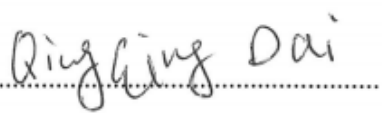




## FACULTY OF SCIENCE AND TECHNOLOGY

### BACHELOR THESIS

|  |  |
|--|--|
| Study program / specialization:<br>Computer Science  | Spring semester, 2021<br>Open  |
| Author:<br>Qingqing Dai  |  |
| Supervisor (s):<br>Chunming Rong; Torstein Thingnæs(NOV)   |  |
| Title of The Bachelor Thesis:<br>Build a Dashboard Application for NOVs eVolve Automation System |  |
| Credits: 20  |  |
| Keywords:  | Number of pages: 81<br>+ Appendix / other: 16<br>Stavanger, 15/07/2021               |

# **Build a Dashboard Application for NOVs eVolve Automation System**

Qingqing Dai

Department of Electrical Engineering and  
Computer Science, University of Stavanger

July 2021

## **Abstract**

---

The project is completed by cooperating with NOV. The purpose of the project is to make a dashboard application that gathers the essential information, monitors the running status of the devices, and presents an overview of the NOV eVolve Automation System.

NOV eVolve Automation System is a system that includes hardware and software to enable the integration between downhole tools and surface automation systems through Intelliserve Wired DrillPipe. Intelliserve Wired Drillpipe is a solution that enables high-speed data connection with sensors in the hole while drilling.

The goal is to make a dashboard application that shows hardware status, network communication status between the computers, and remote desktop connection from a Windows system machine to remote computers. The dashboard application will be installed and run on a Windows 10 machine, called a Wired Drill Pipe terminal (WDP terminal). C# is an object-oriented and component-oriented programming language studied by the author to build the dashboard application. Chapter 2 presents an introduction of C# and the history and features of the C# programming language.

Chapter 3 presents technology and methodology with the source code. Chapter 4 uses graphics to explain the relationships between the individual event handler and the components on the user interface of the dashboard application. There are two URL links attached in chapter 5. One links to a video for a demonstration of the dashboard application, and the other URL link points to the GitHub repository where all the source code files store.

Chapter 6 gives a conclusion for the project and provides suggestions for further development. The author independently writes fourteen pages of source code to implement the NOV

## *Build a Dashboard Application for NOVs eVolve Automation System*

Dashboard application. Besides, the author finishes the bachelor thesis on her own, which is 97 pages in total.

The completed dashboard application achieves all the goals above and is confirmed by NOV. Besides, the dashboard application can detect the offline time and the duration of the devices' downtime. Moreover, the author adds an extra function to the dashboard application to establish a remote connection with any of the devices over the network communication and builds a JSON file that stores information for later investigation.

Running the completed dashboard application on the WDP terminal helps the NOV system engineers diagnose and resolve the potential problems once there are issues delivered from the offshore rig. Saving the time to identify the issue is the most significant benefit of using the dashboard application, especially when the rig is at downtime by accident.

## Table of Contents

---

|  |             |
|--|-------------|
| <b>ABSTRACT</b> .....  | <b>III</b>  |
| <b>TABLE OF CONTENTS</b> .....   | <b>V</b>    |
| <b>LIST OF ABBREVIATIONS</b> .....                                       | <b>VIII</b> |
| <b>LIST OF FIGURES</b> .....   | <b>IX</b>   |
| <b>LIST OF TABLES</b> .....  | <b>XI</b>   |
| <b>CHAPTER 1: INTRODUCTION</b> .....                                     | <b>12</b>   |
| 1.1 BACKGROUND AND PROJECT DESCRIPTION .....                             | 12          |
| 1.1.1 Overview of the NOV eVolve Automation System.....                  | 12          |
| 1.1.2 NOV Industrial Server .....  | 12          |
| 1.1.3 Graphical Overview of Data Flow .....                              | 13          |
| 1.1.4 Project Description.....   | 14          |
| 1.2 AIMS .....   | 15          |
| 1.2.1 The Goals of the Project: .....                                    | 15          |
| 1.3 MOTIVATION.....  | 16          |
| 1.3.1 Demand on the Notification System. ....                            | 16          |
| 1.3.2 Save the Time to Identify Issues .....                             | 16          |
| 1.3.3 Detecting the Issues Automatically.....                            | 16          |
| <b>CHAPTER 2: TECHNOLOGY AND THEORY</b> .....                            | <b>17</b>   |
| 2.1 INTRODUCTION .....   | 17          |
| 2.2 THE STUDY OF C#.....   | 17          |
| 2.2.1 Learning Material for Studying the C# Programming .....            | 17          |
| 2.2.2 The Features of C# .....   | 18          |
| 2.2.3 The History and Development of C#.....                             | 19          |
| 2.2.4 The Scope of Application and Usage for C#.....                     | 21          |
| 2.2.5 The Comparison Between C# and Java .....                           | 22          |
| 2.2.5.1 The similarities between C # and Java .....                      | 22          |
| 2.2.5.2 The differences Between Java and C#.....                         | 22          |
| 2.2.6 The Decision Made to Use C# Instead of Java for This project. .... | 24          |
| 2.3 VS 2019 IDE - PROGRAMMING SOFTWARE FOR WINDOWS .....                 | 25          |
| 2.3.1 The VS2019 IDE .....   | 25          |
| 2.3.2 Creating a Windows Forms app in VS 2019 with C# .....              | 26          |

