course, many more quality-of-service requirements such as security, ease of use, reliability, and so on.

2.2 Proposed solution with technologies and development methodologies

This part will present the development architectures and technologies, which were studied at OMNIABIT, Milan, Italy. It is used to solve the mentioned problems in developing software systems, especially for building automation industry. I.e., this development methodology has been built after the real life development of Building Automation System in OMNIABIT. The architectures and technologies are depicted as in Figure 1 and could be briefly described as it is divided into four main parts. Part I is the core server implementing main functionalities of the system. Part II is the Web Service enabling the distributable and multiple platforms of the clients (Web based, Desktop based, Mobile Phone based client applications) communicating with the core sever over the network. Part III the client applications/components are developed to be communication

interfaces between users and the core server. Part IV is the GUI Designer used to design the client applications using client components developed in Part III, this enables the reusability and ease of client application developments. These parts can be briefly described as follows.

Part I: The core server

This part the technology and architecture used should enable the functionalities of the software systems to be extensible. Meaning the new functionalities can be added to the system in the future without any major modification or having to shutdown the currently deployed systems. The technological solution provided in this writing is the *System.AddIn* [5], this enables building extensible application that dynamically loads add-ins into it. It helps in solving challenges such as discovering, activating, and managing the lifetime of the add-ins. For instance, the Building Automation Software may first have add-ins built to control Lightning (turn on, turn off), Camera Surveillance, etc. At a later time if a customer requests further functionalities to control the other devices (Heater, or Television for instance) the developers only need to develop the add-ins for these and copy them into the add-in folder, the add-ins (functionalities) will be enabled dynamically.

Part II: The Communication Channel

This part the technology and architecture used should enable the communication between the client applications and the core servers of the software systems over network. The technology used is the WCF (Windows Communication Foundation) which enables building service-oriented applications that communicate across the web and the enterprises. The client applications then are extensible and decoupled from the server core. We could build Windows based client applications, Web based client applications, Mobile based client

applications as needed and they all could communicate with the single server core via WCF. Therefore, the client applications can be distributed in the network.

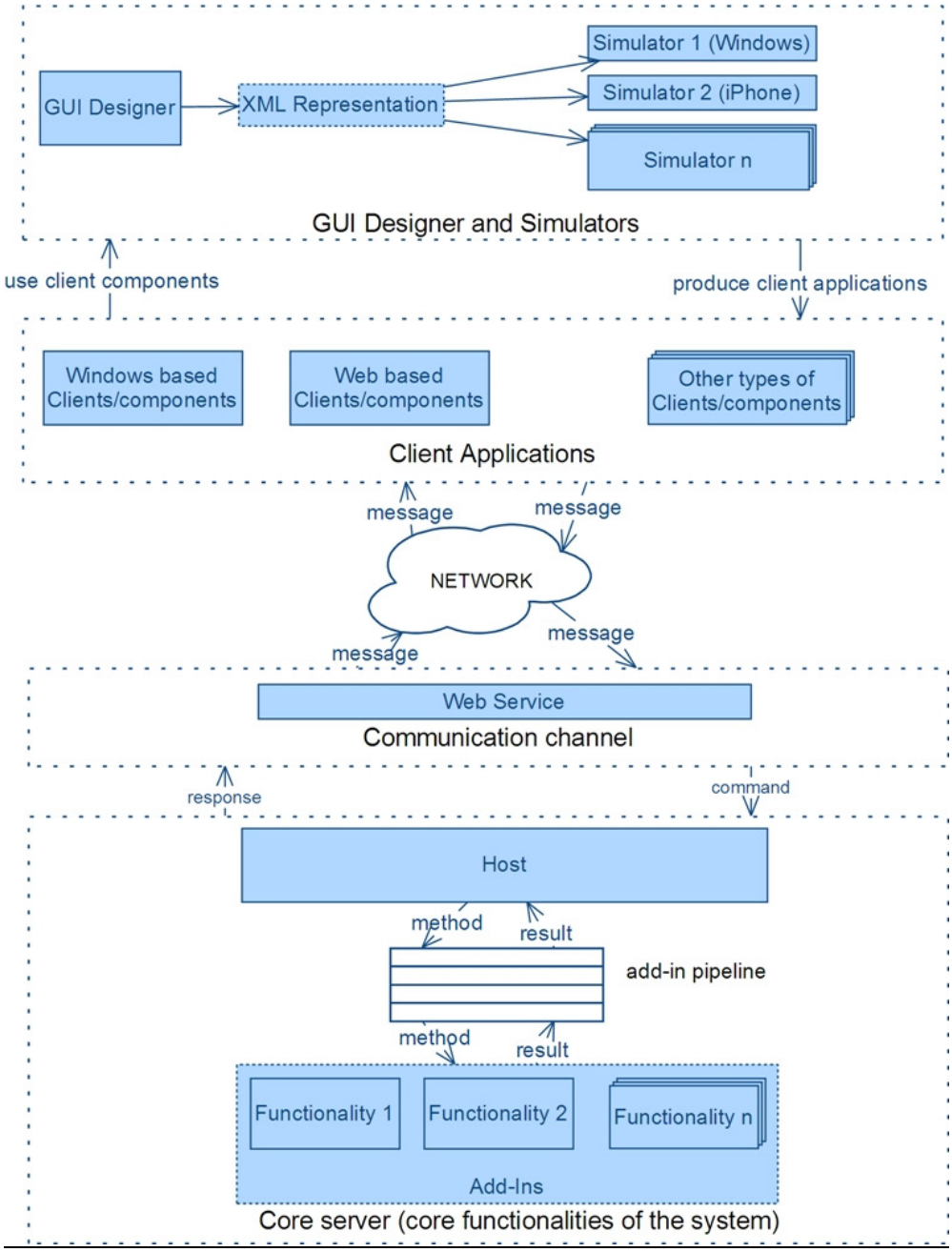


Figure 1 Software Architecture to develop an extensible, and distributed software system

Part III: The client applications/components

This part, the technology and architecture used should enable the ease of development of client applications which is able to communicate with the communication channel over the network, and in many cases should be asynchronous or duplex communication. The technologies used in this part are the WPF (Windows Presentation Foundation) to build the Windows based clients applications/components and the Silverlight to build Web based client applications/components, and some further technologies for specific devices such as CommandFusion for iPhone/iPod/iPad and ProntoScript for ProntoPhilips, and so on. The client components are reused in the GUI Designer developed in Part IV.

Part IV. The client GUI Designer and client simulators

This GUI Designer is required due to the variety of client applications for different customers. One customer may have interests in composition of some client components (developed in part III), the others may have interests in different composition of some other client components. Furthermore, different client may have different platforms. This designer, first, makes use of the basic client components developed in part III, it enables the user to visually drag/drop and wire these components together to meet his/her needs of what client applications should be. This is a GUI Designer, meaning the components are dragged, dropped and wired visually, with less user-programming-experience requirement. This allows the easy and dynamic compositions of client components for clients' needs. Furthermore, the output comes out from this GUI Designer is described in XML. One simulator is built for each platform (one for Mac OS, one for Windows, one for iPhone, etc). These simulators are able to read the XML descriptions of the applications created by GUI Designer in their

designated platforms and realize the real client applications. This solves the problem of having to build one project for each platform.

At the implementation point of view, the implementation pattern used in all the parts of software development is the PRISM developed by the Microsoft patterns & practices group (http://compositewpf.codeplex.com/). This helps design and build flexible client applications using loosely coupled, independently evolvable pieces that work together and are integrated into the overall application. This type of application is known as a composite application. Furthermore, it has built in infrastructure which is used as standard backbone of the application. Enabling developers developing different parts in parallel, integrating together, understanding each other code due to using the same "language" of communication that developed by the PRISM.

As a summary, using these technologies together with the architecture and implementation patterns depicted in Figure 1. The development of the software system is flexible and parallel, the developments of the components are decoupled. Interface, communication, and business logic are well defined and separated. They are still unified and developers easily understand each other using the "language" of PRISM. The software is extensible to meet the flexible changes in functionalities of enterprises. It scales well for many platforms like Windows, Mobile, etc and Interface requirements (by using GUI Designer and Simulator), and is accessible (distributed) over the network (with the Web Service).

Chapter 3

LITERATURE STUDIES

This chapter will present the necessary materials about the technologies used in order to understand and be able to apply the architectures in developing extensible, distributed software systems.

3.1 Add-Ins Pipeline Architectures

This part will present the first technology used in the architecture for the extensibility of the functionalities of the software systems, the *System.AddIn namespace in C#*. The architecture for Managed Add-Ins is briefly described in [5] which could be depicted as Figure 2.

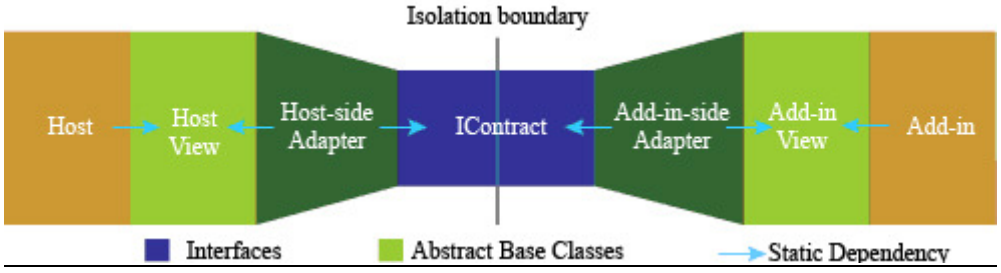


Figure 2 Pipeline Architecture

An Add-In Communication Pipeline defines a series of abstractions that allow hosts and add-ins to version independently and be completely unaware of how the other side is built or how information is passed across isolation boundaries. The host and add-in only ever depend on one component (the view) which itself has no dependencies elsewhere on the system. This means the host and add-in are completely unaware of each other and the contracts that are used to cross the

boundary. Contract is the only thing that the two sides have in common. The pipeline components are Views, Contract, and Adapters.

The Views' components include the host and the add-in views. They are assemblies that contain nothing but a series of abstract classes that represent host and add-ins view of each other. These views assemblies take no dependencies on other components in the system. This is the object model and the public surface area a host would expose to its add-ins or vice-versa.

Next component is the Contract. Its assembly contains the non-versioning types (contracts) that are loaded in both sides of the isolation boundary and define the communication protocol over that boundary. These assemblies also take no dependencies on other components. Neither the host nor the add-in ever program directly against the contract and thus they often sacrifice a nice programmable surface area in favor of a more efficient/stable protocol.

Another component is the Adapters. Its role is to convert to and from the views and the contracts. It is in these adapters that you would put any code to adapt between different versions of views and contracts. For detail documentation about this technology you can read article ".NET Application Extensibility, Part 2" [6].

3.2 Windows Communication Foundation (WCF)

Next technology is the Windows Communication Foundation (WCF) that is used to publish the functionalities built in previous step as network service endpoints. WCF is described in details in article [7]. It is a framework for building service-oriented applications. Using WCF, we can send data as asynchronous messages from one network service endpoint to another. The network communication

channel could be duplex communication for some types of client applications which enable the server pushing data to the client applications as it is available.

A network service endpoint can be part of a continuously available service. A few sample scenarios include: A secure service to process business transactions, a service that supplies current data to others, such as traffic report or other monitoring service, a Silverlight application to poll a service for latest data feeds, etc. While creating such applications was possible prior to the existence of WCF, WCF makes the development of endpoints easier than ever. In other words, WCF is designed to offer manageable approach to creating Web services and web service clients.

The WCF includes the following set of features. For more information see "WCF Feature Details" [8].

- Service Orientation, WCF enables creating service oriented applications. Service-oriented architecture (SOA) is the reliance on Web services to send and receive data. SOA is described in [9]. The services have the general advantage of being loosely-coupled instead of hard-coded from one application to another.

- Interoperability, it implements industry standards for Web service interoperability.

- Multiple Message Patterns, messages are exchanged in one of several patterns. The most common pattern is the request/reply pattern, where one endpoint requests data from second endpoint. The second endpoint replies. There are other patterns such as a one-way message. A more complex pattern is the duplex exchange pattern where two endpoints establish a connection and send data back and forth.

- Service Metadata, WCF supports publishing service metadata using formats specified in industry standards. This metadata can be used to automatically generate and configure client for accessing WCF services.

- Data Contracts, the easiest way to handle data is by creating classes that represent a data entity with properties that belong to the data entity. Once you have created the classes that represent data, your service automatically generates the metadata that allows clients to comply with the data types you have designed.

- Security feature, messages can be encrypted to protect privacy and you can require users to authenticate themselves before being allowed to receive messages.

- Multiple Transports and Encodings, messages can be sent on any of several built-in transport protocols and encodings. These messages can be encoded as text or using an optimized binary format.

- Reliable and Queued Messages, WCF supports reliable message exchange using reliable sessions.

- Durable Messages, it is one that is never lost due to a disruption in the communication. The messages in a durable message pattern are always saved to a database. If a disruption occurs, the database allows you to resume message exchange when the connection is restored.

- Other features such as Transactions, and Extensibility, WCF architecture has a number of extensibility points. If extra capability is required, there are a number of entry points that allow you to customize behavior of a service.

3.3 Windows Presentation Foundation (WPF)

Next technology is the Windows Presentation Foundation (WPF) which is used to design the Desktop based client applications which are able to communicate with the WCF Web Services established in previous step. WPF is described in [10]. WPF is the next-generation presentation system for building Windows client applications with visual user experiences. The core of WPF is a resolution-independent and vector-based rendering engine that is built to take advantage of modern graphics hardware. WPF extends the core with a comprehensive set of application-development features that include Extensible Application Markup Language (XAML), controls, data binding, layout, etc.

3.4 Silverlight

The details about Silverlight are described in "Top Silverlight Features" [12]. Silverlight is a powerful development platform for creating engaging, interactive user experiences for Web, desktop, and mobile applications. Silverlight is built on Microsoft's industrial-strength application development tools and a platform that promotes stability, scalability, reliability, and performance. It works through all major browsers on Mac, Windows, and Linux client operating systems, mobile devices such as Windows Phone 7, Nokia Series 60 and set top boxes. Silverlight extends browser experiences to the desktop and devices with innovative tools, servers and frameworks. Silverlight creates rich Web-based applications that quickly integrate with your existing back-end systems. It enhances existing Web and SharePoint sites by incrementally adding Silverlight components.

3.5 CommandFusion

Currently the iPhone/iPod/iPad devices are the preferable client devices on which the customers would like to be able to run Building Automation Client

Applications. Coding client applications for these devices may need developers to learn to code Objective C in Mac OS or so. This would lead to some development time and cost. However, CommandFusion (http://www.commandfusion.com) helps the windows developers develop Building Automation Applications in Windows OS. The concept behind CommandFusion is its communication protocol [21].

The first important point of this concept is its message format. All messages in the CommandFusion protocol take the form of *[identifier]=[value][h03]*. Each message is delimited with the End of Message (EOM) of hex value of *03*. Another point is the way it divides and defines the Identifier types. The following identifiers are valid within the protocol:

- Digital Joins - d#
- Analog Joins - a#
- Serial Joins - s#
- Password Validation - p
- Initialize - i
- Orientation Mode Change – m
- Heartbeat Message – h
- Lists – l# (lowercase L)

A join is a type of signal which is sent or received among building automation devices. Digital, Analog and Serial join identifiers must be combined with a join

number (starting at 1). For example *d12=0h03*, where the join number is "12", the value of the signal is "0".

The concept of CommandFusion is depicted in Figure 3, it provides users a GUI Designer which has the primitive controls such as a button control, label control, gauging control, text control, and so on, which can be bound to digital join, analog join, or serial join. This GUI Designer lets user visually design the client applications for iPhone/iPod/iPad client devices in Windows Operating System environment.

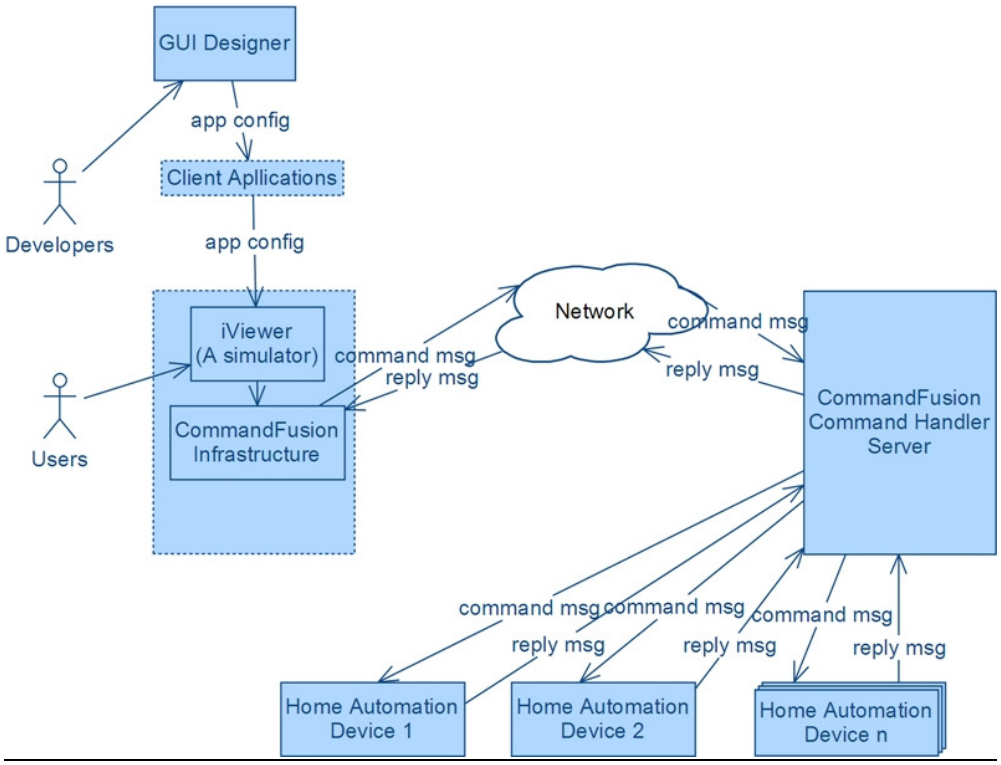


Figure 3 Concept of CommandFusion

There is an infrastructure which enables behaviors of these controls. For instance, a button is bound to a digital join number *12* with the value *1*. When user clicks the button the infrastructure will automatically create a command "*d12 =1\x03*"

and send to the CommandFusion command handler server over the network. In the other way, when the server sends to a client a command "*d12=0\x03*" the infrastructure will parse and see the button bound to digital join number 12, has value 0 (indicating a button is is in an inactive state for instance) ,and therefore, set the button in the inactive state. This GUI Designer, the communication protocol, and the infrastructure enable developers to develop the client applications for building automation systems in Windows OS. In the iPhone/iPod/iPad devices, there is an *iViewer* acts as a simulator which will load the file created from GUI Designer and realize client applications. For further details about CommandFusion please refer to [21].