|  |   |           |
|--|---|-----------|
| 2.4  | THEORY FOR BUILDING DASHBOARD.....                                    | 28        |
| 2.4.1  | <i>The Graphical Composition of the Dashboard.....</i>                | 28        |
| 2.4.2  | <i>The Graphical layout and the Functions of the Dashboard.....</i>   | 28        |
| 2.5  | SUMMARY OF CHAPTER 2 .....  | 31        |
| <b>CHAPTER 3: IMPLEMENTATION .....</b>             |   | <b>32</b> |
| 3.1  | INTRODUCTION .....  | 32        |
| 3.2  | CREATE FORM1.CS.....  | 32        |
| 3.2.1  | <i>Introduction of Form1.Designer.cs.....</i>                         | 32        |
| 3.2.2  | <i>Create a Form1 Class in Form1.cs.....</i>                          | 35        |
| 3.2.3  | <i>Create a Class PCInfo in Form1.cs.....</i>                         | 36        |
| 3.2.3.1  | The Constructor and Properties in PCInfo Class .....                  | 36        |
| 3.3  | CREATE ADDPC.CS.....  | 41        |
| 3.3.1  | <i>Introduction of AddPC.Designer.cs.....</i>                         | 42        |
| 3.3.2  | <i>The AddPC.cs .....</i>   | 43        |
| 3.3.2.1  | Create an Event Handler for Button1 .....                             | 44        |
| 3.3.2.2  | Create an Event Handler for Button2 .....                             | 45        |
| 3.3.2.3  | Setup Button3.....  | 47        |
| 3.4  | CREATE CONNFORM.CS .....  | 50        |
| 3.4.1  | <i>The ConnForm.cs .....</i>  | 51        |
| 3.4.1.1  | Create an Event Handler for Setting Size.....                         | 53        |
| 3.5  | SUMMARY OF CHAPTER 3 .....  | 55        |
| <b>CHAPTER 4: IMPLEMENTATION OF FORM1.CS .....</b> |   | <b>56</b> |
| 4.1  | INTRODUCTION .....  | 56        |
| 4.2  | ADD MORE COMPONENTS TO FORM1 IN DESIGNER.....                         | 56        |
| 4.2.1.1  | Add ContextMenuStrip Component to Form1 .....                         | 56        |
| 4.2.1.2  | Add BackgroundWorker Component to Form1 .....                         | 57        |
| 4.2.1.3  | Add Timer Component to Form1 .....                                    | 57        |
| 4.3  | IMPLEMENTATION OF FORM1 CLASS.....                                    | 57        |
| 4.3.1.1  | Create Constructor for Form1 Class.....                               | 57        |
| 4.4  | INITIALIZE DATAGRIDVIEW1 .....  | 59        |
| 4.5  | CREATE A CLICK EVENT HANDLER TO ADD BUTTON .....                      | 59        |
| 4.6  | ADD A RIGHT-CLICKING CASCADING MENU .....                             | 60        |
| 4.7  | CREATE A CLICK EVENT HANDLER TO EDIT OPTION .....                     | 61        |
| 4.8  | CREATE A CLICK EVENT HANDLER TO DELETE OPTION .....                   | 63        |
| 4.9  | CREATE A CLICK EVENT HANDLER TO CONNECT OPTION .....                  | 63        |
| 4.10   | CREATE A BOOLEAN METHOD TO RETURN THE STATUS OF REMOTE COMPUTER ..... | 64        |
| 4.11   | CREATE A DOWORK EVENT HANDLER TO BACKGROUNDWORKER .....               | 65        |
| 4.12   | CREATE A TICK EVENT HANDLER TO TIMER.....                             | 67        |

*Build a Dashboard Application for NOVs eVolve Automation System*

4.13 CREATE A CELLFORMATTING EVENT HANDLER TO DATAGRIDVIEW ..... 68

4.14 CREATE A FORM.LOAD EVENT HANDLER TO FORM1 ..... 69

4.15 CREATE A FORMCLOSING EVENT HANDLER TO FORM1 ..... 71

4.16 SUMMARY OF CHAPTER 4..... 73

**CHAPTER 5: DEMONSTRATION ..... 74**

5.1 INTRODUCTION ..... 74

5.2 CREATE AN MSI INSTALLER FOR DASHBOARD APPLICATION. .... 74

5.3 THE VIDEO FOR DEMONSTRATION ..... 74

    5.3.1 Modification According to The Comments ..... 74

5.4 THE GITHUB REPOSITORY FOR THE PROJECT..... 77

5.5 SUMMARY OF CHAPTER 5 ..... 77

**CHAPTER 6: CONCLUSION..... 78**

6.1 RESTATE THE THESIS ..... 78

6.2 THE COMPLETED DASHBOARD APPLICATION ..... 80

6.3 THE ADVICE FOR THE FURTHER DEVELOPMENT ..... 81

**REFERENCES..... 82**

**APPENDIX A: ADDPC.CS..... 84**

**APPENDIX B: CONNFORM.CS ..... 86**

**APPENDIX C: FORM1.CS..... 88**

## List of Abbreviations

---

| <b>Abbreviations</b> | <b>Description</b>                                    |
|----------------------|---|
| EFD                  | Equivalent Fluid Density                              |
| LER                  | Local Equipment Room                                  |
| GUI                  | Graphical User Interface                              |
| Stream TV            | Application to interface downhole data from NetCon    |
| VM                   | Virtual Machine                                       |
| WBC                  | Wellbore Connect                                      |
| WDP                  | Wired Drill Pipe                                      |
| VS 2019 IDE          | Visual Studio 2019 Integrated Development Environment |
| CLR                  | Common Language Runtime                               |
| JRE                  | Java Runtime Environment                              |
| API                  | Application Programming Interface                     |
| RDP                  | Remote Desktop Protocol                               |
| TCP                  | Transmission Control Protocol                         |
| RDP                  | Remote Desktop Protocol                               |
| JSON                 | JavaScript Object Notation                            |
| OOP                  | Object-oriented Programming                           |



## List of Figures

---

| <b>Figures</b> | <b>Description</b>                             |
|----------------|--|
| Figure 1       | System overview                                |
| Figure 2       | Surface data flow                              |
| Figure 3       | Downhole data flow                             |
| Figure 4       | The features of C#.                            |
| Figure 5       | Syntax comparison                              |
| Figure 6       | .NET desktop development workload              |
| Figure 7       | Create a new project                           |
| Figure 8       | Form1.cs                                       |
| Figure 9       | Layout of dashboard window 1                   |
| Figure 10      | Add form                                       |
| Figure 13      | An initial Form1.Designer                      |
| Figure 14      | Toolbox  |
| Figure 15      | Form1 designer for Dashboard App               |
| Figure 16      | Properties window                              |
| Figure 17      | Windows Form Designer generated code for Form1 |
| Figure 18      | Class diagram for PCInfo                       |
| Figure 19      | Add new Windows form                           |
| Figure 20      | AddPC form                                     |
| Figure 21      | Password                                       |
| Figure 22      | Button3 DialogResult property                  |
| Figure 23      | Class diagram for AddPC                        |
| Figure 24      | ConnForm designer                              |

|           |  |
|-----------|--|
| Figure 25 | Check Microsoft RDP Client Control – version 9 |
| Figure 26 | Remote Desktop Connection App                  |
| Figure 27 | Class diagram for ConnForm                     |
| Figure 28 | Design Form1                                   |
| Figure 29 | BackgroundWorker                               |
| Figure 30 | Timer  |
| Figure 31 | Main form                                      |
| Figure 32 | Form1 form                                     |
| Figure 33 | Cascading Menu for edit, delete and connect    |
| Figure 34 | Properties of timer1                           |
| Figure 35 | Offline duration                               |
| Figure 36 | Modify the production name and manufacturer    |
| Figure 37 | Add NOV.ico                                    |
| Figure 38 | Modify icon of the shortcut                    |
| Figure 39 | After installaton 1                            |
| Figure 40 | After installation 2                           |

## **List of Tables**

---

| <b>Tables</b> | <b>Description</b>                     |
|---------------|--|
| Table 1       | The Overview for the development of C# |
| Table 2       | The comparison between C# vs Java      |
| Table 3       | Table 3 AddPC                          |
|               |  |
|               |  |
|               |  |
|               |  |

## **Chapter 1: Introduction**

---

### **1.1 Background and Project Description**

#### **1.1.1 Overview of the NOV eVolve Automation System**

The National Oilwell Varco (NOV) eVolve Automation System uses the high-speed telemetry network provided by the IntelliSer wired drill pipe that transfers the downhole to the surface at high speeds up to 57,600 bits per second.

The NOV eVolve Automation System provides real-time measurements from sensors embedded throughout the drill string at regular intervals. These are collar-based tools placed along the string that acquire and transmit high-speed measurements independently of mudflow. Data is then sent to the top drive modified with wired components, including a DataSwivel. Surface cabling along the TopDrive service loop is installed to transfer data from the DataSwivel to the network control system, which is called NetCon.

The functions of NetCon are to receive downhole data and transfers it to third-party vendors, as well as amplifies the signal and convert it to standard ethernet communication.

#### **1.1.2 NOV Industrial Server**

An industrial server is installed in the Local Equipment Room (LER), where the data stream is connected and treated. The *Figure 1 System overview* shows that the NOV industrial server contains a wide range of hardware components, e.g., Stream TV, Wellbore Connect (WBC), NOV Equivalent Fluid Density (EFD), Datavault, and RigSense.

A wired drill-pipe terminal (WDP terminal) connected to the NOV industrial server processes the data correctly via the network. The NOV Industrial server also needs connections to NetCon and other systems.

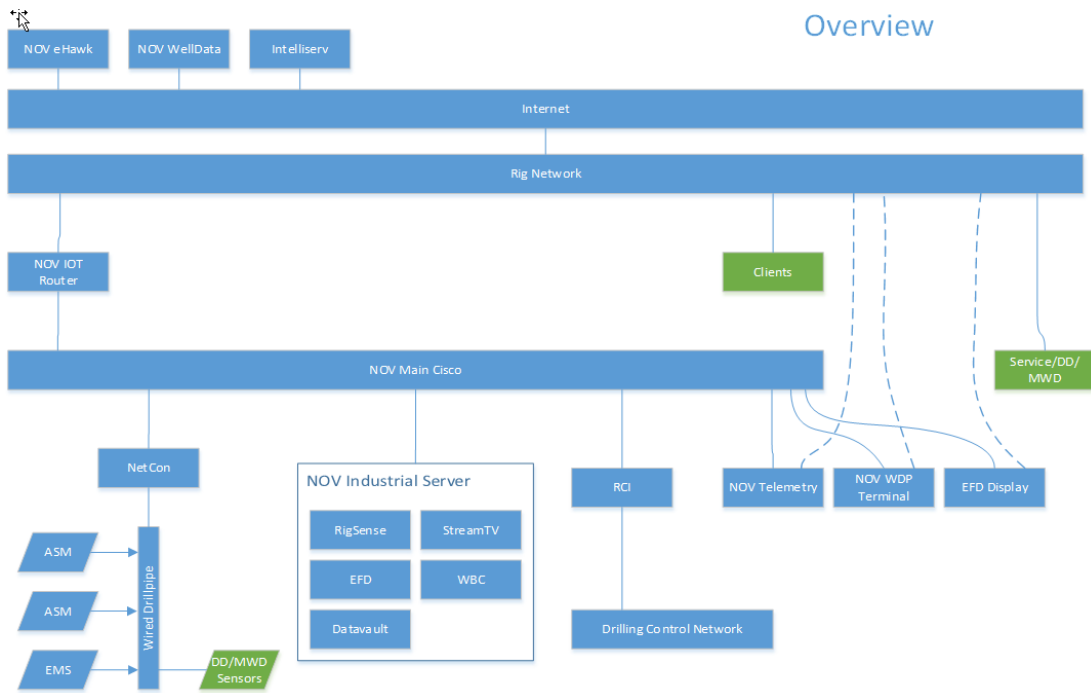


Figure 1 System overview

### 1.1.3 Graphical Overview of Data Flow

Figure 2 Surface data flow shows the Rigsense receives the data, which NetCon already converts, then the data will flow separately to WBC and EFD. To the end, the Stream TV will display the data from WBC.