3.6 ProntoPhilips

In Building Automation Industry, one client device which is often being used is the ProntoPhilips (http://www.pronto.philips.com). Figure 4 is the Pronto TSU9800 which is being used at Omniabit s.r.l. as a client device.

Figure 4 Pronto TSU9800 Device

The ProntoPhilips is a device which is able to let developers develop and execute client applications with ProntoScript programming language [22]. ProntoScript allows one to add flexible 2-way communication and dynamic UI's to the Pronto system, bringing an even higher level of home automation sophistication. It is based on JavaScript, a popular and proven scripting language. Integrated into ProntoEdit Professional, it unlocks the full power of the WiFi-enabled Prontos and Extenders:

- JavaScript is a modern, very high level programming language, allowing rapid development of rich end user applications

- The web offers plenty of references and solutions to general programming challenges in JavaScript, more than any other language.

- Encapsulated into a single Pronto Activity (Device), that can be merged into projects; the complexity of the code can be shielded completely from the custom installer. Users just want to plug in a 2-way module for controlling his selected equipment. A few standardized hidden pages with instructions and parameters allow users to configure the module to operate seamlessly within his specific system.

3.7 PRISM - Composite Application

The complete description of PRISM is described in "Composite Application Guidance for WPF and Silverlight" [13]. It is a set of guidance designed to help developers more easily manage the complexities they may face when building enterprise-level Windows Presentation Foundation (WPF) client applications and Rich Internet Applications with Silverlight. This will help developers design and build flexible client applications using loosely coupled, independently evolvable pieces that work together and are integrated into the overall application. The

approach it use to solve these challenges is to partition the application into a number of discrete, loosely coupled, semi-independent components that can then be easily integrated together into an application to form a complete task. Composite applications provide many benefits, including the follows:

- They allow modules to be individually developed, tested, and deployed by different individuals or teams. The modules can be modified or extended with new functionality more easily, thereby allowing the application to be more easily extended and maintained.

- They provide a common main interface composed of interface components contributed from various modules that interact in a loosely coupled way. This reduces the coupling that arises from multiple developers adding new functionality to the interface, enabling a common appearance.

- They promote re-use and a clean separation of concerns between the application's capabilities, such as logging and authentication, and business functionality that is specific to applications.

- They help maintain a separation of roles by allowing different individuals or sub-teams to focus on a specific task or piece of functionality according to their focus or expertise. It provides a cleaner separation between the user interface and the business logic of the application—the interface designer can focus on creating a richer user experience.

Composite applications are highly suited to a range of client application scenarios. For example, a composite application is ideal for creating a rich end-user experience over a number of disparate back-end systems. Figure 5 shows an example of this type of a composite application.

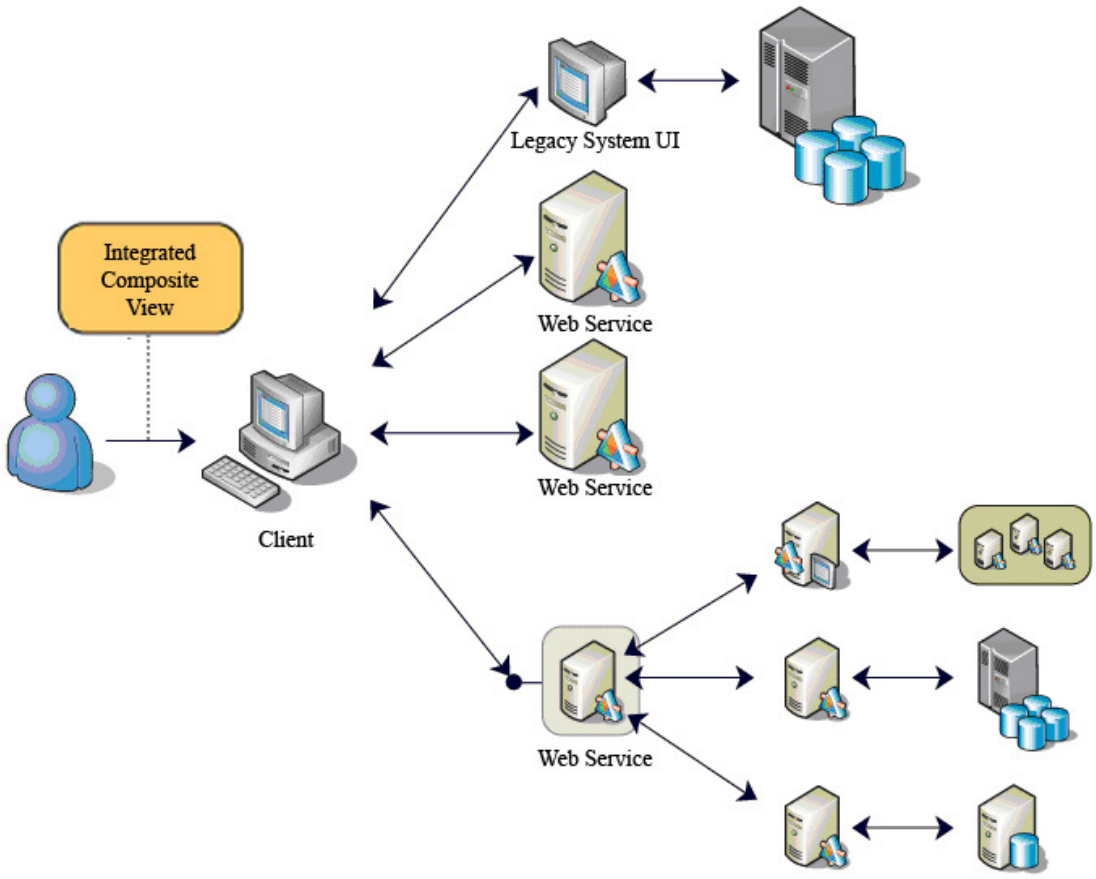


Figure 5 An example of a composite system

3.8 Terminologies definitions

This part summarizes again some terminologies that frequently used in this writing together with their intended meanings.

WCF (Widows Communication Foundation). It is a framework for building service-oriented applications.

WPF (Windows Presentation Foundation). It is a graphical subsystem for rendering user interfaces in Windows-based applications.

Silverlight. It is a powerful development platform for creating engaging, interactive user experiences for Web, desktop, and mobile applications when online or offline.

Prism. It is formerly known as Composite Application Guidance for WPF and Silverlight, designed to help you more easily build modular Windows Presentation Foundation (WPF) and Silverlight client applications

CommandFusion. It is a software vendor who provides a GUI Designer to let developers develop home automation software in Windows Operating Systems.

ProntoPhilips. It is a new class of programmable touchscreen remote controls for intuitive operation.

Extensibility. It is the ability to enhance, extend, or replace pieces of the software system without requiring you to redesign the system.

Add-in (also called Add-On, Extension, Plug-In, Snap-In). It is a component you can add to a system to increase its capabilities.

Host. It is an application that supports extensibility.

Pipeline. It is the machinery that enables the host and add-in to communicate over a version-resilient, secure protocol.

GUI Designer. It is software which lets the users visually design their client applications.

Simulator. It is software which runs in client devices and realizes the client applications based on XML description of the client applications.

Chapter 4

DEVELOPMENT METHODOLOGIES

This chapter will explain the methodologies in developing the extensible, distributed Building Automation Systems. It will explain the steps of applying the technologies and the architectures in developing the Building Automation System at OMNIABIT, Milan, Italy. It will then show in each part of the architecture, which technologies were applied with rational explanations of the choices of the technologies together with how to apply them in solving the software system development challenges. This chapter can be used as guidance to readers who would like to experience the development of extensible and distributed software systems, especially for building automation industry using the development methodology proposed in this writing.

The development steps will be divided as the architecture depicted in Figure 1. We start with development of the core server component, then the development of the communication channel, the development of the client applications/components, and the development of the GUI Designer and the simulators.

4.1 Development of the core server component

This part will first present the challenge in developing the core server (core functionalities of the to-be-built system) of the software systems. Then it will present the reasoning about technologies used in the development of the core server together with the steps of how to apply the technologies in developing it for the Building Automation System as a real life experiment of the architecture.

The challenge in this part that we need to overcome is the ability to extend the functionalities of the software systems. The functionalities may need to be added or modified over time due to business requirements change, or due to new devices, new technologies come to life. For instance, in case of the Building Automation System, the developers can make the modules to control, let's say, a Lighting system, a Camera surveillance system, a Security system, a Heater, or some other popular devices that the developers can think of. However, in this era of pervasive computing, there are many devices in Building Automation Industry that the developers can't think of and those may be needed to be controlled in the future.

In addition, the potential customers of the Bilding Automation System are unknown. They may have different needs in controlling devices. For instance, some may only need to control a lighting system and a heater system. Other may only need to control the camera surveillance system and not the others. Even the developers could make a system which is able to control all the types of devices (in many cases, developers cannot) but it may not be efficient because of the differences in the needs of customers. Therefore, the functionality should be modularized and could be added, removed, or modified easily without affecting the other functionalities.

The solution architecture of this core server is depicted in Figure 6. One functionality (one device to be controlled, for instance) will be implemented by one add-in. The developers will first develop a set of specific add-ins in order to control the popular devices in the current markets such as a lighting control add-in, a heater control add-in, a camera surveillance control add-in, etc. Once a client comes with a new device or new requirements, the add-ins will be developed and put into the "add-ins" folder and the functionalities for the requirements will be enabled in the system.

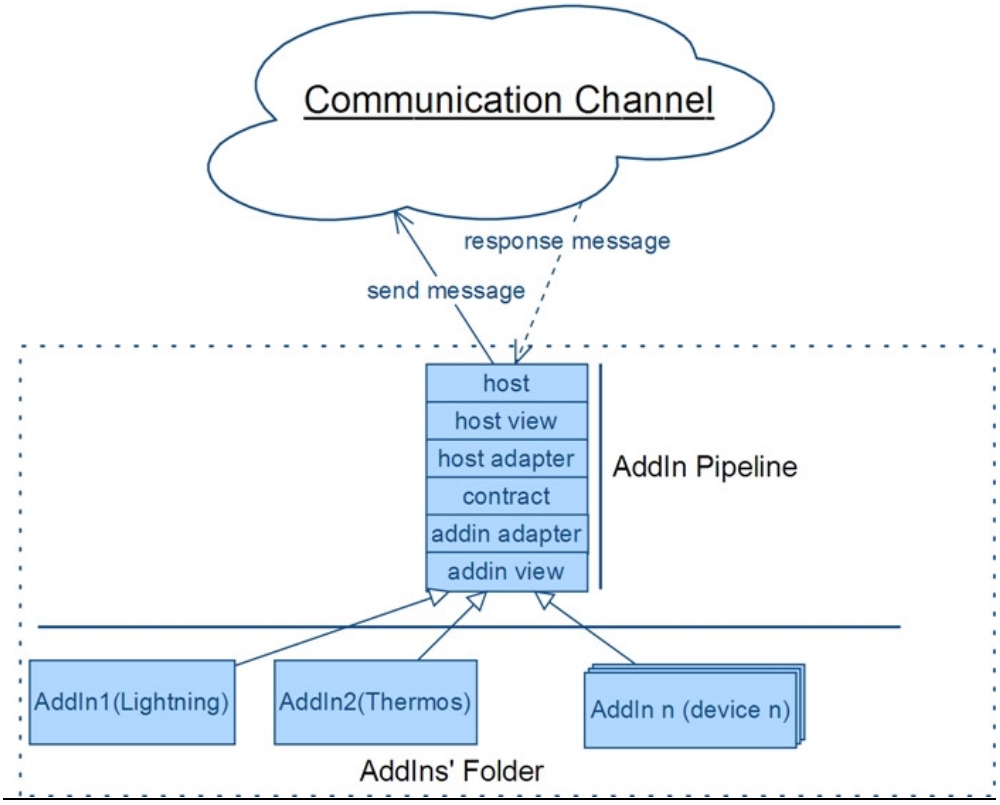


Figure 6 Core Server Architecture of the Building Automation System

In the implementation point of view, it uses the *System.AddIn framework in C#*. This framework solves the problem well. It automatically loads, activates, and manages life time of the add-ins. In another case, when the customer wants to modify the behavior of a controlling system the add-in is easily identified, isolated, and modified without affecting the other add-ins. Even, when a client would like to remove a device control when it is not needed the add-in is just simply removed from the "add-ins" folder, to increase the performance for instance.

Another reason for choosing the *C# System.AddIn framework* to implement this server core is that at the developers' point of view, there are several tools and

samples [14] which make the development a lot easier. The noticing tool that we have used in implementing the add-in feature is the Pipeline builder tool (which can be downloaded from http://clraddins.codeplex.com). Using this tool, the developers only need to implement some relatively simple programming interfaces (the add-in view) and putting some annotation attributes (saying this module is an add-in, and specifying add-in name, etc.,) in order for the *System.AddIn* framework know the module is an add-in. It then will automatically load, activate, and manage the life time of the add-in without developers having to know how to code the tasks themselves.

With this tool, the code for enabling the add-in behavior (discovering, loading, and activating) is done for developers and is managed code. In other words, Pipeline builder is designed to offer manageable approach to creating expandable applications. Developers only need to concentrate on their own business logic which is how to control a device. Moreover, the developers can be divided into teams which develop different add-ins in parallel at a time and at the later time can still work together seamlessly.

The more difficult part in the real implementation of this architecture is the design of the *contract* (explained in Chapter 3). The *contract* is a programming interface that both the *add-in* and the *host* must implement in order for the host to be able to activate the *add-in*. And the *contract* must be a stable programming interface which means its set of members and methods must be fixed. However, as mentioned, there are many types of devices with many different functionalities leading to many different possible functions in each add-in and many are unknown in advance. The *contract* must be as generic as possible and should be in a form which is able to express all the possible functions in add-ins. Therefore, the contract will expose one generic method called "*Execute*". The method is of the form "*Execute (int id, string methodName, Object [] arguments)*". The *id* is the id of

the add-in module, the *methodName* is the name of the method which is being called, and the *arguments* is a generic array which contains the set of arguments passing as the parameters of the calling *methodName*.

The *host* is in charged of parsing the *Execute* method arguments to know which add-in is being called to load/activate the corresponding add-in (if it's not already done) and call to the corresponding method passing the *arguments*. Having a generic method to express all the methods is a very important design decision since it solves all the problems of the unknown number of functions in unknown add-ins. Furthermore, the programming interface in communication channel also needs to have a stable interface to be published to the client applications. This will be further explained in next part when we develop the communication channel of the system.

As a short summary, the development the core server (implementing core functionalities) of the software applications for building automation industry has many unknown information challenge. The add-in architectures together with its tools and samples help the development of the extensible core server possible. Furthermore, the design decision of making a generic method "*Execute(int id, string methodName, Object[] arguments)*" as a *contract* between the *host* module and all the add-ins makes it possible for the *host* to accommodate all types of methods in different add-ins. In addition, this design decision also helps in the next step when we develop the communication channel which needs a stable and generic programming interface to be exposed to the client applications over the network.

4.2 Development of the communication channel (Web Service) component

After having the core server which implements the functionalities of the Building Automation System as extensible to scale for the varieties of customers' needs, the next challenge is to make the system (its functionalities) as easily accessible

and extensible over the network as possible i.e., to scale over the network. This means to be able to expose the server core functionalities over the network in an end point with a stable view (programming interface) which the client applications are able to communicate with via some network protocols (TCP, HTTP, and so on). The view (programming interface) given to the end-users should be a unified view (stable programming interface) and the communication over the network should be secured and trustable.

The client programs need no more to discover where to find the server for the relevant functionality. The client programs need no more to interact with each functionality server individually (if they are located in several servers). This part will explain about why we should use WCF to implement the communication channel among the core server and the client applications. Then it will present the detail implementation architectures of the WCF Web Service of the Building Automation System. Specifically, the **Communication Manager** will be emphasized. This part also mentions one important aspect of any communication service, which is the Security technologies applied in order to secure the communication between client applications and the WCF Web Service.

There are several ways to enable a application to be accessible over the internet such as Middleware, Enterprise Application Integration (EAI) [15, 16, 17 and 18] and Web Services. Middleware provides the programmer with functionalities which, otherwise, should be built anew each time needed. It is a large software infrastructure required in order to create these programming abstractions such as RPC (Remote Procedure Call) based systems, TP Monitors (Transaction Processing Monitors), Object Brokers, Message Based Systems (asynchronous). Enterprise Application Integration (EAI) extends the middleware concept from new application logics creation to complex application integration. It mainly uses

asynchronous communications such as message-broker, publish/subscribe paradigm, workflow management systems, etc.

The Web service could be a procedure, a method or an object provided with a stable and public interface which can be invoked by clients. It is a way to expose functionalities of an information system making them available through standard web technologies. It is a software application identified by a URI (Uniform Resource Identifier). Its interfaces and connections can be defined, described, and discovered by means of XML components. It supports direct interactions with other agents by means of XML messages exchanged via internet protocols. The web services are used as sophisticated wrapper in a tier above conventional middleware services. Web Service is the reincarnation of traditional EAI solutions. In this implementation of Building Automation System we are using WCF (Window Communication Foundation) to build web service as its communication channel.

WCF is a framework which is compatible with the technologies used in developing core server functionalities (*System.AddIn in the C# 3.5*). It is a part of the Microsoft *.NET Framework* that provides a unified programming model for rapidly building service-oriented applications which communicate across the web and the enterprises. The WCF Web Services also provides set of ready to use development tools, embedded with Visual Studio 2008, which ease the development of the Web Services in software systems. In other words, WCF is designed to offer manageable approach to creating Web services and web service clients.

Communication between the WCF Web Service and its client applications is via XML messages. The transportation protocol could be HTTP, TCP, and others. The communication protocol via HTTP is very common for computer based client applications. Furthermore, for many other types of client devices such as

iPhone/iPod/iPad, or the Pronto Phillips, Camera, etc., are equipped with processors which are able to communicate with other devices via TCP. Therefore, this solution for building the communication channel using WCF Web Service would satisfy the needs for network communication for all types of client devices.

The biggest challenge in making the WCF Web Service as flexible yet stable to be published to the client applications is to be able to have one generic interface which can expose all the possible methods coming from all add-ins for to-be-controlled devices. However, in our add-in architecture the methods are different and changing over time. For instance, one add-in, say for lightning, may have the "*dim_up*", "*dim_down*" methods, another add-in, say for Sound System, may have "*volume_up*", "*volume_down*" methods. Even if the method names are foreseen, it will be a big challenge to expose all the methods for all the add-ins as programming interface for this communication endpoint (web service). In fact, the developers will never know the unforeseen method names for unforeseen add-ins and unforeseen devices from unforeseen customers. These many levels of unforeseen information that make the WCF Web Service seem to be unreasonable to provide a stable interface to the publics since the methods are not stable.

As mentioned in the step developing the core server of the system (**Part 4.1**), the solution to this problem is that instead of exposing all the methods, the WCF Web Service exposes only one generic method which is *Execute*. The *Execute* method is of the form *Execute(int id, string methodName, Object[] arguments)*. The first parameter is the *id* which is the id of the add-in in which the client application would like to execute the method. The method in the add-in is specified by the next parameter as the string, the *methodName*. The third parameter is also another "trick" which helps to solve another problem of the method varieties in number

of arguments. One method may have one parameter, another may have two, some may not have any, etc. Therefore, a generic array of generic *Object* type could express all of these problems. I.e., if the method has only one parameter, the array should only have one element, if it has two the array should have two, and so on. Another noticing point is the *Object* generic type, the **Communication Manager** will just pass the arguments to the method, the method will itself know what types of parameters it is expecting and cast the generic *Object* type to the types it is expecting.

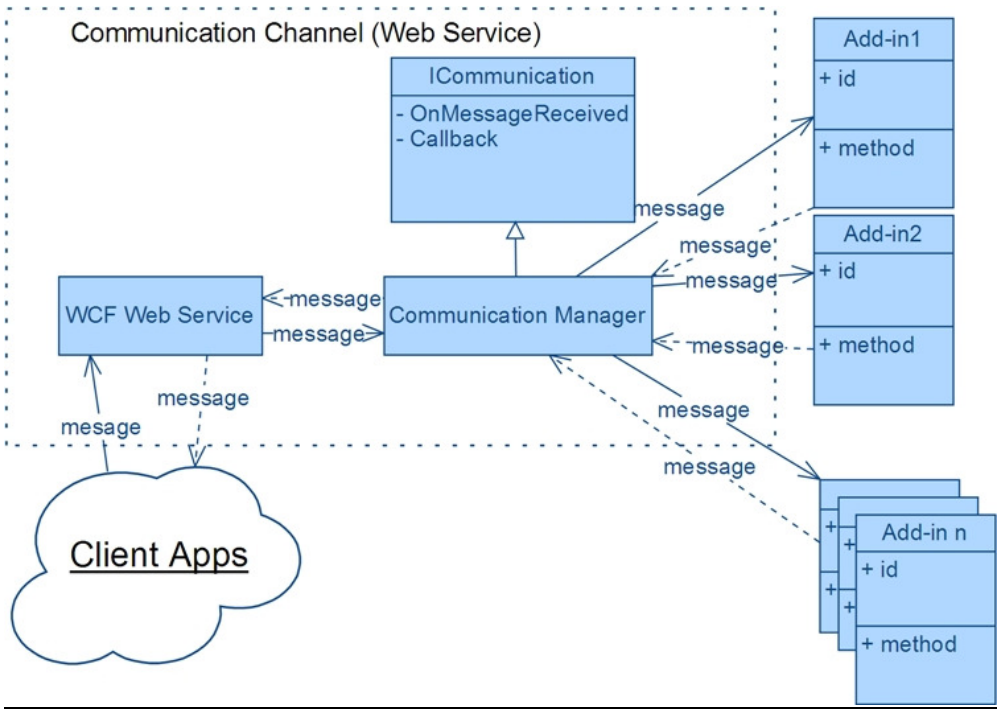


Figure 7 Design Architecture of the Communication Channel

Figure 7 depicts the design architecture of the WCF Web Service. The main component of this architecture is the **Communication Manager** which has the responsibility to direct the command message coming from client applications to corresponding add-in based on the *"id"* in its message. Furthermore, it has also

another important method which is a callback method, for the add-in to be called when there is some event happened in the add-in. This callback method then will forward the data sent by client devices to the WCF and finally to the client applications who subscribed for the event.

The WCF interface is simply exposing the methods of this **Communication Manager** which is stable and satisfies the needs from all client applications and be able to execute all the unforeseen methods from unforeseen add-ins. And therefore, the interface published to the public is stable.

Another important aspect in developing software systems which are distributed over the Network is about the Security of the communication channel. The security used in this communication channel between the WCF Web Service and the client applications is made at two levels: the authentication and the authenticity. The authentication is implemented by the user credential information such as the username and password. The Authenticity is made by using the Certifications. Since the service is published over the network, there may be many unauthorized devices who would like to access the network. The certification will only authorize the devices installed the client applications which have the certificates installed to be able to access to the communication channel.

Many client devices for Building Automation Systems are supporting Certificate installation such as the Computers (of course), PDA(s), Smart Phone(s), etc. For some devices which do not support certification, the simpler version of security will be applied such as the username, password simple authentication method.

As a short summary, the development of the Communication channel part using WCF together with the implementation design architecture depicted in Figure 7 allows the to-be-built system to expose all the possible methods developed in all possible device add-ins a stable public programming interface to client

applications over the network. This communication channel has two main methods. The first one is the generic method called "*Execute"* which processes commands coming from the client applications. The second one is the *Callback* method which is used to receive messages coming from add-ins and forward to the client applications. For security (authentication and authenticity) of the communication channel, the certificates and user credentials (username and password) are used.

4.3 Development of the client components

After having the Web Service developed with the external architecture which eases the client application accessibility to the servers with possibility to discover, reference, and invoke the services provided by the WCF Web Service. This part will present the rational reasons for choosing technologies to develop client applications and the detail implementation architectures of the client applications using the chosen technologies. This part will start with the explanation about the varieties of types of client applications which make the case of "no magic bullet" as there is no one complete solution in development of client applications. It will next explain the design architectures in development of client applications using WPF and Silverlight. It then explains about the design architectures and development of client applications for other devices (such as ProntoPhilips, iPhones, iPod, etc.,) using other technologies (ProntoScript, CommandFusion, etc).

As depicted in Figure 8, in this era of pervasive computing, there are many devices in building automation industry with different processing powers and different memories capabilities (Cell-phones, PDA(s), Portable Devices, Computers, and so on). They also have different operating environments. In addition, for different customers there are different client applications' requirements. Therefore, the development of the client applications seems to be a

difficult step at the design aspect. A solution should be as compatible with as many devices, operating environments, and as flexible in building the client applications as possible.

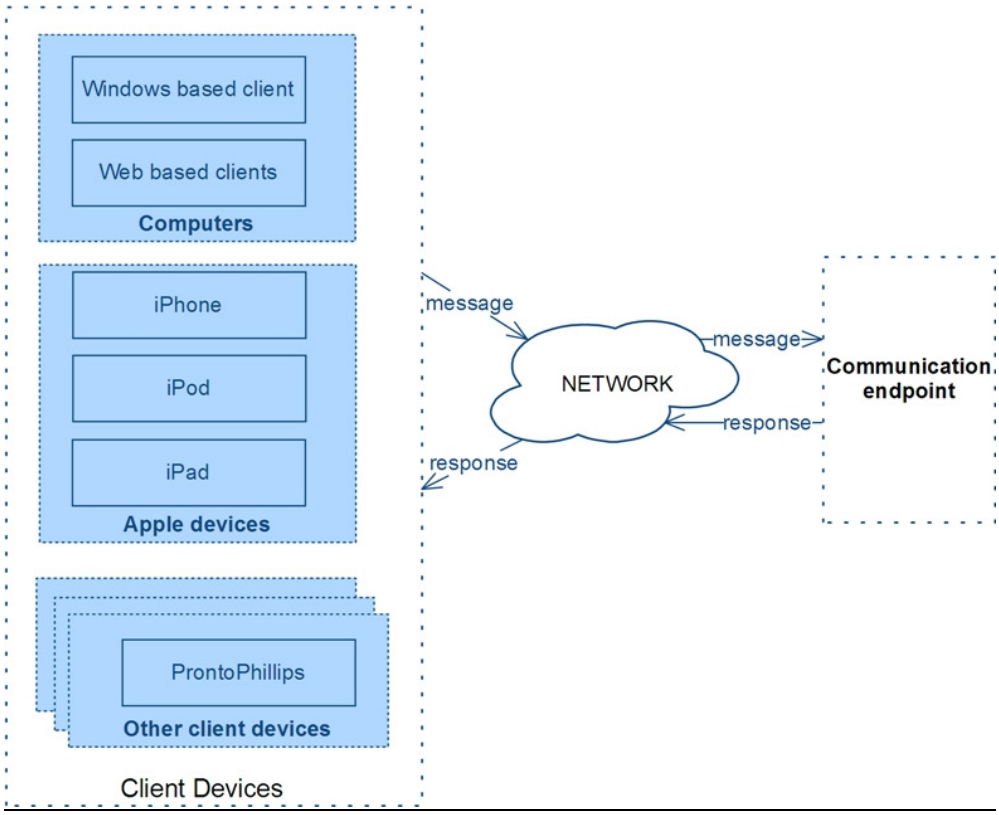


Figure 8 Varieties of Client Devices in Client Applications Development

There is no "magic bullet" as there is no "one complete solution" could solve all these problems. For instance, one requirement is about the duplex asynchronous communication between the Web Services with the client applications. One example of such case is the *Security System* from a *House* would like to push an alarm signal to a web-client which the user is browsing in a browser. The server cannot push data to the normal web-client due to the security reason.

If WPF or Silverlight is used to develop the client applications, the WCF Web Service is able to push data to the WPF and the Silverlight client applications (with little modification and code for Silverlight). In case of the WPF, a WPF application is a Desktop based application and a user needs to install it and executes it before having it running in their devices. That means the client trusts this client application and therefore the WCF Web Service is able to initialize a communication channel to this client and push data to it. In case of Silverlight, Silverlight is actually also an application which is running in the web browser. It is as Flash application but it has a more powerful feature which is instead of using Scripting programming language as its code-behind, Silverlight has C# as its programming language for it code-behind. *Microsoft Silverlight 4 Beta* unveiled at PDC 2009 enables Silverlight clients to have a duplex communication with a Windows Communication Foundation (WCF) service using the net.tcp protocol [19].

However, the WCF Web Services cannot push data to a normal web application such as a normal ASP Web Application. The reason for a server cannot push data to a web browser is due to security concern of client devices running the browsers. In case, a normal web application (such as ASP web application) would like to receive update data coming from a server, it needs to implement a polling implementation to poll the server every period of time to receive the update information. The polling is relatively complex to be implemented and managed. Moreover, the receiving data coming from polling may not be up to date (since the data is polled every period of time).

Another reason for using WPF and Silverlight for developing client applications is their projects' types, if developed in Visual Studio 2008, has wizards allows user discover, reference, and create an reference instance to the web service with only few clicks. They make the development of client applications easier than ever. In other words, they are designed to offer manageable approach to creating web service clients. With the way of designing the core application server as described

in **Part 4.2,** if the client application has a reference to the service provided by the WCF Web Service and it would like to send a command to the core server, it only makes a call to the server with the corresponding message. The core server once gets the command will know how to parse the command to get information about which command and for what add-in device it should raise an event to ask the add-in device to execute the command. With these reasons, the WPF Framework is used to develop Desktop based client applications and the Silverlight is used to develop Web based client applications for computers as client devices.

In Figure 9, the main programming interface for the WPF and Silverlight application is the *IWPF interface*. It contains two methods, the first one is of form "*update(string)"*, this method lets the application update the interface to reflect changes sent from the WCF communication server. Another member is the event *OnCommandRequest(EventArgs)* an event which is raised when a client interface would like to send a command to the WCF Communication server.

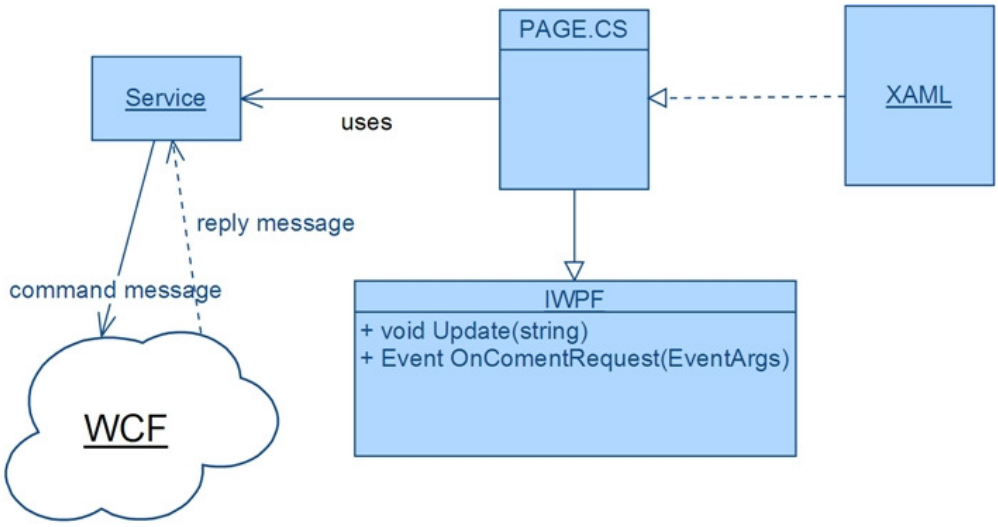


Figure 9 Client Design using WPF and Silverlight Technologies

As depicted in Figure 8 the development of the client applications, besides the Desktop based applications and the Web based applications, there are many other

devices with different capabilities. The first client platform besides the other mentioned two (WPF and Silverlight) is the Windows Mobile OS which runs on many Mobile devices. As the time of writing this document, there is one potential solution for this which is Microsoft promises to support Silverlight in its Windows Mobile OS in the near future. Therefore, if there is no rush from some specific clients the solution for client applications for devices running Windows Mobile OS could wait till the Silverlight is supported in these operating systems.

The next device type which, currently, many customers are requesting to be supported as device which is able to run Building Automation System client applications is the iPhone/iPod/iPad devices type. Programming for iPhone/iPod/iPad devices normally requires working with Objective C programming in Mac OS. This technologically means that there should be another team which is expert at Objective C or a team ready to learn to code Objective C in Mac OS and some costs at buying Mac Computers and Mac OS(s). Fortunately, there is a program called CommandFusion [20] which enables the developers of the building automation systems to design client applications in Windows OS(s) and the learning barrier is relatively small.

The concept of the CommandFusion for developing building automation systems is that it divides commands or communication messages between a client device and a server into several types as followings:

- Digital join – its value is either 1 or 0. For instance, 1 means *on* and 0 means *off*.

- Analog join – its value is normally range from 1 to 65355. For instance, this value may be used to describe the light intensity or the volume level of a sound system, etc.

- Serial join – its value is normally a string message. For instance, this could be used to send a message saying the title of a song playing in a sound system, etc.

- There are many more types of joins (please refer to iViewer Developers' Manual [21] for details list of types and their meaning).

Moreover, it has a communication protocol which could be shortly described as follows (please refer to iViewer Developers' Manuel [21] for further details about this communication protocol). Corresponding to the mentioned types of joins, there are corresponding message types for communication such as a digital join message which is of the form "*d#=value\x03*", an analog message which is of the form "a#=value\x03", and serial join message which is of the form "*s#=value\x03*". In which, the "#" means the number of the join defined by the developer (for instance 1 for button number 1 and 2 for label 2, etc), value is the value of the join that the command would like to send (0, 1 for digital join for instance). And "*\x03*" is the message delimiter, which is used to identify end of message.

The described definitions of join types, and the communication protocol for message exchanging are the concepts to build the infrastructure of CommandFusion. With this infrastructure, CommandFusion provides developers with a GUI Designer and a simulator called "*iViewer*". The GUI Designer is a Windows application which lets users specify the client application interfaces with primitive controls such as button, panel, label, etc., and bind them to digital join, analog join, serial join, etc., correspondingly. This GUI Designer is only a tool for the Windows Developers to develop the interface and the commands of the client applications. In other words, it is used to describe the client interfaces and the client behaviors. The result is a configuration file which is not the real application going to run in iPhone/iPod/iPad devices. The real interfaces and

behaviors of the client applications are realized by the *iViewer* application which runs in the iPhone/iPod/iPad devices. *iViewer* acts as a simulator which parses the configuration files and builds the real client.

This solves well the problems of building the client applications for the iPhone/iPod/iPad devices. One problem left is that the CommandFusion uses its own communication protocol as briefly described above which is different from that of the Core Server and the WCF Web Services which have been built to handle the commands' messages sent from client applications. However, the two protocols are pretty much similar, and it is relatively simple to build an adapter which translates messages coming from the iViewer client applications to messages which are understandable by the WCF Web Services and the Core Server application built.

The next famous and important client device in the area of the current building automation industry is the ProntoPhilips (http://www.pronto.philips.com) device which supports the ProntoScript programming language. The ProntoScript is built based on the JavaScript standards. Therefore, making client applications for ProntoPhilips is actually developing them using JavaScript. The main idea in developing client applications for this type of device is to reuse as much as possible the development artifacts we have got so far. Moreover, the client interfaces should be unified with those for other devices (such as the interfaces of applications developed in CommandFusion for the iPhone/iPod/iPad devices). The programming or communication protocol should be the same between client applications for this device type and other developed client applications for other device types. Specifically, the communication from client applications developed for ProntoPhilips device could communicate well with severs developed for CommandFusion applications. Therefore, the communication protocol should be the same. Meaning it also divides it signal types into list of join types such as

digital joins, analog joins, serial joins. The message format is also the same, of the form *"joinType#=value\X03"*. For detail information, please refer to [21] or previous part about the message format and the communication protocol between CommandFusion client applications and its server.