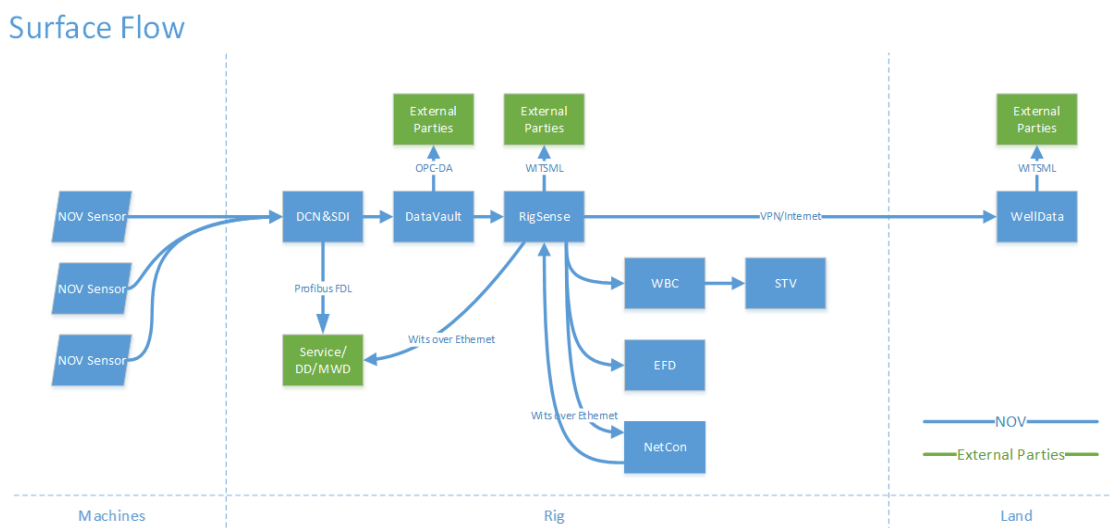


Figure 2 Surface data flow

Figure 3 Downhole data flow shows the data stream from the downhole to the components for NOV industrial server, which is slightly different from the above surface data flow.

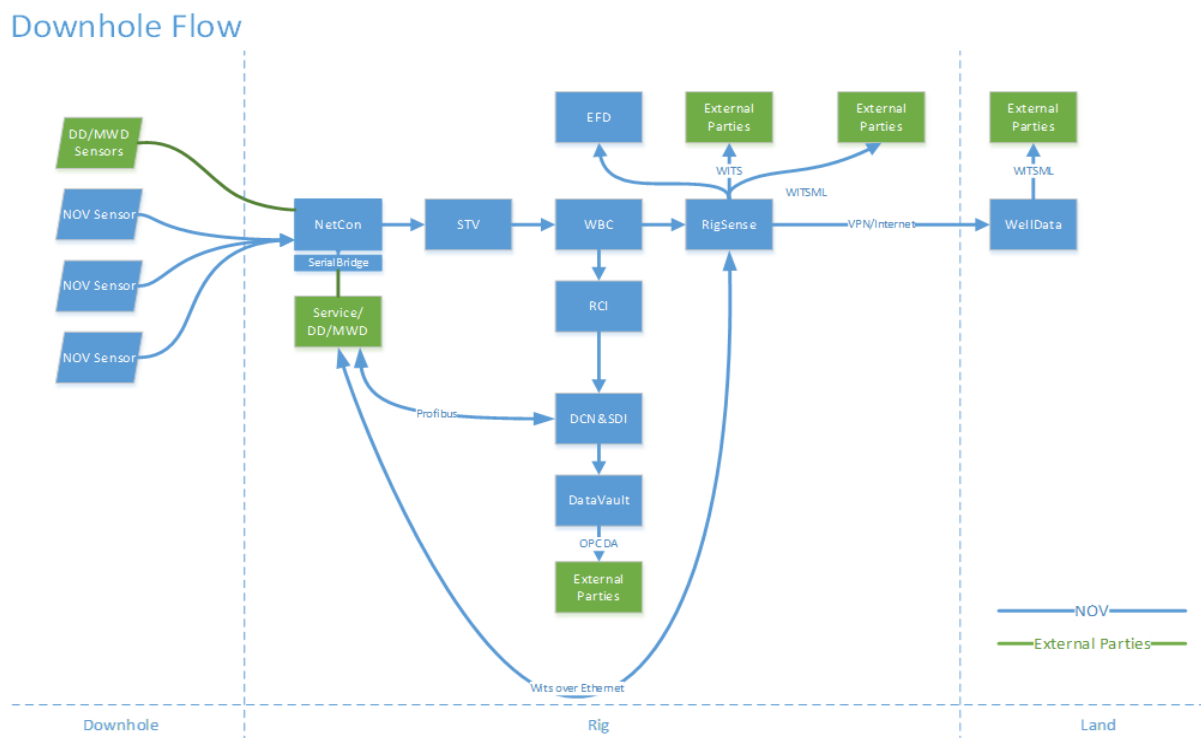


Figure 3 Downhole data flow

### 1.1.4 Project Description

The hardware mentioned above (STV, WBC, EFD, Datavault, and RigSense) are also known as computers. Many of them have their user interfaces in the NOV industrial server. There is, however, no standard dashboard or overall system graphical user interface to monitor the entire system.

The dashboard application provides an overview of each computer's status, and the system will use those statuses to monitor the network communications between these computers. Furthermore, the dashboard application also monitors the downtime duration of each computer.

The dashboard application will be installed on the WDP terminal, a physical machine shipped to the offshore rig, and it has access to all the computers as mentioned above. That means it is

possible to remotely control any of the computers in the NOV Industrial server by logging on to the WDP terminal.