The solution for this problem is a ProntoScript infrastructure which builds the controls and commands as used in the CommandFusion for the ProntoPhilips devices. In CommandFusion the GUI Designer provides the infrastructure which is built in their "*iViewer*". It decides the behavior of its controls (sometimes called widgets in CommandFusion). For instance, a button can be bound with a digital join and a join number. When a button is bound with a join number (10 for example), and it has a value (1 for example) CommandFusion has an infrastructure to automatically build and send a command "*d10=1\x03*" to the CommandFusion command handler server when the button is clicked. In the other way, when a message is received from the server such as "*d9=0\x03*", the infrastructure will parse the message and know this message is for widget bound to digital join number 9, and the value is 0 (indicating the button should be in an inactive state for instance), then this infrastructure sets the button to inactive state. Similarly the infrastructure builds the behaviors for the other types of controls (widgets). The aim of this ProntoScript infrastructure is to build a ProntoScript (JavaScript) library which implements ProntoScript controls with behaviors as widgets in CommandFusion [20].

Figure 10 depicts the architecture of the implemented infrastructure (as a ProntoScript library). It has a *TCPConnection* component which is used to send/receive the commands to/from the Command Processing Server (CommandFusion command processing server, in this case). It builds the control classes for widgets (button, panel, gauging, etc). For each type of widgets, there is a *MessageProcessor*. The appearance of a widget depends on its *MessageProcessor*.

When a message comes from *TCPConnection*, it is parsed by the *MessageProcessor*. This will pass the message to corresponding *MessageProcessorType* (button message processor, panel message processor, etc) based on the header of the message (*d1* for button 1 or *a1* gauging 1 for instance).

The widget message processor (*MessageProcessorType*) will parse the value in the message and decide the appearance of the widget. One example of the flow is: a message of form "*d10=1\x03*" comes to the *TCPConnection*. The message processor will process this and know that its header is "*d10*" which is a digital join and should be processed by the *DigitalJoinMessageProcessor*. The *DigitalJoinMessageProcessor* processes the message and see the value is "*1*". This indicates the button should be in the "*active state*" and therefore sets the button in "*active state*". Similarly the behaviors are built for other widgets types.

For the way of communication coming from the client widget to the CommandFusion command handler server, when a command is activated, its content (value and join header) is forwarded to the command maker. Based on these values, the command maker makes a command and uses the *TCPConnection* to send the command to the server. For instance, a button which is bound to digital join 10 with value 1, when it is clicked, its values (*d-digital join, 10-join number, 1-value*) are passed to the command maker, then it will make a command of form "*d10=1\x03*" and then use the *TCPConnection* to send the command to Command Handler. The command handler knows the digital join number 10 is used to switch the light for instance, and therefore, switch the light on (as 1 indicates on status).

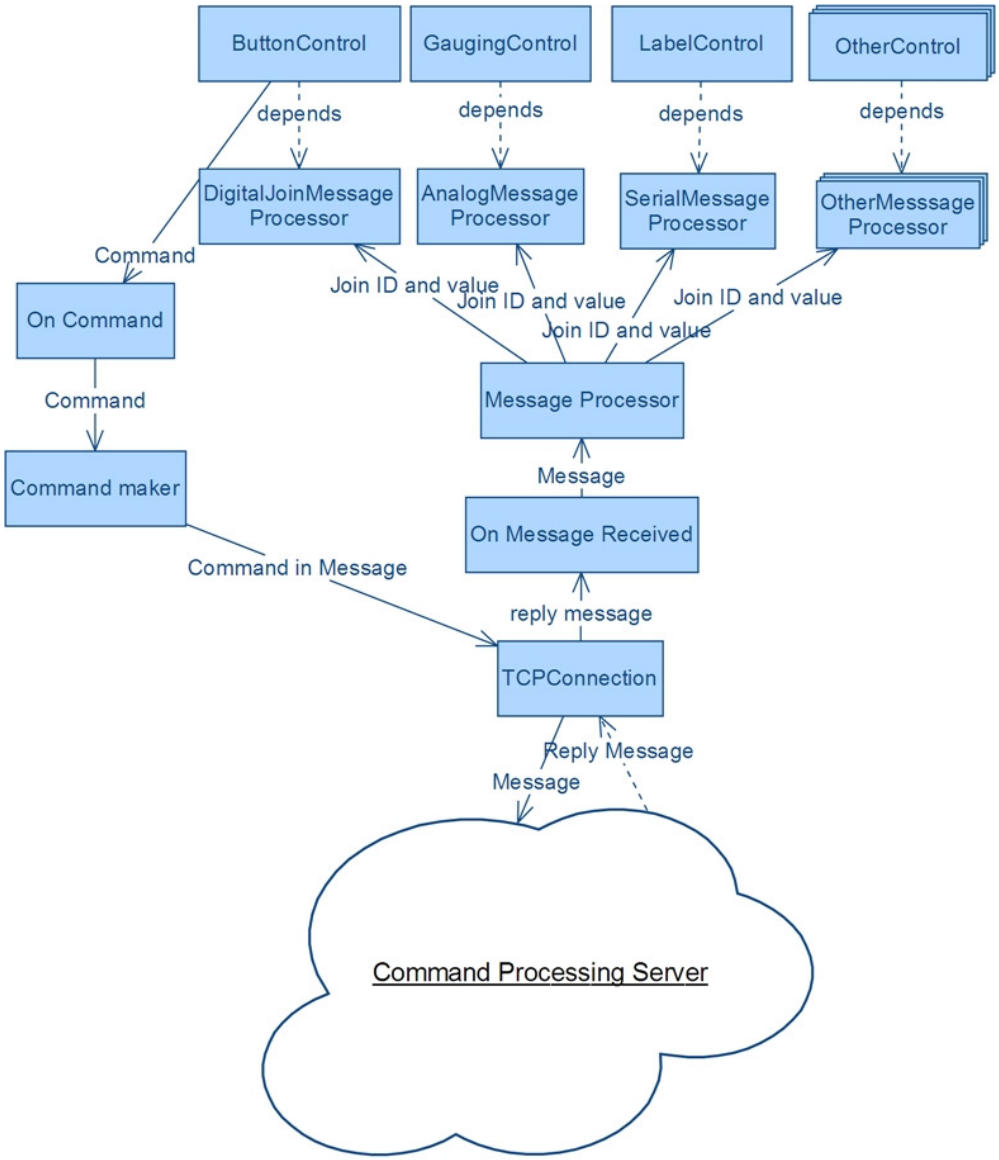


Figure 10 Architecture implementing ProntoScript Widgets.

Using this infrastructure, the programmers can easily develop ProntoScript client applications with less coding requirements since the behaviors of the widgets are already implemented by the infrastructure. Moreover, the developers can develop client applications for ProntoPhilips as the way they design client applications for

CommandFusion. This leads to ease and rapid client applications' development. Furthermore, client applications' behaviors and interfaces are unified with what appears in "*iViewer*" client applications for iPhone/iPod/iPad devices. These unifications lead to higher usability.

As a short summary, there is no "magic bullet" or no single-complete-solution to build a generic client application for all types of client devices in building automation industry. They have different processing powers/memories capacities, different operating systems and different programming bases. So the aim is to maximize the reusability of the artifacts made from one client type in others. In addition, the interfaces and the behaviors for different types of client devices should be as unified as possible.

In this implementation, WPF is used to build desktop based client applications; Silverlight is used to build web based client applications for Computers used as client devices. Silverlight is also used to build client applications for some client devices running Windows Mobile OS which supports Silverlight client applications. For iPhone/iPod/iPad devices, CommandFusion is used. For ProntoPhilips device, a library is built in ProntoScript which simulates the infrastructure of CommandFusion. This library allows developers develop client applications for ProntoPhilips as developing for CommandFusion. Furthermore, the client applications for the two types have the unified interfaces and behaviors.

4.4 Development of the GUI Designer and Simulators' components

This part will present the challenges which are the varieties in client operating systems with different capabilities and different client interface requirements. Then it will provide a solution which is a GUI Designer and explain how providing a GUI Designer would help solving the problem of varieties in client needs in interface requirements. Next it will explain why and how having GUI

Designer output as an XML configuration file to be loaded/executed by simulators will solve the requirements in multiple programming platforms and operating systems.

Previous part (**Part 4.3**) developed some main client applications for popular types of devices and popular known types of users. It also described the development of the client applications which mentioned times the varieties of the different needs in client application interface requirements as unknown information. These problems could be mainly categorized into two types:

- The first is the unknown information about what type of user needs in client applications. For instance, one customer may need to have a camera surveillance system together with a lightning system in their client applications; another may need a camera surveillance system together with a heater control system and a sound control system, not a lightning system (for some reason). It is nearly impossible, or not efficient to have one single client interface program which satisfies all these combination of needs.

- The second is that we can't know what types of client devices with what types of operating systems and basing on what types of programming languages which are going to run the client applications. For instance, one customer may have a computer which is running Windows OS, another may like to run their client applications in an iPhone/iPod/iPad device, some may want to run in Windows Mobile, some other may use ProntoPhilips, etc.

The first problem can be solved by providing users a Client GUI Designer which enables the users (customers or developers) with little programming experiences to build applications by their own. The interface components are built in previous

step (**Part 4.3**), they can be primitive controls (such as button, label, and so on) or custom defined controls (such as set of controls for lightning control system and so on). The GUI Designer will let the users visually design (drag and drop) the interface components and wire them together to create their own needs of client applications.

The second problem of varieties in client devices and their platforms is solved as each client application produced by the GUI Designer will be in a configuration file which is in XML format. XML format is generic and common for all types of programming languages in all types of operating systems (in building automation industry) because they all can read and parse XML file for information. For each type of devices or operating systems, there is one simulator developed which will be able to parse the common XML configuration file and builds the interface and behaviors of the client applications as defined in the XML file.

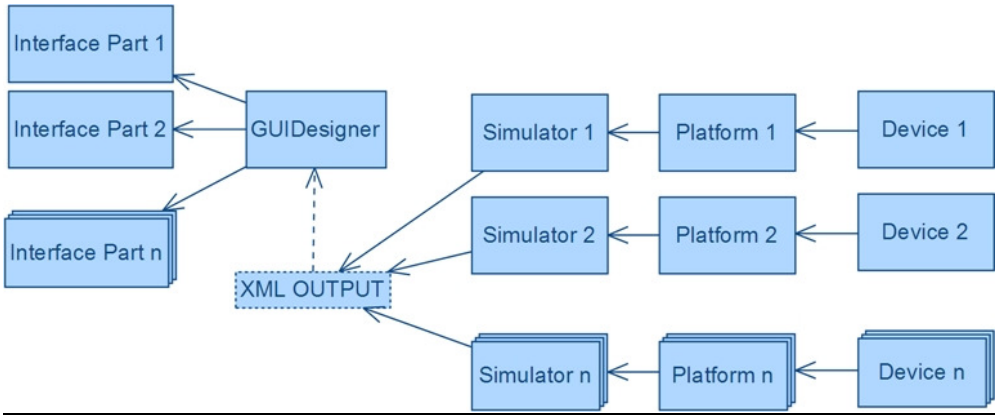


Figure 11 Generic Design Architecture of GUI Designer

Figure 11 depicts the generic design architecture of the development of the GUI Designer implementation. In this implementation, the common interface components are identified (and mostly were built in previous part, **Part 4.3**).

These common interface components could be primitive controls such as a button, a panel, a label, etc or a set of user defined, related set of controls such as a set of controls for a dimmer including of two buttons "*dim-up*" and "*dim-down*" and so on. The developers need to analyze the client applications to see the reusability of the interface components to create them. Next, the GUI Designer is a graphical interface which allows the developers to drag and drop interface components into the designer, arrange them, and wire them together as the user needs. Finally, the result is actually a configuration file in XML output which describes the interface components, their values, and their behaviors in XML format. For each type of platforms for client devices there is one simulator built. This simulator will read the XML Configuration file and create the interface with behaviors for client applications for its own platform correspondingly.

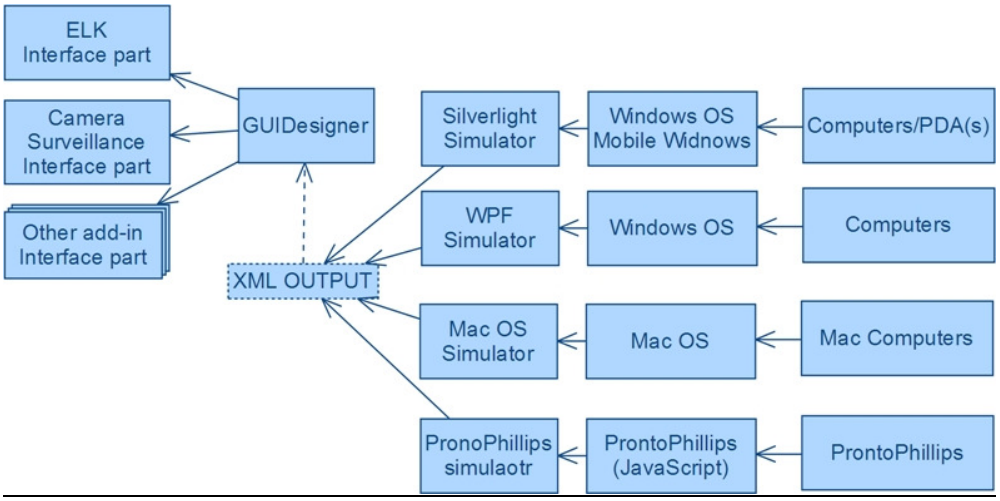


Figure 12 Application of GUI Designer Architecture in Building Automation System

Figure 12 depicts the application of the architecture described in this part in implementing Building Automation System. The interface components are ELK Interface component, Camera Surveillance interface component, and so on. They are built in client development part, **Part 4.3**. Several simulators are developed

for types of devices and their operating systems such as Silverlight simulator, WPF Simulator and so on.

As a short summary, applying this architecture in developing client applications overcomes the varieties in the needs of client interfaces by providing a simple GUI Designer which lets users create client applications by visually combining the common interface components with less programming experience requirements. Its XML Output and Simulators overcome problem of having to create many applications for many platforms.

Chapter 5

IMPLEMENTATIONS

This chapter will present some technical descriptions of the implementations of the Building Automation System at Omniabit.s.r.l. The readers who are not interested in technical details could skip this chapter. My main job at Omniabit. s.r.l is not coding the real applications but more about researching what system architectures, what technologies (programming languages, frameworks, and libraries) should be used in implementation of each part of the specified Building Automation System. Based on the studying, I will only code the samples which I will present and discuss with my Supervisor Mr. Manuel Elia and other developers about how to use them. Specifically, I code one ELK add-in for the core server, one ProntoScript library for the client applications run in ProntoPhilips device and the GUI Designer.

5.1 Implementation of core functionalities server

This core functionalities server is implemented using *System.Addin framework* in *C#* programming language. The pipeline is in charge of discovering, loading and activating its add-ins. It is implemented using the pipeline builder [14] which is made by Microsoft. My job in this part is to implement a sample add-in to control the ELK M1 security system [23]. The M1 Gold offers a rock solid security foundation with the most imaginative automation functionality available. Key features include the versatile keypad, flash memory firmware, fast PC programming, and ease of installation. It offers the flexibility to integrate with industry standard systems and components for easy upgrades and system expansion. This ELK add-in is implemented in C#. It includes the following functions.

RequestAllLogs() function. It has no parameters and is used to request all the logged data which were logged in the ELK Device. According to the ELK Protocol [24], the ELK device is able to record 511 lines of log which is indexed from 1 to 511. If a client program calls to this function the entire log file will be returned.

RequestPagedLog(int pageIndex, int pageSize) function. This function is used to request for specific number of log lines (***pageSize***) starting from a specific index (***pageIndex***) instead of entire number of logs.

- ***pageSize*** is the number of log line per page that the client application would like to display (for instance 10 means every page there are 10 lines of log).

- ***pageIndex*** is the starting index of the log

RequestPageCount(int pageSize) function. This function is used to request number of pages of log based on the pageSize. For instance if the total number of log lines are 500 and the pageSize is 10, this function will return 5 as number of log pages. This is used to display to the client applications the available log pages that the user could view.

- ***pageSize*** is the number of lines of log for a single page.

RequestStatus() function. This function is used to request the current status of the ELK Device. The status could be ***arm***, ***disarm***, or ***alarm***. For specific descriptions of what is the meaning of each status please refer to the ELK Protocol [24].

RequestArmStatus(int area) function. This function is used to request the arm status in specific area. The arm status could be ***a0*** for ***disarmed***, ***a1*** for ***armed to away***, etc.

- ***area*** is the id of the area that caller would like to ask for the arming status.

RequestReadyStatus(int area) function. This function is used to request the ready status in specific area. The ready status could be ***ready to arm*** or ***not ready to arm***, etc.

- ***area*** is the id of the area that caller would like to ask for the ready status.

RequestAlarmStatus(int area) function. This function is used to request the alarm status in specific area. The alarm status could be ***fire alarm*** or ***medical alarm***, etc.

- ***area*** is the id of the area that user would like to ask for the alarm status.

RequestArmStatusMessage(int area) function. This function is used to request for arm status message ("***disarmed***", "***armed to away***", etc instead of their id as "***a0***", "***a1***", and so on).

- ***area*** is the area that the caller would like to ask for arm message

Disarm(string userCode, string area) function**.** This function is used to disarm the ELK system.

- ***userCode*** is the four or six characters of the user code.

- ***area*** is the area that the user would like to disarm.

The next functions are ***ArmedAway, ArmedStay***, ***ArmedStayInstant***, ***ArmedToNight***, ***ArmedToNightInstant***, ***ArmedToVacation***, ***ArmStepToNextAwayMode***, and ***ArmStepToNextStayMode*** functions**.** They all have ***(string userCode, string area)*** as their parameters. These functions are used to activate the Armed Away, Armed Stay, Armed Stay Instant, Armed To Night, Armed To Night Instant, Armed To Vacation, Armed Step To Next Away, Armed Step To Next Stay Mode correspondingly for the specific area of the ELK system.

- ***userCode*** is the four or six characters of the user code.

- ***area*** is the area that the user would like to disarm.

For summary of these functions, their descriptions and their parameters please refer to **Appendix A**. After this add-in is implemented and added to the add-in folders. The core server has the functionalities used to control the ELK System. Other add-ins of the Building Automation System are developed by other developers using this add-in as their sample add-in.

5.2 Implementation of client applications

The client applications are implemented in several programming languages such as WPF, Silverlight, CommandFusion, and ProntoScript. The programming languages such as WPF and Silverlight are familiar with the developers at Omniabit s.r.l.. For iPhone/iPad/iPod devices, the CommandFusion GUI Designer allows users with less programming requirements visually design the client applications. Therefore, the developers at Omniabit s.r.l do not have much problem developing client applications using these programming languages and tools. However, developing client applications for ProntoPhillips devices requires developers to study to code with ProntoScript. To reduce the development

barrier, I developed two ProntoScript libraries which let the developers easily create the client applications for ProntoPhillips with less ProntoScript programming requirements.

The first library is the ***com.omniabit.PageLibrary***. It is called a page library because it is going to be loaded in every page of the client applications run in ProntoPhilips devices. This library has one ***TCPSocket*** named ***socket*** which is used to send and receive message to/from the core functionalities server. It has the ***onData*** handler which is used to parse every message comes to the ProntoPhilips device running the library. This ***onData*** handler will parse message as analog join, digital join, serial join, and heartbeat join with regular expressions "***^a***", "***^d***", "***^s***", "***^h***", correspondingly and bind the value to widget which is bound to the join.

This library also provides a class called ***CFButton*** (CommandFusion Button). This class is used to enable a primitive button to behave as button in CommandFusion with properties such as Simulate Feedback, SyncState, on press value, on release value, and so on (for the complete list of properties and their meanings please refer to [21]). This CFButton class has a function called ***btn_initialize(ipbutton, simulatefeedback, syncstates, value, state, releaseValue).*** This function is used to set the properties for the button.

- ***ipbutton*** is the reference to the button that we would like to set the property values

- ***simulatefeedback*** if is set to true, the button will simulate feedback (toggles its state as pressed or released without waiting for feedbacks from the server), false otherwise.

- ***syncstates*** if is set to true, the button will track the feedbacks from the server for its state and toggle its state based on the value from the feedbacks, false otherwise.

- ***value*** is the value that should be sent to the server when the button is pressed (could be 0 or 1 for digital join bound to this button).

- ***state*** is the state of the button, 1 means it is in normal state (released state) and 0 means it is in pressed state.

- ***releaseValue***, if not null, is the value to be sent to the server when the button is released.

This class also provides two more methods which is the ***onPressHandler*** and ***onReleaseHandler***. They are used to send the ***value*** and ***releaseValue*** to the server when the button is pressed or released correspondingly.

The second library is the ***com.omniabit.CVS***. This library has a Base64 object. This object is used to encode the user name and password the Authentication to the camera server.

This library provides one main function which is ***getImage(cam_w_s, url, username, password)***. This function gets an image from a ***url*** with credentials information of ***username*** and ***password*** and sets the image to the widget named ***cam_w_s***. This function uses a http request (***HttpRequest***) built in ProntoPhilips Http Library (***com.philips.HttpLibrary***) to get the image from the network. Since cameras provide their videos as sequence of images, calling this function every period of time (5 seconds for instance), we will have the widgets displays the videos from the requesting cameras.

Appendix B is a tutorial which guides developers how to use these libraries to create Building Automation System client applications for ProntoPhilips using ProntoScript.

5.3 Implementation of the GUI Designer

This GUI Designer allows users (customers, developers) with less programming experiences to design client applications for Building Automation System. The users can move, resize and rotate objects of any type on a canvas. It has the following functions

- ***MoveResize***: This function is used to move and resize objects without WPF Adorners.

- ***MoveResizeRotate***: This function is used to provide rotation of objects, still without WPF Adorners.

- ***MoveResizeRotateWithAdorners***: This function is used to move, resize and rotate items with the help of WPF Adorners. It also shows how Adorners can be used to provide visual feedback to indicate the actual size of an object during a resize operation.

This GUI Designer has several important classes such as DesignerCanvas, Designer Item, Toolbox, and ToolboxItem.

- ***DesignerCanvas*** is the space that users use to design the client applications. It will keep its size, even if you drag an item far beyond the borders of the canvas. ***DesignerCanvas*** will never notify the ***ScrollViewer*** of a size change, just because there is none. The solution is that we must force the ***DesignerDanvas*** to adjust its size every time an item is moved or resized. Fortunately the Canvas class provides an

overridable method named ***MeassureOverride*** that allows the ***DesignerCanvas*** to calculate its desired size and return it to the WPF layout system.

- ***DesignerItem*** is inherited from ***ContentControl***, so that we can reuse the ***ControlTemplate***. The ***DesignerItem*** provides an ***IsSelected*** property to indicate if it is selected or not. Then it implements an event handler for the ***MouseDown*** event to support multiple selections of items. Finally it implements the template for the ***DesignerItem*** such that the ***resize decorator*** is only visible when the ***IsSelected*** property is true, which can be handled with a simple ***DataTrigger***.

- ***Toolbox*** is an ***ItemsControl*** that uses the ***ToolboxItem*** class as default container to display its items. For this we have to override the ***GetContainerForItemOverride*** method and the ***IsItemItsOwnContainerOverride*** method. Additionally the ***Toolbox*** uses a ***WrapPanel*** to layout its items.

- ***ToolboxItem*** is the place where drag operations are actually started if you want to drag an item from the ***toolbox*** and drop it on the ***canvas***. It will take care of how to copy an item from the drag source (***Toolbox***) to the drop target (***DesignerCanvas***). In this case, it uses the ***XamlWriter.Save*** method to serialize the content of the ***ToolboxItem*** into XAML.

This GUI Designer also provides several commands such as Open, Save, Cut, Copy, Paste, Delete, Print, Group, Ungroup, Align (Left, Right, Top, Bottom, Centered horizontal, Centered vertical), Distribute (horizontal, vertical), and Order (Bring forward, Bring to top, Send backward, Send to back).

- **Grouping, its** approach to group items is to use a ***DesignerItem*** object that should work as a group container. For this, it creates a new instance of the ***DesignerItem*** class with a ***Canvas*** object as its content. On this canvas, it positions the designer items to be grouped.

- **Save,** to save a design, it uses a combination of XML and XAML. For the *DesignerItem* related data it uses XML, and the content is serialized to XAML.

- **Open,** when loading a design from an XML file, we have to start with the designer items because we need their information to wire them together.

- **Copy, Paste, Delete, and Cut,** the Copy and Paste commands work analogous to the Open and Save commands, except that they are applied only to the selected items and that they read and write the serialized content to the Clipboard. The Delete command simply removes all selected items from the designer canvas' Children collection, and the Cut command finally is a combination of Copy and Delete command.

- **Align, and Distribute,** reference item for alignment is the item that was selected at first (also called primary selection). This works only when you select items with the LeftMouseButton + Ctrl, or LeftMouseButton + Shift, but not if you use **rubberband** selection.

- **Order,** the ***Panel*** class (from which ***Canvas*** is derived) provides an attached property named ***ZIndex*** that defines the order on the z-plane in which the children appear, so we only have to change that property to bring an item forward or backward.

As a short summary, this chapter presents some implementation details of the development of the Building Automation System. It dose not present all the implementations of the whole systems but only the development of some special samples (the ELK add-in) and some libraries (the ProntoScript library). The other developers could use these samples to code the other parts of the system. Additionally, it also presents the implementation of the GUI Designer.

Chapter 6

RESULTS AND DISCUSSIONS

This chapter will evaluate the results coming out from applying the architectures and technologies in developing the Building Automation System at Omniabit s.r.l. Then it will present some further discussions about noticing points in applying the architectures and using the technologies.

6.1 Results and discussions of the core server development

The artifacts coming out from this development step is a core server which implements main functionalities for the Building Automation System. These functionalities can be easily modified or extended. Any unforeseen functionalities from unforeseen customers to control unforeseen building automation devices can be relatively simple to be extended to the core server without affecting other functionalities even if the system is already deployed in the industry.

At the development point of view, the functionalities are divided into modules. This increases decoupling among modules for functionalities and they can be developed in parallel. Besides, some customers may require some functionalities some other customers may require other functionalities. The developers only have to add/remove (copy/delete) the add-ins for those functionalities in the add-in folder.

The important design decision in development of this part is having a generic method of form "*Execute (int id, string methodName, Object[] arguments)*" to represent any method in add-in *id*, with method name *methodName*, and arguments stored in *arguments*. This generic method representation lets the *contract* between the host

and the add-ins stable but still let the client programs which know the method call it.

The idea of having this implemented using *System.AddIn namespace in C#* is just implementation decision for ease of coding and availability of development tools. The add-in features can be implemented using different methods with different programming languages as long as we are able to develop the add-in separately and deployed relatively simply and then they are discovered and loaded automatically. In other words, using add-ins for extensibility of functionalities can be done in many programming languages in many applications. However, in this specific case, due to the availability of tools and programming experience of developers, the *System.AddIn namespace in C#* is used.

6.2 Results and discussions of the communication channel development

The artifact coming out from this development part is a network communication endpoint as a Web Service implemented in WCF Framework. This WCF Web Service is able to push data to client applications which support duplex communication. It also enables client applications to communicate with it in several transportation protocols, especially HTTP and TCP protocols due to many devices in Building Automation area are able to communicate with each other in the network via HTTP and TCP connection.

In this era, the internet connected systems such as world-wide-web is the best access interface over the heterogeneous interconnected systems with mobile components. Furthermore, this communication channel is actually a tier in the famous, currently practiced "three tiers architecture" with *Presentation tier* as browser (Netscape navigator, Internet Explorer, etc) and other client applications, *Functional tier* as application and network management functions server, and *Data tier*. Furthermore, making these Building Automation Systems distributed systems

meaning they are parallel systems executed in multiple processors. This increases the performance and scalability over the network.

This Web Service as the network communication endpoint is a stable interface published to the public so that the client applications need no more to discover where the functionalities are but always send the command to this service endpoint. The interface is stable and unified. It only needs to have enough information about the method it is calling (in what add-in, what method name, and the required arguments). Then it will call to the generic method of form "*Execute(int id, string methodName, Object[] arguments)*".

The important component in this part is the *Communication Manager* (in Figure 7) implemented in the *host* of the *communication pipeline*. This *Communication Manager* is built in the event based style. Meaning when there is a message coming from a client application the event *OnCommandReceived* is raised. The *handler* of this event will parse the message and direct the method call to appropriate add-in and call the function. The communication is asynchronous. Meaning when an add-in has a message (may be after processing the method call from client or some events happened in the being-monitored devices) to send back to client applications the *OnMessageReceived* of the *Communication Manager* will be raised. Its *handler* will create the message and forward it to the appropriate client applications.

The important design decision in this communication channel is the design of the communication protocol. What message and message format are important in order for client applications and the communication managers to understand each other in order to forward the method calls. Different devices may have different communication protocols. Therefore, it is important to design a standard protocol which is compatible with as many other protocols as possible or at least it is easy to create an adapter component which is able to convert the message from one protocol to another. In the development of Building

Automation System in Omniabit s.r.l., the protocol used is the protocol for communication of CommandFusion mentioned in [21]. This protocol is simple but sufficient for applications in building automation industry. It has ability to control all the building automation devices and ability to deliver response messages from them to the processing server.

Also in this development part, choosing WCF to implement this communication endpoint is just a choice of technologies due to its availability of tools, experiences of programmers, developing programming languages, and the compatibilities with the programming languages and tools used in developing other components such as *System.AddIn* in server core component, WPF, and Silverlight in client components. WCF makes the development of endpoints easier than ever. In other words, WCF is used to offer manageable approach to creating Web services and web service clients

6.3 Results and discussions of the client development

The artifacts coming out from this development part are the client applications for some specific devices, their specific operating platforms, and specific programming languages they are based on. These may be the ready-to-use client applications or client application components (part of a complete application). The ready-to-use client applications are the ones for the popular types of customers with specified interface requirements or for the specific types of devices. Normally, these are the devices which are hard to build simulators for. These may also be client application components. These components will be used by the GUI Designer to design complete client applications. The ideal development of this GUI Designer should be as specified in next part, the GUI Designer should be able to describe all the interface types and behaviors of client applications in XML and then there can be simulators which will be able to read these XML configuration files and realize the client applications.

These client components should be analyzed carefully so that they should be as common as possible, yet still be useful. Since some primitive controls (such as button, label and so on) are always common but they are not so useful in reusability compared to, let's say, a set of primitive controls which control a device such as an interface component which controls the Lightning system. This set may have two buttons for light on, light off, then gauging control which displays light level, or a slider for light level modifier, etc. These primitive controls can be grouped into client components. Then they will be reused by many client applications. They will be used by the GUI Designer to design client applications configuration. These configuration files will be used by the Simulators to really realize the client applications running in the clients' devices.

The main important part in this client development is to have the client applications running in different devices basing on different programming languages, yet they still have as similar interface appearances and similar behaviors as possible. This increases the usability client applications. For instance, the CommandFusion provides the users with a GUI Designer to design client applications for the iPhone/iPod/iPad devices. This GUI Designer provides users with primitive widgets (controls) with built in behaviors. One example of such the behaviors is for instance, when a button named *buttonName* is clicked and it is bound to a digital *join number 10*, with a *value 1* then a command as "*d10=1\x03*" will be sent to the defined server. This behavior of primitive controls are built by an infrastructure of CommandFusion (please refer to [21] for detail information about this infrastructure).

In the other hand, the ProntoPhilips device has ability to run client programs in ProntoScript. ProntoScript is designed based on JavaScript standard. This means, the client programs may have the primitive controls such as buttons, labels, text, etc., as CommandFusion. However, there are no such the built-in behaviors as of

the controls of CommandFusion. Furthermore, there is already a CommandFusion command handler which is able to handle the command sent from the CommandFusion client applications. Therefore, there should be a ProntoScript library built to enable the behaviors of the ProntoScript primitive controls as of the controls in CommandFusion. Moreover, this library will enable also the developers to design programs for ProntoPhilips in the similar way as they do for CommandFusion. This would reduce the learning barrier to code in ProntoScript for other programmers once the library is done. Another advantage of such library is that the programmers don't have to code again and again such the behaviors for every primitive control in every place of the client applications. And the client applications developed using this library will behave as the client applications developed by CommandFusion GUI Designer.

6.4 Results and discussions of the GUI Designer and simulators development

This development part is a very challenging part, out from which the ideal artifacts that the developers are striving to have are a GUI Designer and simulators for realizing the client applications, produced by the GUI Designer, in individual platforms.

The GUI Designer ideally should allow users to define the client applications for any platform (iPhone/iPod/iPad, Computers, ProntoPhilips, etc.,) and output them to XML configuration files. However, for many of device types, many of programming bases it is difficult to build simulators for them. The current version of the GUI Designer supports the design of Silverlight and WPF client applications. However, it is still feasible to build a GUI Designer application which allows the users to design client applications for different types of devices with different capacities running in different platforms and basing on different programming languages. The main idea of such GUI Designer is providing the

users with common client components and let the users visually design client applications then export the result to XML configuration files. Since the output is in XML format, the job is now for the simulators to read/parse the file and realize the client applications as described.

In other words, it is feasible to build one simulator for each client device with its platform and based programming language. For instance, there should be one WPF Simulator, one Silverlight Simulator, one iPhone/iPod/iPad simulator, one ProntoPhilips simulator, and so on. However, some client devices are based on less powerful programming language such as the ProntoScript for the ProntoPhilips device. Currently the GUI Designer allows users to design the WPF and Silverlight client applications. We still have to develop the client applications for other devices such as iPhone/iPod/iPad, ProntoPhilips using their own provided GUI Designers.

This development part will take more effort to have a unified and stable solution compared to developments of other parts. Specifically, the development of the core server with extensible functionalities is stable. The communication channel with stable interface for all types of controlling communication is also stable. Only the client development part would take more time and effort to be really stable and flexible.

Chapter 7

CONCLUSIONS

This writing presents a four step methodology for development of extensible, distributed, software systems which satisfy the challenge of requirements change over time. In each step it also presents the design architectures and the technologies used to develop the components of the over all software system. The four steps include:

- Step 1: development of the core server
- Step 2: development of the communication channel
- Step 3: development of the client applications and client application components (parts of complete client applications)
- Step 4: development of a GUI Designer and simulators.

In Step 1, the design decision is to have the functionalities built as add-ins. The core server is first developed with core functionalities for popular devices and popular customers. For each new functionality comes to life, a new add-in is implemented to realize such the functionality. The add-in is discovered, loaded, and activated automatically by the system. In the implementation point of view, the *System.AddIn framework in C#* is used to realize the add-in behaviors. It is in charge of discovering, loading, and activating the add-ins. The main important design point in this step and also the next step is having a generic method of form "*Execute(int addInID, string methodName, Object[] arguments)*". This generic method is able to express all the functions inside any add-in. It helps to have a

stable programming interface among the add-ins, the communication channel, and the client applications.

In Step 2, the design decision is to have the system's functionalities exposed over the network via a Web Service. This web service enables the client applications over the network communicate with the functionalities built in the core server. The client applications need no more complicated code to discover where the functionality is located or how to load, activate, and call to the functionalities. This step makes the system a distributed system which benefits not only about the accessibility but also the performance increase due to multi-processors are processing the tasks in parallel. The technology used in this step is the Window Communication Foundation (WCF) framework. WCF is used due to the availability of its tools, ease of development, and development background of developers.

In Step 3, the design decision is to have the complete client applications built for most popular devices and most common customers' needs and at the same time identify and build the common client components (partial components of complete client components) which will be combined together to create complete client applications at the later time. These client components are important, because there is no "magic-bullet" for building one complete client application for all types of devices and all types of customers' needs. Therefore, we have to provide these client components which will be combined together by the customers for their needs (using a GUI Designer). In this step the Windows Presentation Foundation (WPF) and Silverlight are used to build Desktop based and Web based client applications/components correspondingly. Furthermore, for specific devices and their operating platforms such as iPhone/iPod/iPad the CommandFusion is used and for ProntoPhilips the ProntoScript is used to build client applications.

In Step 4, the design decision is to provide the users (can be customers or developers) a GUI Designer that users can visually design their client applications then save them to XML configuration files and to have a set of simulators which realize the client applications in individual client devices' platforms. The GUI Designer will help users with little programming experiences develop their own client applications as they need. The GUI Designer describes the client applications in XML. Each client device platform will have a simulator built for it. This simulator will be in charge of realizing the real client applications in the device based on the descriptions in the XML configuration file.

Chapter 8

FUTURE WORKS

While we have got the core server and the communication channel shown promising efficiency implementation; it is clear that at the current version of the Building Automation System, the development of the client applications is still below what would be desirable. This chapter will present the possibility to make the Building Automation System be more scalable for many types of clients, client devices, and client based programming languages.