The thesis presents the process of building the dashboard application. Chapter 2 primarily introduces the technology and methodology to make the dashboard application. In other words, chapter 2 describes the theoretical concept for achieving the goal in detail.

Chapter 3 and chapter 4 mainly present the code in the separate source code file. Chapter 3 presents the specific process for how the dashboard application is created on VS 2019 and produces the foundational classes invoked in chapter 4. Chapter 4 analyzes the code for event handlers and components that implement and functionalize the dashboard application.

Chapter 5 mainly demonstrates the completed dashboard application and lists some necessary modifications according to the NOV employees' suggestions to improve the dashboard application. Chapter 6 gives a conclusion and some pieces of advice about further development.

## **1.2 Aims**

### **1.2.1 The Goals of the Project:**

1. To study programming language C# and create a graphical user interface for a dashboard C# application using the Visual Studio 2019 integrated software development environment.
2. To present an overview of the running status of the computers and software, e.g., WBC, RigSense, EFD, and STV mentioned in the NOV industrial server section.
3. To complete a dashboard application for monitoring the status of the network communications between the above computers.
4. To establish the remote control using the dashboard application to other computers with the Remote Desktop Connection App embedded.

5. To give more ideas and suggestions to multi-functionalize the dashboard application in the future.

## **1.3 Motivation**

### **1.3.1 Demand on the Notification System.**

Today, several applications, which are running on various computers, monitor the NOV eVolve Automation System. However, there is no standard notification system. Therefore, it becomes a high demand to have an application program to sketch an overview of the whole system.

### **1.3.2 Save the Time to Identify Issues**

The NOV system engineer group often receives phone calls from the offshore rig asking for support when there is an issue. Still, they cannot provide enough information for the NOV system engineer to diagnose the cause of the error. The dashboard helps to find the specific device where the issue occurred and fix the problem quickly.

### **1.3.3 Detecting the Issues Automatically**

The offshore rig is working in shift, and there are a lot of crew working and using the same computer or device. Usually, the crew is well-trained to have the professional abilities to make sure that they will perform their work task accurately. However, there will always be accidents, e.g., someone plugs out the power supply accidentally. The dashboard application can quickly identify such accidents since it can detect the running status of the computer.



## **Chapter 2: Technology and Theory**

---

### **2.1 Introduction**

Chapter 2 begins with a brief introduction to the programming language C#. The VS 2019 IDE programming platform is used for compiling code, building, and developing the dashboard application. The technologies, functions, and tools created and designed to meet the project's needs are also presented subsequently in the later sections.

### **2.2 The Study of C#**

C# pronounces “see sharp”. It is a programming language. This section will bring readers a tour of knowing this programming language, starting with introducing the features of language C# and the graphical explanation for a good understanding and the history, development, and scope for the usage of the C#. Besides, there is a comparison between C# and the other programming language at the end of the section, e.g., Java.

#### **2.2.1 Learning Material for Studying the C# Programming**

The learning materials are :

1. Book: Beginning C# 6.0 Programming with Visual Studio 2015,  
Author: Benjamin Perkins, Jacob Vibe Hammer, Jon D. Reid
2. C# Tutorial  
Online source: W3Schools  
URL: <https://www.w3schools.com/cs/>
3. C# Documentation  
Online source: Microsoft Documentation  
URL: <https://docs.microsoft.com/en-us/dotnet/csharp/>

### **2.2.2 The Features of C#**

In this day and age, C# became one of the most popular programming languages. It is an open-source language that encompasses static typing, strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented, and component-oriented programming disciplines (International, 2017).

One of the object-oriented language features is providing a clear structure to the program that makes it possible to recycle the code in separate programs, significantly reducing the cost and saving time for compiling.

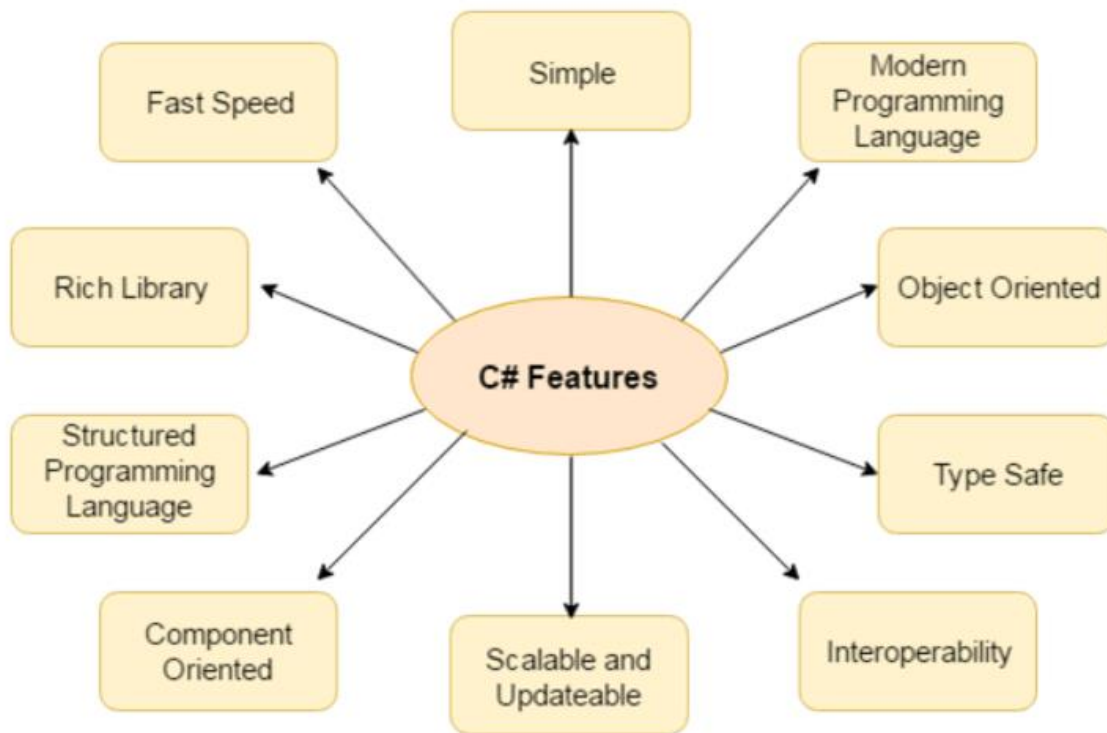
Besides, C# also integrates the following features that are very helpful to create robust and durable applications:

1. Garbage collection.
  - It automatically regains the memory space occupied by the object that is not in use.
2. Nullable reference types.
  - It dereferences variables to allocated objects.
3. Exceptions and exception handling.
  - It helps to deal with the error caught while running a program.
4. Lambda expressions.
  - It creates anonymous functions.
5. Language-Integrated Query (LINQ).
  - It is an integration of query language embedded into the C# language.
6. Support for asynchronous operations
  - It enables the code to read in a sequence but run in a much-complicated order, like doing asynchronous work.
7. Embed a unified type system

- For example,

The primitive types: int and double.

Please see more features in the below *Figure 4 The features of C#*.



**Figure 4 The features of C#**

### **2.2.3 The History and Development of C#**

Microsoft Corporation created C# in 2000. Anders Hjjlsberg was the principal designer who formed a team and designed this programming language. C # was called “Cool” initially. After that, the .NET announced a new programming language version and renamed it to C#.

As time goes by, C# comes to be an open-source programming language. Now it is improved by the community.

Please see *Table 1 The Overview for the development of C#*. The project uses C# 9, which is the latest version.

| <b>Date</b> | <b>Versions</b> | <b>Features</b>  |
|-------------|-----------------|--|
| Jan. 2002   | 1.0             | Not clear.   |
| Oct. 2003   | 1.2             | Modern, object-oriented, type-safe, automatic memory management, versioning control.   |
| Sep. 2005   | 2.0             | Generics, partial classes, anonymous types, iterators, nullable types, static classes, delegate interface.   |
| Aug. 2007   | 3.0             | Implicit types, object and collection initializers, auto-implemented properties, extension methods, query and lambda expressions, expression trees, partial methods.   |
| Apr. 2010   | 4.0             | Dynamic binding, named and optional arguments, Generic covariance and Contravariance, Embedded interop types.  |
| Jun. 2013   | 5.0             | Async methods, Caller info Attributes.   |
| Jul. 2015   | 6.0             | Roslyn (compiler-as-a-service), exception filters, await in catch/finally block, auto property initializer, string interpolation, operator's name, dictionary initializer.   |
| Mar. 2017   | 7.0             | Tuples, pattern matching, record types, local functions, Async streams.  |
| May 2018    | 8.0             | Readonly members, default interface methods, declarations, static local functions, disposable ref structs, nullable reference types, asynchronous streams & disposable<br><br>Indices and ranges, Null-coalescing assignment, unmanaged constructed types, stackalloc in nested expressions, enhancement of interpolated verbatim strings. |
| Sep. 2020   | 9.0             | Records, init only setters, top-level statements, pattern matching enhancements, performance, and interop, native sized integers,  |

|  |  |   |
|--|--|---|
|  |  | function pointers, suppress emitting localsinit flag, Fit and finish features, target-typed new expressions, static anonymous functions, target-typed conditional expressions, covariant return types, extension GetEnumerator support for foreach loops, lambda discard parameters, attributes on local functions, support for code generators, module initializers, new features for partial methods. |
|--|--|---|

**Table 1 The Overview for the development of C#**

#### **2.2.4 The Scope of Application and Usage for C#**

Although C# is still a very young programming language, this does not affect the high demand for using it, and it turns out to be the most popular computer programming language. Nowadays, many software engineers use C# to build a number of different programs and applications. Additionally, C# is suitable for creating many websites and web apps using the .NET platform or other open-source platforms.

Over the above, C# also fits for creating many other programs, for example:

- Mobile applications
- Desktop applications
- Cloud-based services
- Games
- VR
- Enterprise software
- Database applications.
- Windows applications

## 2.2.5 The Comparison Between C# and Java

This section contains two parts, the similarities and the differences between C# and Java. A table exhibits a list of divergences between the two programming languages to the end of the section,

### 2.2.5.1 The similarities between C # and Java

1. Both are object-oriented programming languages.
2. Both are parts of the C family; in other words, both inherit from C language.
3. Both support some features, such as garbage collection and multiple class inheritance.
4. Two languages have a similar syntax; please see *Figure 5 Syntax comparison* (wikipedia.org) as an example below.

| Java   | C#   |
|--|--|
| <pre>class PassByRefTest {     static class Ref&lt;T&gt; {         T val;         Ref(T val) { this.val = val; }     }      static void changeMe(Ref&lt;String&gt; s) {         s.val = "Changed";     }      static void swap(Ref&lt;Integer&gt; x, Ref&lt;Integer&gt; y) {         int temp = x.val;          x.val = y.val;         y.val = temp;     }      public static void main(String[] args) {         var a = new Ref(5);         var b = new Ref(10);         var s = new Ref("still unchanged");          swap(a, b);         changeMe(s);          System.out.println( "a = " + a.val + ", " +             "b = " + b.val + ", " +             "s = " + s.val );     } }</pre> | <pre>class PassByRefTest {     public static void ChangeMe(out string s)     {         s = "Changed";     }      public static void Swap(ref int x, ref int y)     {         int temp = x;          x = y;         y = temp;     }      public static void Main(string[] args)     {         int a = 5;         int b = 10;         string s = "still unchanged";          Swap(ref a, ref b);         ChangeMe(out s);          System.Console.WriteLine("a = " + a + ", " +             "b = " + b + ", " +             "s = " + s);     } }</pre> |