The client applications were primarily developed as separated for individual clients' needs. This way of development of the client applications is not constructed to scale well to satisfy the needs in client applications for many different types of clients. In our future works, we plan to address this issue by reworking on the Client GUI Designer. The current version of the GUI Designer allows the users to visually design the client applications from client components. However, the client components and the target applications are of either WPF or Silverlight applications. The GUI Designer is not yet able to support design for client applications of other types of programming languages (the developers are currently developing them separately). When we have the GUI Designer which supports other client applications in different programming languages (such as ProntoScript for ProntoPhilips, Objective C for iPhone/iPod/iPad, etc.,) the issue will be diminished.

Another pressing area of future works is to be able to accommodate the different types of client devices with different capacity in processing powers, memories capacity, operating systems, and supporting programming languages. The first work in this part is to design a schema to describe the results of client

applications (interfaces and behaviors) designed by the GUI Designer in XML. Since at the current era, any client device in building automation industry can read and parse XML data, having the applications described in XML will enable the devices to understand what the interfaces of the applications are and how they behave to construct the corresponding client applications. The current version of the Building Automation System, the GUI Designer is able to describe only the client applications written in WPF or Silverlight. The future version this GUI Designer should be able to describe client applications in different programming languages such as ProntoScript, or Objective C, etc.

The second work in this part is building the simulators for each type of client programming languages. Current version of Building Automation System we are able to build the simulators which could load the XML description of the client applications designed by GUI Designer for WPF and Silverlight client applications and realize the real client applications running in the client devices correspondingly. However, for other types of programming languages we haven't got simulators. The client applications were primarily developed separately for individual client devices running different operating systems basing on different programming languages. Such implementation is not constructed to scale under many types clients' devices. In our future works, we plan to address these issues by working on building the simulators for other types of client applications such as one ProntoScript simulator for ProntoPhilips, one Objective C simulator for iPhone/iPod/iPad, and so on.

While developing the perfect application which is extensible for different types of clients' needs, different types of devices and different types of programming languages represents a particularly difficult challenge. We believe that the development efficiency promises of a GUI Designer which is able to describe client applications for many different types of programming languages in XML

together with simulators built for these types of programming languages in different types of devices will adequately let the Building Automation Systems scale well.

BIBLIOGRAPHY

[1] Jack. G and Jesse. K ".NET Application Extensibility, CLR Inside Out, 2007, Mcrosoft (http://msdn.microsoft.com/en-us/magazine/cc163476.aspx accessed May 25, 2010).

[2] Distributed computing, http://en.wikipedia.org/wiki/Distributed_computing accessed May 25, 2010

[3] Elmasri. R, Navathe. S. B. (2000), "Fundamentals of Database Systems" (3rd ed.), Addison–Wesley, ISBN 0-201-54263-3., Section 24.1.2.

[4] Sape. M. Distributed Systems, 2nd Edition, 1993, Addison Wesley Publishing Company.

[5] Jesse. K. "Brief Introduction to our Architecture for Managed Add-Ins", Microsoft MSDN, Microsoft. (http://blogs.msdn.com/clraddins/archive/2007/02/23/brief-introduction-to-our-architecture-for-managed-add-ins.aspx accessed May 25, 2010)

[6] Jack. G and Jesse. K ".NET Application Extensibility, Part 2" Microsoft MSDN, Microsoft, (http://msdn.microsoft.com/en-us/magazine/cc163460.aspx accessed May 25, 2010).

[7] "What Is Windows Communication Foundation", Microsoft MSDN, Microsoft, (http://msdn.microsoft.com/en-us/library/ms731082.aspx accessed May 25, 2010).

[8] "WCF Feature Details", Microsoft MSDN, Microsoft, (http://msdn.microsoft.com/en-us/library/ms733103.aspx accessed May 25, 2010)

[9] Service Oriented Archtiecture. http://en.wikipedia.org/wiki/Service-oriented_architecture (accessed date May 26, 2010).

[10] "Metadata", Microsoft MSDN, Microsoft (http://msdn.microsoft.com/en-us/library/ms731823.aspx accessed May 25, 2010)

[11] ".NET Framework 4 - Windows Presentation Foundation", Microsoft MSDN, Microsoft (http://msdn.microsoft.com/en-us/library/aa970268.aspx accessed May 25, 2010)

[12] "Top Silverlight Features", Microsoft Silverlight, Microsoft (http://www.microsoft.com/silverlight/features/ accessed May 25, 2010)

[13] "Composite Application Guidance for WPF and Silverlight - October 2009", 2009 by Microsoft Corporation.

[14] Jesse. K. , "System.AddIn Tools and Samples", Last edited Feb 8 2008 at 11:19 PM by Jesse Kaplan, version 12, http://clraddins.codeplex.com (accessed May 25, 2010).

[15] "Middleware", From Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Middleware (accessed June 5, 2010).

[16] In its April 2001 report for AIIM International, "Enterprise Applications: Adoption of E-Business and Document Technologies, 2000-2001: Worldwide Industry Study," Gartner defines EAI as "the unrestricted sharing of data and business processes among any connected applications and data sources in the enterprise."

[17] Gable, Julie (March/April 2002). "Enterprise application integration". Information Management Journal. http://findarticles.com/p/articles/mi_qa3937/is_200203/ai_n9019202. Retrieved 2008-01-22.

[18] Trotta, Gian (2003-12-15). "Dancing Around EAI 'Bear Traps'". http://www.ebizq.net/topics/int_sbp/features/3463.html. Retrieved 2006-06-27.

[19] "WCF net.tcp protocol in Silverlight 4", last updated Wednesday, November 18, 2009, from http://tomasz.janczuk.org/2009/11/wcf-nettcp-protocol-in-silverlight-4.html, accessed June 6, 2010.

[20] CommandFusion, http://www.commandfusion.com, accessed June 6, 2010.

[21] "iViewer Developer's Manual v1.2", http://www.commandfusion.com/downloads.

[22] "ProntoScript Developer's Guide", 2009, Koninklijke Philips Electronics N.V., http://www.pronto.philips.com

[23] "Experience the Powerful M1 Control Family", ELK Products, Inc. P.O. Box 100, Hildebran, N.C. 28637 USA (828) 397-4200.

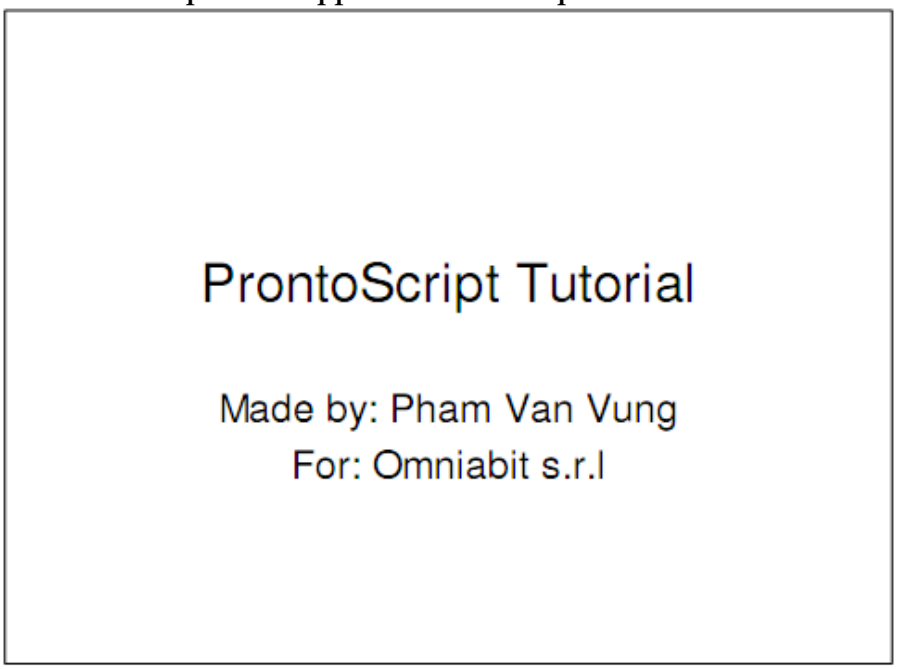
[24] "M1 Security and Automation Controller". Elk Products, Inc. Hildebran, NC 28637 USA.

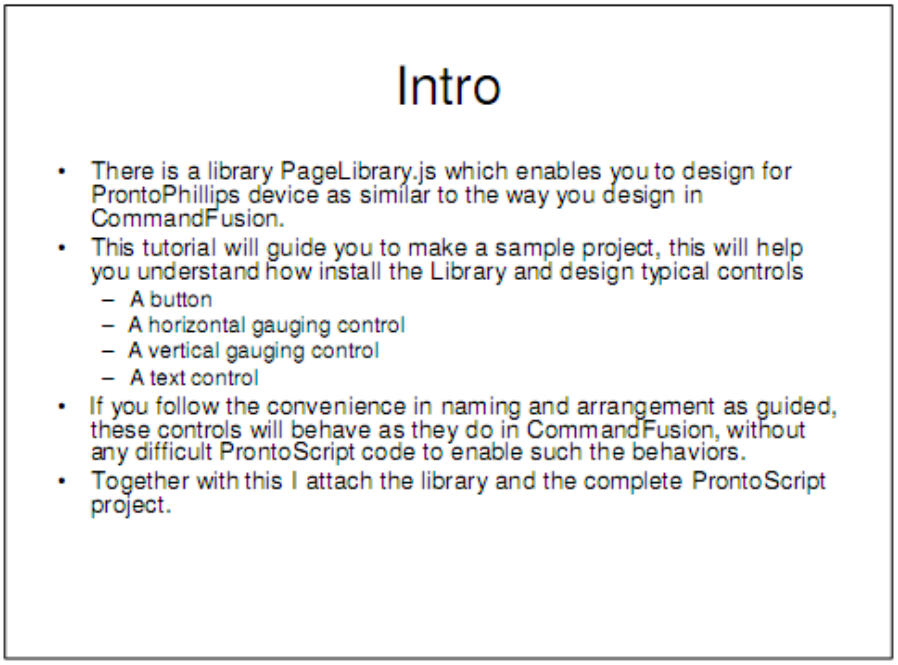
APPENDIX

A. Summary of ELK Functions

Name	Description	Param1	Param2
RequestAllLog()	To request entire log file		
RequestPagedLog(int pageIndex, int pageSize)	To request a page of pageIndex with the size of pageSize	pageIndex: Index of the page	pageSize: Number of rows in a page
RequestPageCount(int pageSize)	To request number of pages of log based on the pageSize	pageSize: Number of rows in a page	
RequestStatus()	To request all the status (arm, ready, alarm)		
RequestArmStatus(int area)	To request arm status (a0: for disarmed, a1: for armed to away...)	area: The arm status for the area	
RequestReadyStatus(int area)	To request ready status (ready to arm, not ready to arm...)	area: The arm status for the area	
RequestAlarmStatus(int area)	To request alarm status (FireAlarm, MedicalAlarm...)	area: The arm status for the area	
RequestArmStatusMessage(int area)	To request arm status message (disarmed, armed to away... messages, not id like a0, a1...)	area: The arm status for the area	area: The arm status for the area
Disarm(string userCode, string area)	To Disarm	userCode: the 4 or 6 chars of user code	area: The arm status for the area
ArmedAway(string userCode, string area)	To activate Armed Away	userCode: the 4 or 6 chars of user code	area: The arm status for the area
ArmedStay(string userCode, string area)	To activate Armed Stay	userCode: the 4 or 6 chars of user code	area: The arm status for the area
ArmedStayInstant(string userCode, string area)	To activate Armed Stay Instant	userCode: the 4 or 6 chars of user code	area: The arm status for the area
ArmedToNight(string userCode, string area)	To activate Armed to Night	userCode: the 4 or 6 chars of user code	area: The arm status for the area
ArmedToNightInstant(string userCode, string area)	To activate Armed to Night Instant	userCode: the 4 or 6 chars of user code	area: The arm status for the area
ArmedToVacation(string userCode, string area)	To activate Armed to Vacation	userCode: the 4 or 6 chars of user code	area: The arm status for the area
ArmStepToNextAwayMode(string userCode, string area)	To activate Arm, Step To Next Away Mode	userCode: the 4 or 6 chars of user code	area: The arm status for the area
ArmStepToNextStayMode(string userCode, string area)	To activate Arm, Step To Next Stay Mode	userCode: the 4 or 6 chars of user code	area: The arm status for the area

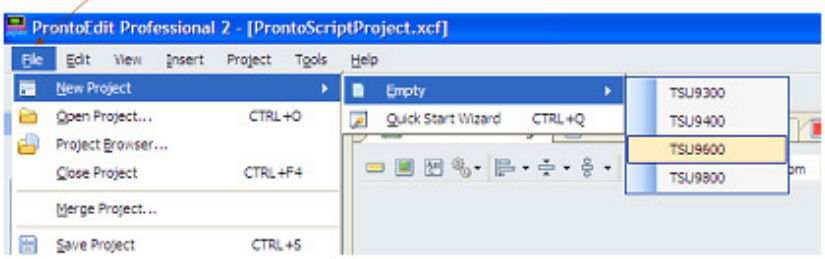
B. ProntoScript client applications development tutorial





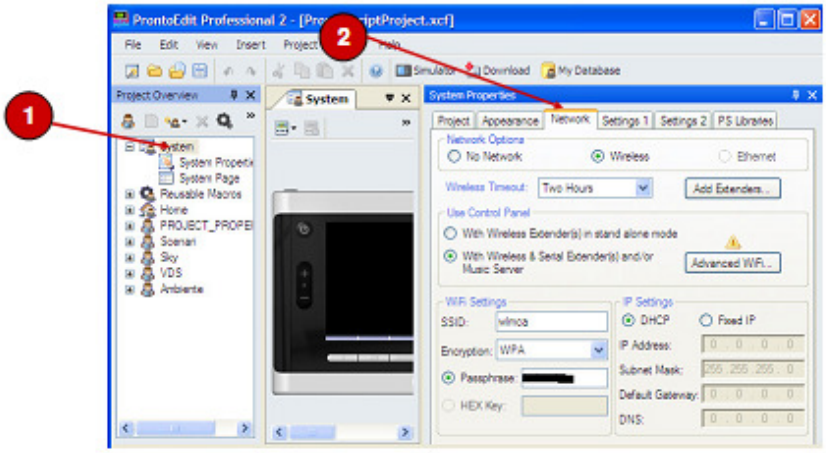
1. Create new project

File > New Project > Empty > TSU9600



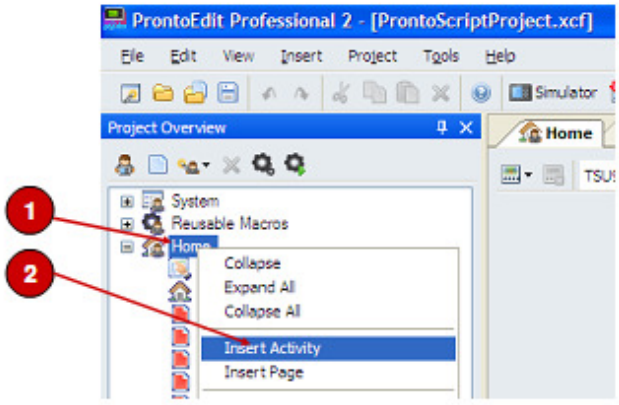
2. Setting wireless info

1. Click on the System activity
2. Choose the Network tab



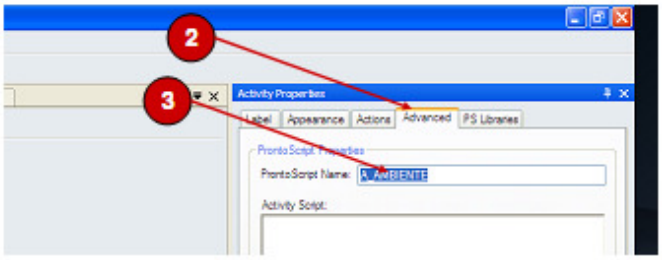
3. Insert an activity

1. Right click in Home
2. Choose Insert Activity



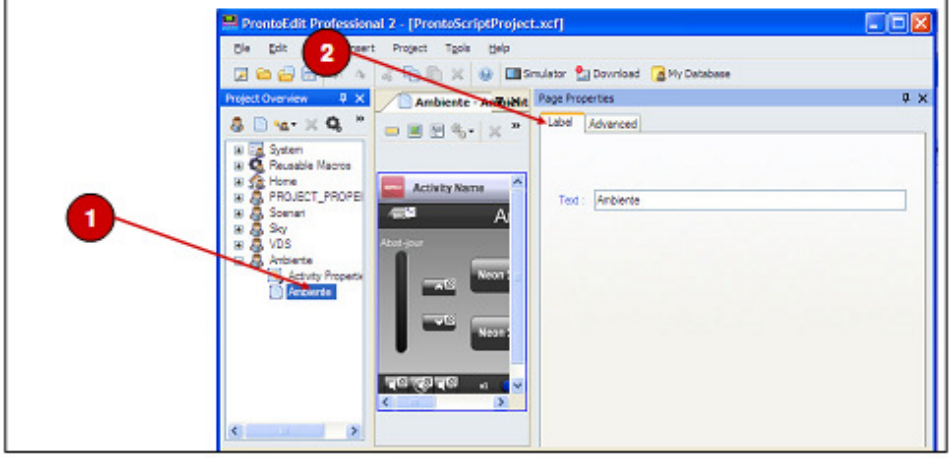
4. Name the activity

1. Name the activity as Ambiente (Click New Activity -> Label)
2. At the Activity Properties View click Advanced tab
3. Name the ProntoScript Name as A_AMBIENTE



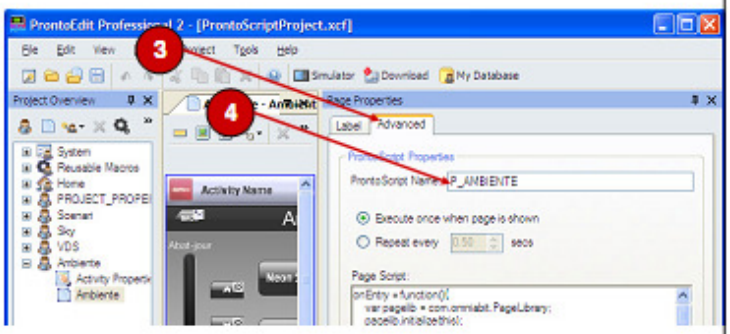
5. Name the page

1. Rename the page to Ambiente
2. Label the page as Ambiente



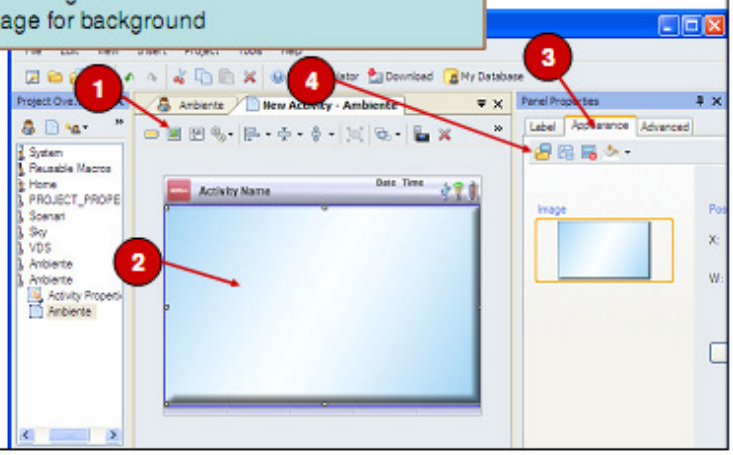
5. Name the page

3. Click the Advanced tab
4. Rename the ProntoScript to P_AMBIENTE



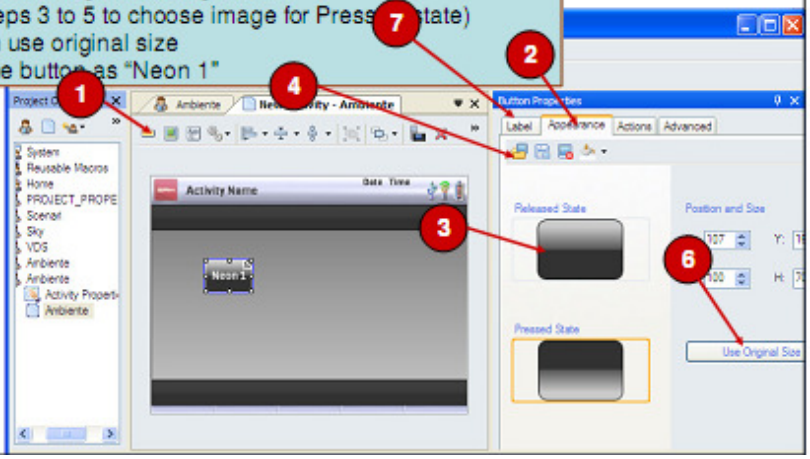
6. Add The background panel

1. Click the panel icon to add a panel
2. Resize the panel to fit the background
3. Choose the Appearance tab in Panel Properties
4. Click on the Add Image icon
5. Choose the image for background



7. Neon 1 button

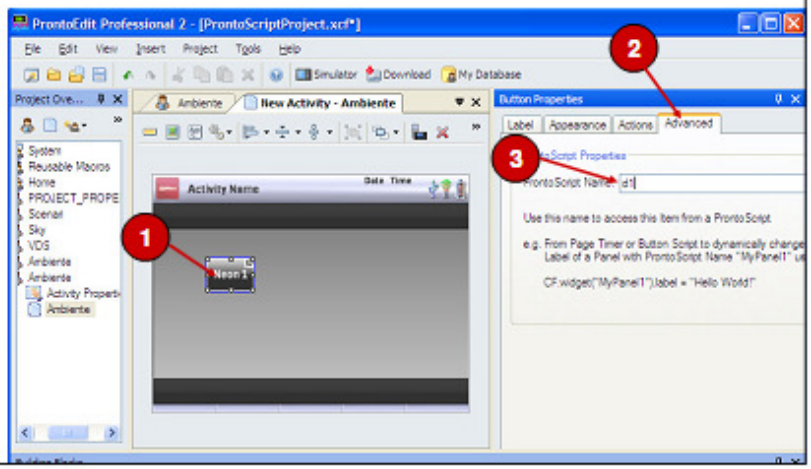
1. Click the button icon to add a button
2. Choose the Appearance tab in Panel Properties
3. Click on the Release State image
4. Click on the Add Image icon
5. Choose the image for background
(Follow steps 3 to 5 to choose image for Press[illegible]tate)
6. Click on use original size
7. Label the butto[illegible]as "Neon 1"



8. Name the button

1. Click on the button
2. Click Advanced tab
3. Rename the ProntoScript name as digital join bound to it (d1)

(Further explanation about naming will be in next page)



9. Naming the button

- In general if we are binding a button to digital join, name the button as that digital join.
- For instance, when a button is bound to digital join1 we name it as “d1”
- This will help the library when it gets a message from server, it will know which button (based on the join) to update the view.
- Similar way of naming is true for other panels, such as serial join for panel, analog join for gauging control, etc.

10. Install the library

- If you haven't done so, copy the com.omniabit.PageLibrary.js to the ProntoPhillips library folder
- Normally located in:

(For WinXP)

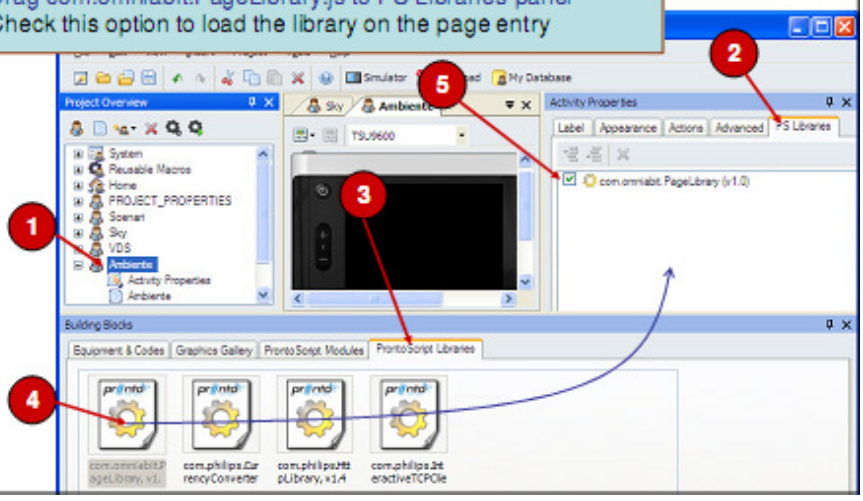
```
C:\Documents and Settings\All Users\Dati applicazioni\Philips\ProntoEdit
   Professional 2\Libraries
```

(For Vista or Win 7)

```
C:\ProgramData\Philips\ProntoEdit  Profes-sional 2\Libraries
```

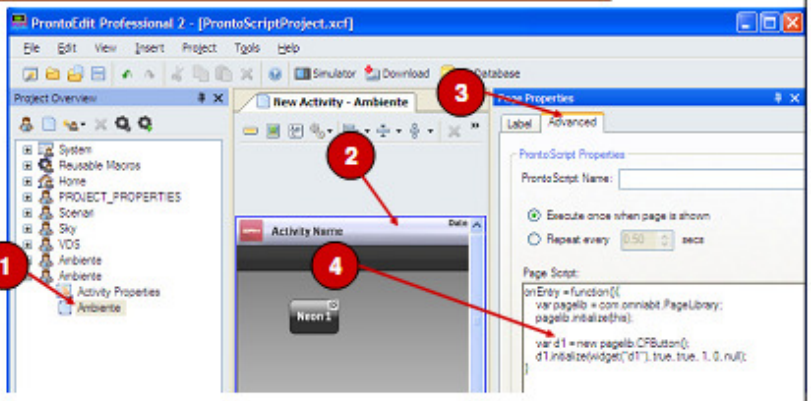
11. Adding the library to page

1. Click on Ambiente activity
2. Click PS Libraries
3. Click ProntoScript Libraries tab
4. Drag com.omniabit.PageLibrary.js to PS Libraries panel
5. Check this option to load the library on the page entry



12. Adding Code

1. Click on Ambiente Page
2. Click page title bar
3. Click Advanced Tab
4. Type the code (will be explained in next page)



13. Code Explanation

- onEntry
  - This function will be called at every Entry of the page
- var pagelib = com.omniabit.PageLibrary;
  - To have a reference to the added PageLibrary
- pagelib.initialize(this);
  - Initialize the page library with "this" which is the current page
- var d1 = new pagelib.CFButton();
  - Declare a new button named as "d1" (the digital join bound to this button)
- d1.initialize(widget("d1"), true, true, 1, 0, null);
  - Initialize buttons with parameters as in CommandFusion button properties
  - The properties will be explained in next page

14. Button Properties

- btn_initialize(ipbutton, simulatefeedback, syncstates, value, state, releaseValue)
- Example: d1.initialize(widget("d1"), true, true, 1, 0, null);
  - Argument 1 is the reference to the button, e.g., button "d1" which is widget("d1")
  - Argument 2 is the simulate feedback value of the button
    - If set to true, the button will toggle its appearance without waiting for feedback signal from the server (for the sync value)
    - If set to false, the button will be back to its normal state when user release
  - Argument 3: sync state, if set to true, the value coming from server (feedback message, sync message) will be parsed and toggle the state of the button correspondingly
  - Argument 4: Value of the button, which is the value which will be sent when the button is click
    - It can be 1, e.g., when "d1" is clicked a message with value 1, as "d1=1\x03" will be sent to server
    - It can be also 0, in that case the message is "d1=0\x03"\
  - Argument 5: Sate, is the state of the button, can be active(1) or inactive(0).
    - If it sync state is true this value should set to 0 as default, because as the communication protocol of CommandFusion, some joins with value 0, will not be sent with initialize message. So we set the default to 0, if something different from 0, then the initialize message is received and the value will be updated automatically
    - Its normal state, should be 1 (it is active)
  - Argument 6: releaseValue, this value if is not null the button will have a handler and on release of the button a message with that value will be sent to the sever.
    - E.g., if releaseValue = 0, when the button is released, a message "d1=0\x03" will be sent to server

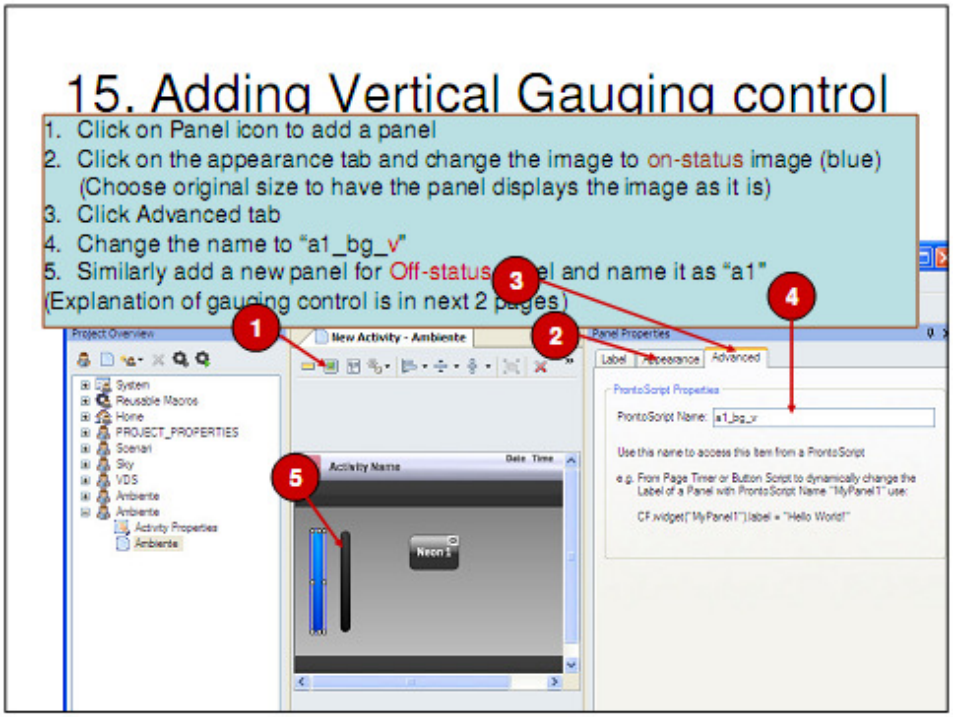
Testing

- If you would like now to test only one button
- You can jump to the testing part at the end
- Otherwise, you can add some more controls to your Program.

15. Adding Vertical Gauging control

1. Click on Panel icon to add a panel
2. Click on the appearance tab and change the image to on-status image (blue) (Choose original size to have the panel displays the image as it is)
3. Click Advanced tab
4. Change the name to "a1_bg_v"
5. Similarly add a new panel for Off-status [illegible]el and name it as "a1"

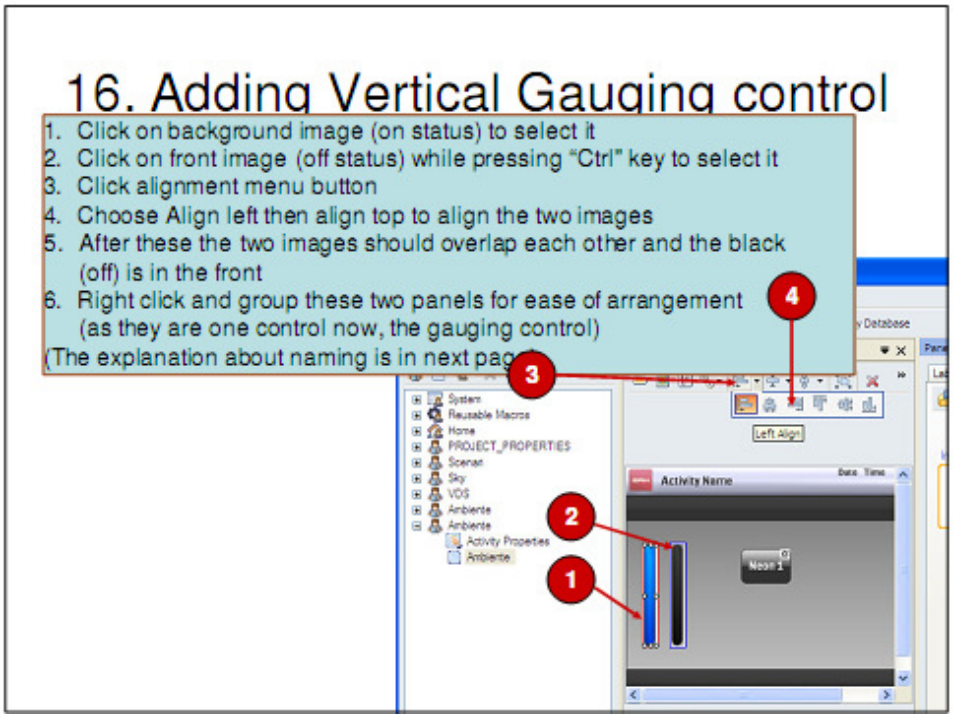
(Explanation of gauging control is in next 2 pages)



16. Adding Vertical Gauging control

1. Click on background image (on status) to select it
2. Click on front image (off status) while pressing "Ctrl" key to select it
3. Click alignment menu button
4. Choose Align left then align top to align the two images
5. After these the two images should overlap each other and the black (off) is in the front
6. Right click and group these two panels for ease of arrangement (as they are one control now, the gauging control)

(The explanation about naming is in next pag[illegible])

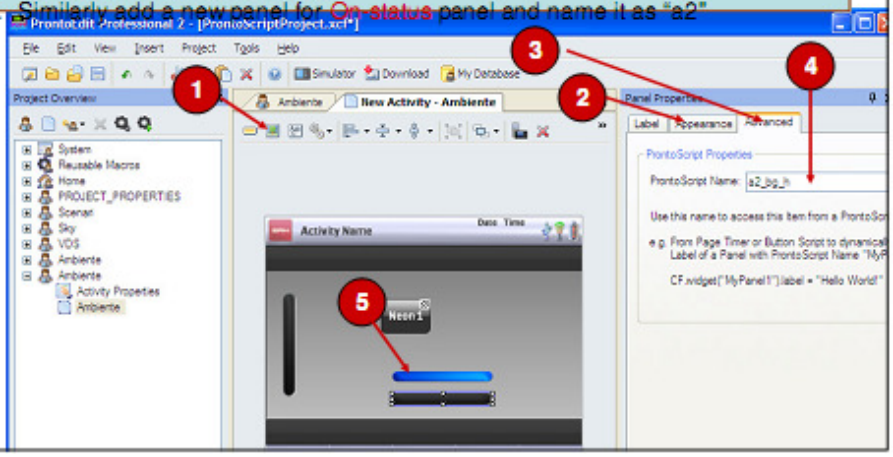


17. Vertical Gauging control

- Vertical gauging control is made of two images
- The first image (which indicates the on status – blue one) is used as background image of the control and is named as the join number we would like to bind and "*_bg_v*"
  - One example is "a1_bg_v"
    - "a1" indicates that this gauging control is bound to analog join "1".
    - "bg" means this image is used as the background;
    - "v" means this gauging control is vertical one
- The second image (which indicates the off status – black one) is used as front image of the control and is named as the join number.
  - One example is "a1" meaning this gauging control is bound to analog join 1.

18. Horizontal Gauging Control

1. Click on Panel icon to add a panel
2. Click on the appearance tab and change the image to off status image (black) (Click the "Use original size" button to have the panel displayed image as it is)
3. Click Advanced tab
4. Change the name to "a2_bg_h"
5. Similarly add a new panel for On-status panel and name it as "a2"



18. Horizontal Gauging Control

- Similar to step 16
- Select the background image, then select the front image while pressing “Ctrl” key, then align, and group them.
- The explanation of the naming is in next page.

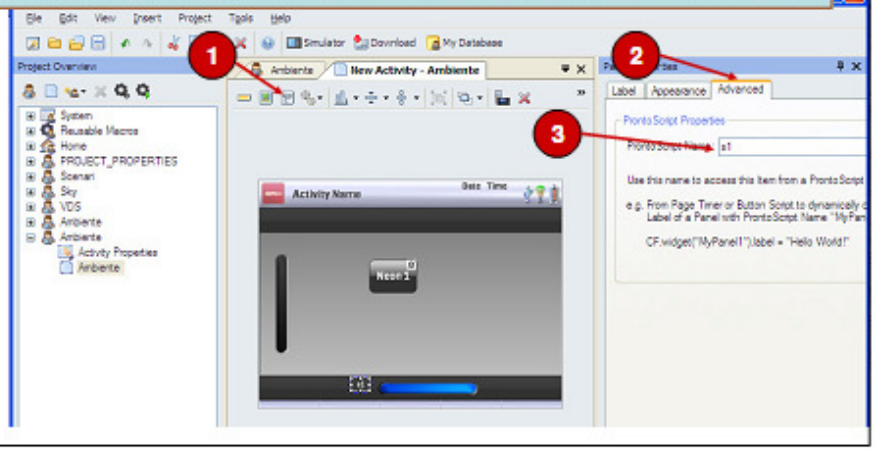
18. Horizontal Gauging Control

- Horizontal gauging control is made of two images
- Its behavior and naming, due to JavaScript image clipping properties is little different.
  - This time we use the on-image (blue) as the front image, and the off-image (black) as the background image, the inverse of the vertical control
- But always, the name of the front image is the analog join which is bound to the control, and the background image is named as that analog join plus “_bg_h”.
  - One example is “a2_bg_h”
    - “a2” indicates that this gauging control is bound to analog join “2”.
    - “bg” means this image is used as the background;
    - “h” means this gauging control is horizontal one
- The second image (which indicates the on status – blue one) is used as front image of the control and is named as the join number.
  - One example is “a2” meaning this gauging control is bound to analog join 2.