Figure 5 Syntax comparison

### 2.2.5.2 The differences Between Java and C#

1. Different Runtime environments,

## *Build a Dashboard Application for NOVs eVolve Automation System*

- C#: runs on CLR (Common Language Runtime)
  - Java: runs on JRE (Java Runtime Environment)
2. Different programming software,
- C#: use Visual Studio IDE
  - Java: requires Java Development Kit
3. Different usage/application,
- C#: aim at developing an application for Microsoft platforms
  - Java: aim at building a complex application
4. A different way to handle the exceptions
- C#: has only one type of exception.
  - Java: put the exceptions into different classes, that is checked exception, such as `FileNotFoundException`, and unchecked exception, for example, `ArithmeticException`, `ArrayStoreException`, and `ClassCastException` are unchecked exceptions in Java

There are more comparisons for the two programming languages showing in the following

*Table 2 The comparison between C# vs Java (Arora, 2020)*

| <b>Parameters</b>           | <b>C#</b>   | <b>Java</b>                               |
|-----------------------------|---|---|
| <b>Programming Paradigm</b> | Object-Oriented, component-oriented, functional, strong typing. | Class-based, an Object-Oriented language. |
| <b>Application</b>          | Windows applications. Web and game development.                 | Complex web-based applications.           |
| <b>Scope</b>                | Server-side language with a good programming foundation.        | Server-side interaction.                  |

|                                |   |   |
|--------------------------------|---|---|
| <b>Tools</b>                   | Visual Studio, Mono Develop.  | Eclipse, NetBeans, Intelli J<br>IDEA.   |
| <b>Public Classes</b>          | Supports multiple public classes in source code.  | Java source code can have only one public class.  |
| <b>Checked Exceptions</b>      | Does not support checked exceptions.  | Supports checked and unchecked exceptions.  |
| <b>Platform Dependency</b>     | C# is cross-platform and supports both Windows and Unix based systems.  | Java is platform-independent but needs JVM for its execution.                                     |
| <b>Conditional Compilation</b> | Supports conditional compilation.   | Does not support conditional compilation.   |
| <b>Go to statement</b>         | Supports the go-to statement.   | Does not support the go-to statement.   |
| <b>Structure and Union</b>     | Supports structures and unions.   | Does not support structures and unions.   |
| <b>Floating Point</b>          | The result of floating-point numbers may not be guaranteed to be the same across all platforms as C# does not support strictfp keyword. | The strictfp keyword is supported by Java, and hence the result is the same across all platforms. |

**Table 2 The comparison between C# vs Java**

### **2.2.6 The Decision Made to Use C# Instead of Java for This project.**

After the above study on the C# and the comparison between C# and Java, it is evident that the C# is easily integrated into Windows. Since the dashboard application should be installed on a Windows system computer, the C# is the best programming language for this project.



## **2.3 VS 2019 IDE - Programming Software for Windows**

The introduction of the programming software, VS 2019, will be presented generally at the beginning of this section. The following section sketches the construction of a simple windows forms C# application on VS 2019 IDE.

### **2.3.1 The VS2019 IDE**

Microsoft publishes Visual Studio to be an Integrated development environment. It is the most widely used programming software to develop websites, mobile applications, and computer programs. It does not support any programming languages, but it can install the programming language as a language service package. When the specific programming language is mounted, for example, the C#. the functionalities, such as syntax coloring, brace matching, statement completion, parameter information tooltips, member lists, and error markers for background compilation, are available as a service. Withing the functions above, it becomes one of the most popular software development platforms. However, the Visual Studio is not free of charge; it only has a few months for a free trial.

#### **The features for Microsoft Visual Studio:**

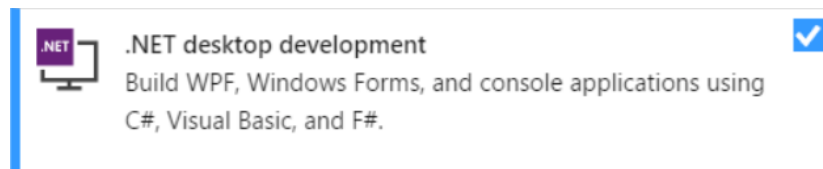
- IntelliCode editor
- Debugger
- Designer
  - The Windows Forms designer is used for building the dashboard applications for this project.
- Test Explorer
- Microsoft Edge Insider support
- Pinnable Properties tool

### 2.3.2 Creating a Windows Forms app in VS 2019 with C#

The example refers to a tutorial about creating a Windows Forms app in Visual Studio with C# from Microsoft Doc (Doc) is going to be introduced step by step in the section.

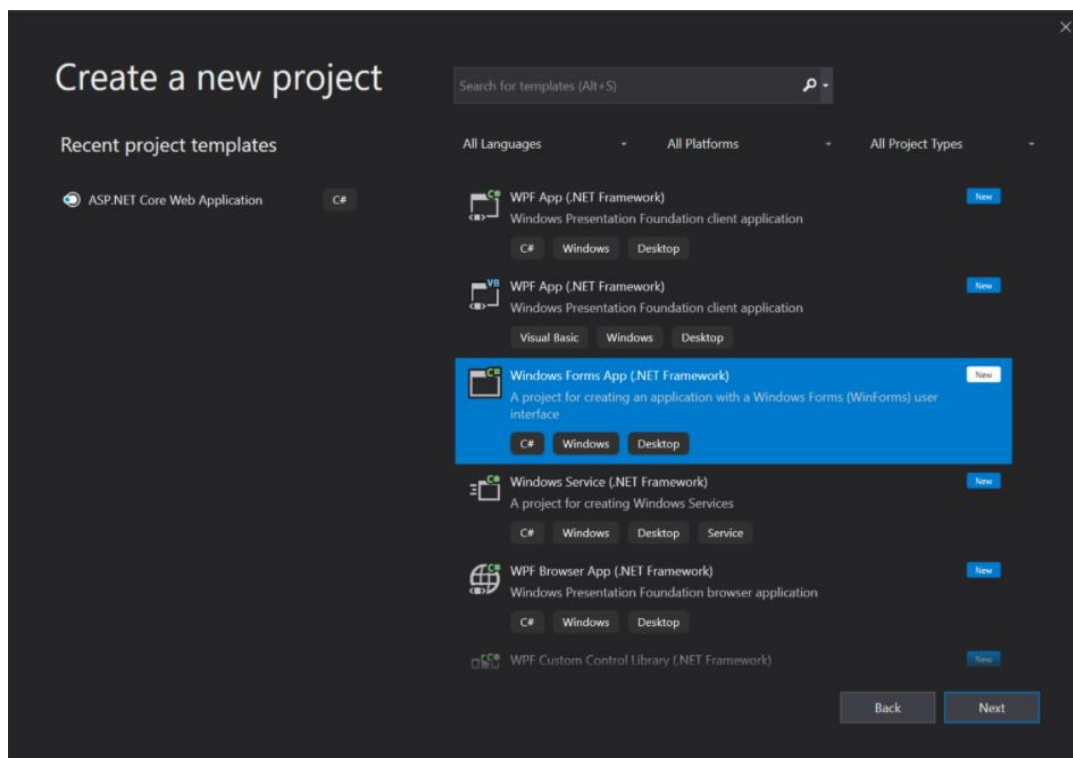
Step 1. The prerequisites.