19. Adding a serial join

1. Click on Text icon to add a text field
2. Click Advanced tab
3. Change the name to "s1"

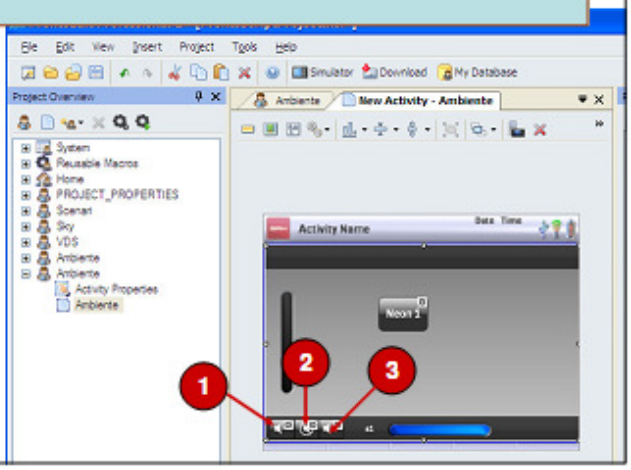
(Just name the Text field with the serial join you would like to display the value, the PageLibrary will know how to parse the value and display to this text field)



19. Adding buttons for sound

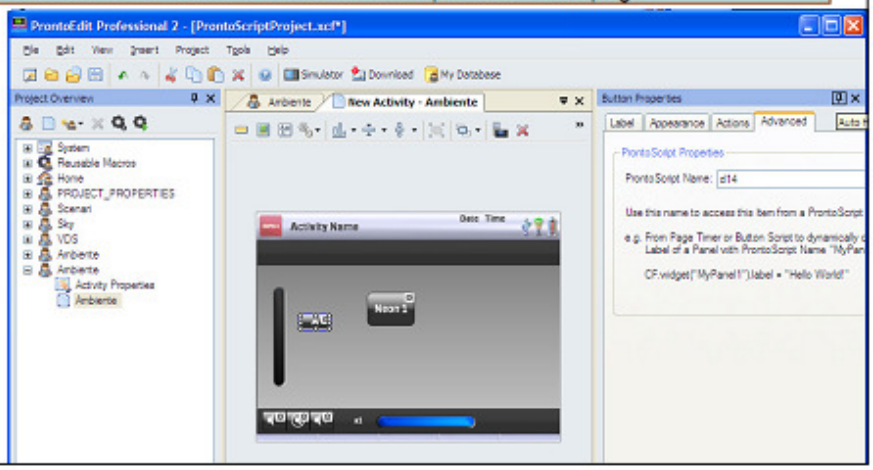
Following the steps in adding button for Neon 1, we add 3 buttons for sound control

- Volume down (name it d6)
- Volume on/off (name it d7)
- Volume up (name it d8)



20. Adding Button with release value

1. Following similar steps in creating button Neon 1 to create a button for Dim-up
2. Name the button as d14 (bind it to digital join 14).
3. The initialization code for this button is explained in next page



20. Adding Button with release value

- For some buttons such as the dim-up, dim-down buttons
  - Besides sending a command when the button is pressed
  - A command will be sent when it is released too
  - So the argument 6 of the button initialize method should have a value (not null but 0 for instance)
  - The following button is one example
  - d14.initialize(widget("d14"), false, true, 1, 1, 0);
    - Button d14 when is pressed will send "d14=1\x03"
    - On released will send "d14=0\x03"

21. Page code modification

- We can code the buttons for sound as the way we coded for Neon 1.
- The difference is that the sound buttons appear in all the pages. Therefore, we put the code in the PageLib library as a string. And "eval" the string when we would like the code to be executed:
- Now we modify the page code in step 12. Add the following line in the body of the onEntry function:
  - eval(pagelib.soundInitString);
  - This line of code initializes the code for sound buttons (d6, d7, d8) so we don't have to code for them anymore.
- Furthermore, the sound buttons, gauging controls, and the neon button are synchronized with server state. Therefore, at the startup of the page, after initialize the buttons, we need to request for initial states, so we add also the following line
  - eval(pagelib.pageStateString);
  - The pageStateString, is the string representation of JavaScript code which will use a socket and send initial request message "i=1x\03" to request for initial states of the server.
  - This code will be repeatedly used in many pages, so we also put it as a string in the library and "eval" this string in every page to execute it.
- The complete code for page script now should be as in next page

22. Complete code of page script

```
onEntry = function(){
    var pagelib = com.omniabit.PageLibrary;
    //Initialize PageLibrary with this page
    pagelib.initialize(this);
    //execute the code for initializing sound buttons
    eval(pagelib.soundInitString);
    //Button for neon1
    var d1 = new pagelib.CFButton();
    d1.initialize(widget("d1"), true, true, 1, 0, null);
    //buttons for dim-up
    var d14 = new pagelib.CFButton();
    d14.initialize(widget("d14"), false, true, 1, 1, 0);
    //Ask for initial informations about buttons states.
    eval(pagelib.pageStateString);
}
```

23. Adding other buttons

- To add other buttons in a series, we can use an excel as following to ease the coding job
- Double click on the object, to copy the coding formula
- Then copy the code to the body of your onEntry function in every page
- The example is used to make code for d21 to d25

37	d	var d37 = new pagelib.CFButton();	d37.initialize(widget('d37'), false, false, 1, 1, null);
38	d	var d38 = new pagelib.CFButton();	d38.initialize(widget('d38'), false, false, 1, 1, null);
52	d	var d52 = new pagelib.CFButton();	d52.initialize(widget('d52'), false, false, 1, 1, null);
53	d	var d53 = new pagelib.CFButton();	d53.initialize(widget('d53'), false, false, 1, 1, null);
54	d	var d54 = new pagelib.CFButton();	d54.initialize(widget('d54'), false, false, 1, 1, null);
14			

Coding

For most of other pages, you should design it using the controls mentioned (such as buttons, horizontal/vertical gauging controls, text all have done in previous step.

The code of every page should be the same as following:

```
onEntry = function(){
    var pagelib = com.omniabit.PageLibrary;
    //Initialize PageLibrary with this page
    pagelib.initialize(this);
    //execute the code for initializing sound buttons
    eval(pagelib.soundInitString);

    //************************************************
    //    Area for your other buttons' initializations
    //************************************************

    //Ask for initial informations about buttons states.
    eval(pagelib.pageStateString);
}
```

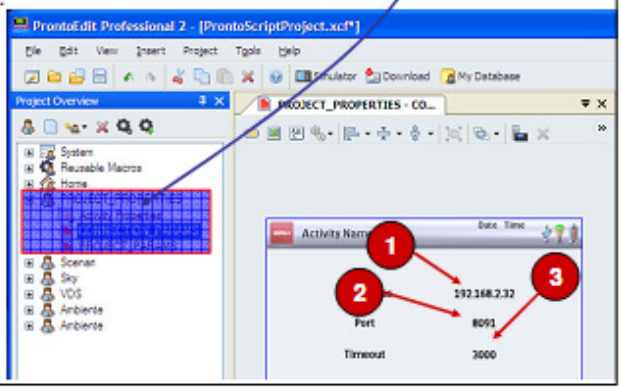
The attached project

The attach project has an project properties activities which has invisible pages which is used to store configuration data such as

1. IP address of the CommandFusion server
2. Port number
3. And connection timeout.

PageLibrary use these configuration information here to create a socket connection to CommandFusion server.

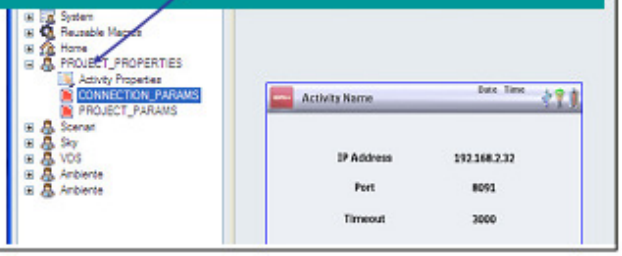
You can change these values before downloading the code to ProntoPhillips device to reflect your configuration.



Testing – Project properties

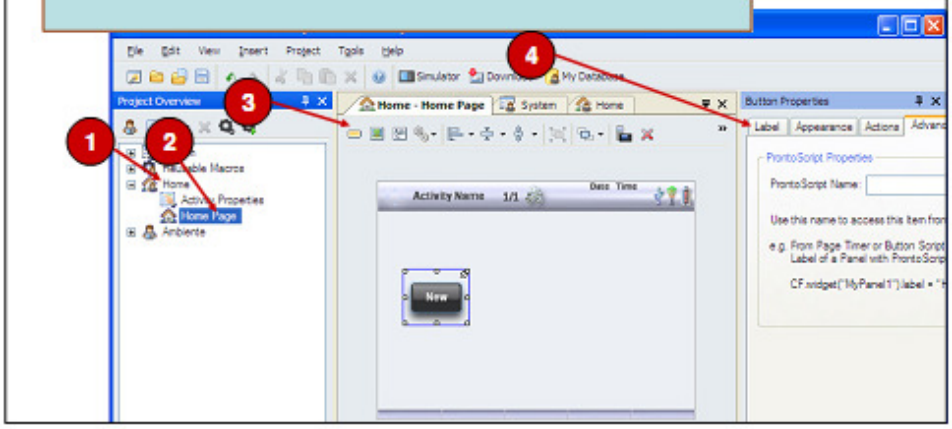
The following steps are must. Since the PageLibrary uses information from these hidden page to create connection to CommandFusion command handler

1. Open the Attached project and copy the PROJECT_PROPERTIES activity to your project.
2. This project contains the configuration information about the CommandFusion command handler server (with IP, and Port) in CONNECTION_PARAMS
3. It also contains PROJECT_PARAMS which you can enable/disable debugging feature (printing or not the command you will send when a button is clicked)



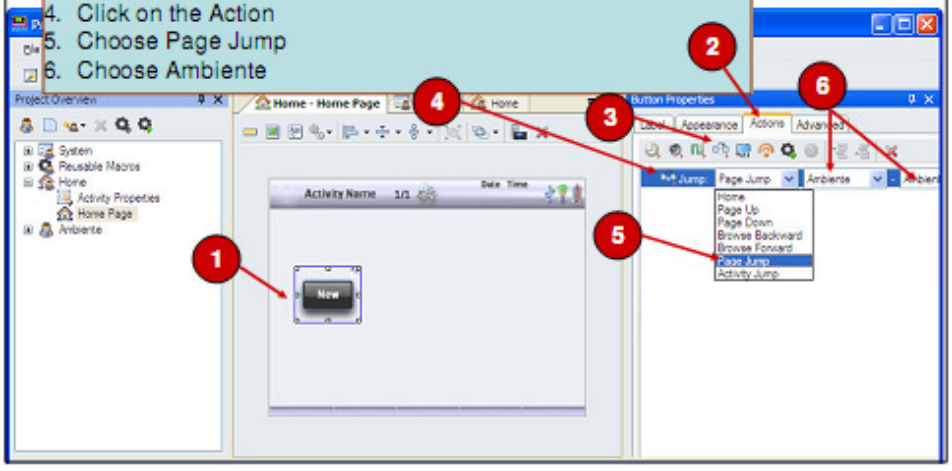
Testing -Create a link from Homepage

1. Click on Home activity
2. Choose Home Page
3. Click Button icon
4. Choose Actions tab



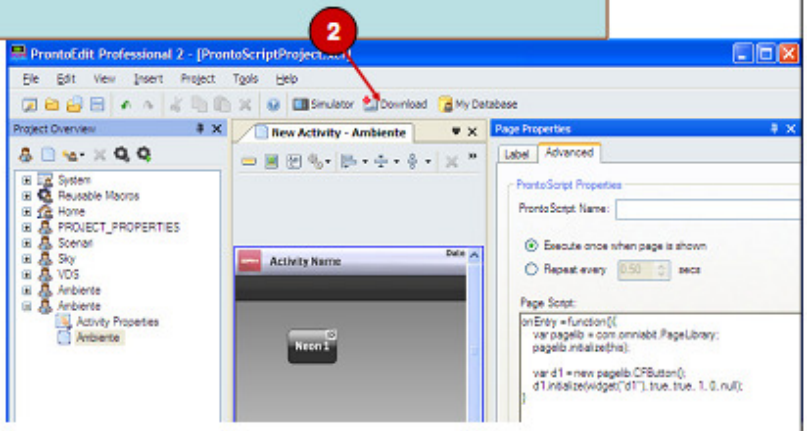
Testing - Create a link from Homepage

1. Select the button
2. Choose Actions tab
3. Click "Add Jump Action" icon
4. Click on the Action
5. Choose Page Jump
6. Choose Ambiente



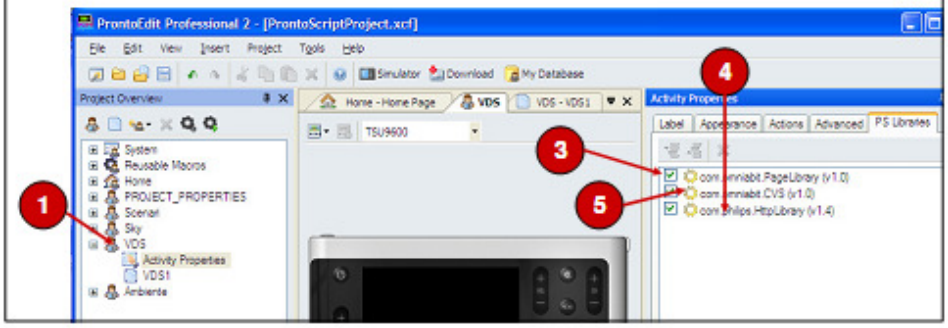
Testing the button

1. Connect your ProntoPhillips device
2. Click the "Download" button
3. Test the button from ProntoPhillips device



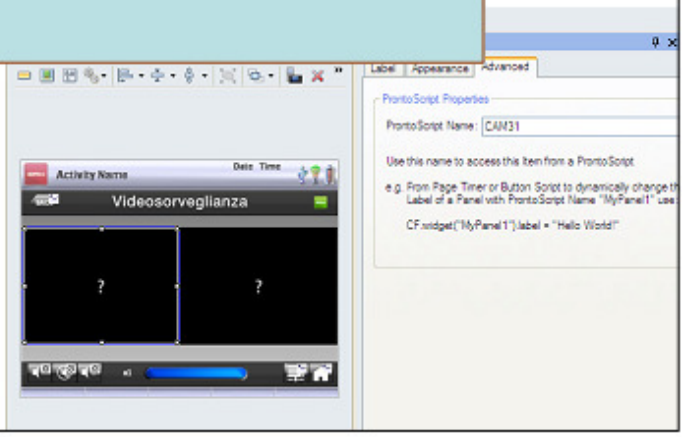
Activity for Camera Surveillance

1. Add an activity for VDS
2. Install the (com.omniabit.CVS) lib if you haven't done so
3. Add PageLibrary for the behaviors of CF buttons, labels, etc
4. Add com.philips.HttpLibrary lib for the Http communication
5. Add com.omniabit.CVS lib to get image from server to a panel



Activity for Camera Surveillance

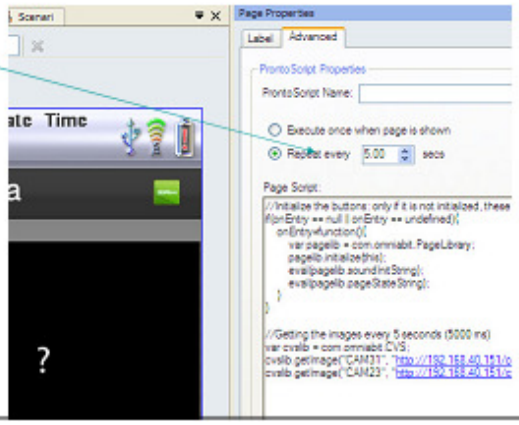
1. Add two panels and browse for their default image (?)
2. Name the panel (CAM31, and CAM23)



Code to get image from a camera

```
//Getting the images every 5 seconds (5000 ms)
var cvslib = com.omniabit.CVS;
cvslib.getImage(widgetName, cameraURL, username, password);
```

Remember to choose the option "Repeat every…5 s" so the image is refreshed every 5s



Code explanation

- First part of the code is the initialization for buttons as we have done for other pages
- The only difference is now we put a conditional check if we have created the onEntry function or not since this code is executed every 5s. We should check to avoid reinitializing the buttons all the time.
- The next part is:
  - First we create a reference to the cvs library:
    var cvslib = com.omniabit.CVS;
  - For each camera make a call as:
    cvslib.getImage("CAM23", "URL", "admin", "berta1");
    - The first argument is the name of the Camera panel
    - The second is the URL to get the image for the camera
    - The third and fourth are username and password to access to camera

Complete Code

```
//Initialize the buttons: only if it is not initialized, these lines are for
teh sound buttons, and their initial states
if(onEntry == null || onEntry == undefined){
    onEntry=function(){
        var pagelib = com.omniabit.PageLibrary;
        pagelib.initialize(this);
        eval(pagelib.soundInitString);
        eval(pagelib.pageStateString);
    }
}

//Getting the images every 5 seconds (5000 ms)
var cvslib = com.omniabit.CVS;
cvslib.getImage("CAM31", "http://192.168.[redacted]/cgi-
bin/mini/image?cam=31&size=318x238", "admin", "[redacted]");

cvslib.getImage("CAM23", "http://192.168.[redacted]/cgi-
bin/mini/image?cam=23&size=318x238", "admin", "[redacted]");
```