- Download the VS 2019 IDE
- Install the Windows Forms App (.NET Framework) template for C# by checking the check box at the top-right. Please see *Figure 6 .NET desktop development workload* for reference below.



**Figure 6 .NET desktop development workload**

- Create a new project. The language selection should be C# and select the Windows Forms App (.NET Core), as *Figure 7 Create a new project* shows below.

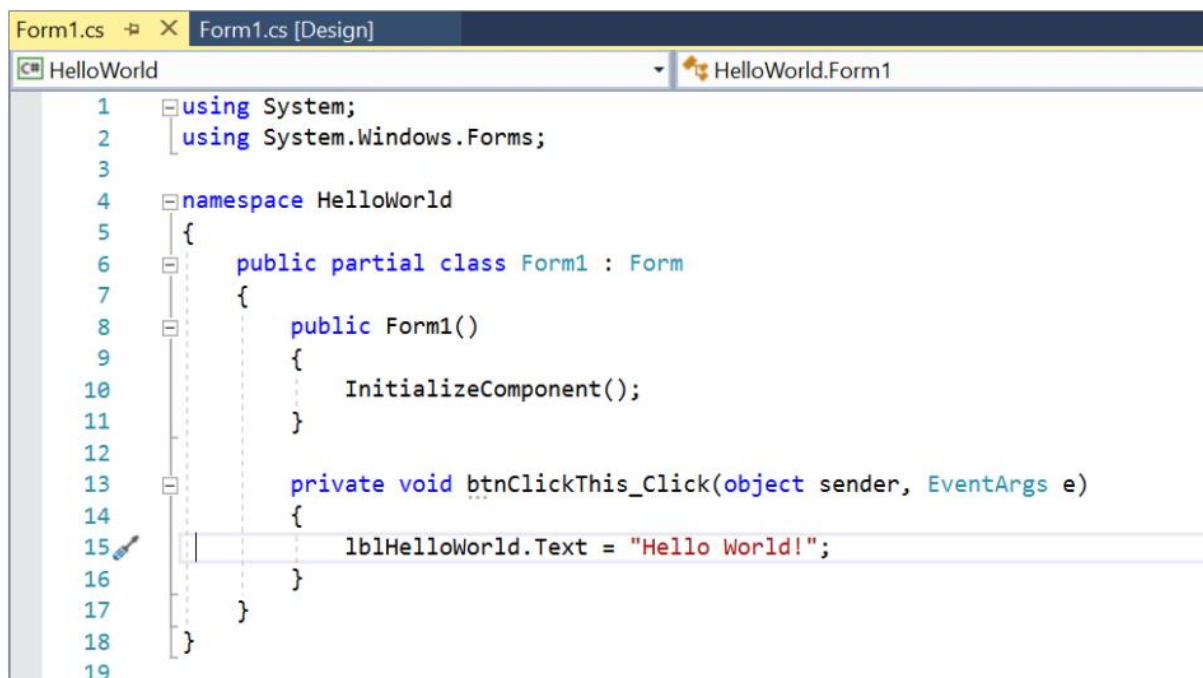


### Figure 7 Create a new project

- Name the newly created project on the configuration window.

Step 2. Start the new project

- Three files are generated by creating the new project. One is named Form1.cs, one is named Form1.designer.cs, and the other is named Program.cs. The following explanation for those three files refers to the answer from stackoverflow.com (Rashedul.Rubel).
  - The file Form1.cs is the coding file of the windows forms app. It is the *class* file of the Windows Forms app where the necessary methods, functions, and events are customized written. *Figure 8 Form1.cs* gives a picture of how Form1.cs looks like (Doc). It shows an example for printing the text “Hello World!”



```
1  using System;
2  using System.Windows.Forms;
3
4  namespace HelloWorld
5  {
6      public partial class Form1 : Form
7      {
8          public Form1()
9          {
10             InitializeComponent();
11          }
12
13         private void btnClickThis_Click(object sender, EventArgs e)
14         {
15             lblHelloWorld.Text = "Hello World!";
16         }
17     }
18 }
19
```

Figure 8 Form1.cs

- The file Form1.designer.cs is the designer file where form elements are initialized. If any component is dragged and dropped in the form window, then that element

will be automatically initialized in this class. This file should not be removed, and it is not recommended to be changed either.

- The file Program.cs is the *main* of the application, which is a static method called `static void Main()`. This is executed first when the application runs.

## **2.4 Theory for Building Dashboard**

This section introduces the design and layout for creating the dashboard application with a graphical presentation of assumptions. The functions of the dashboard are also involved.

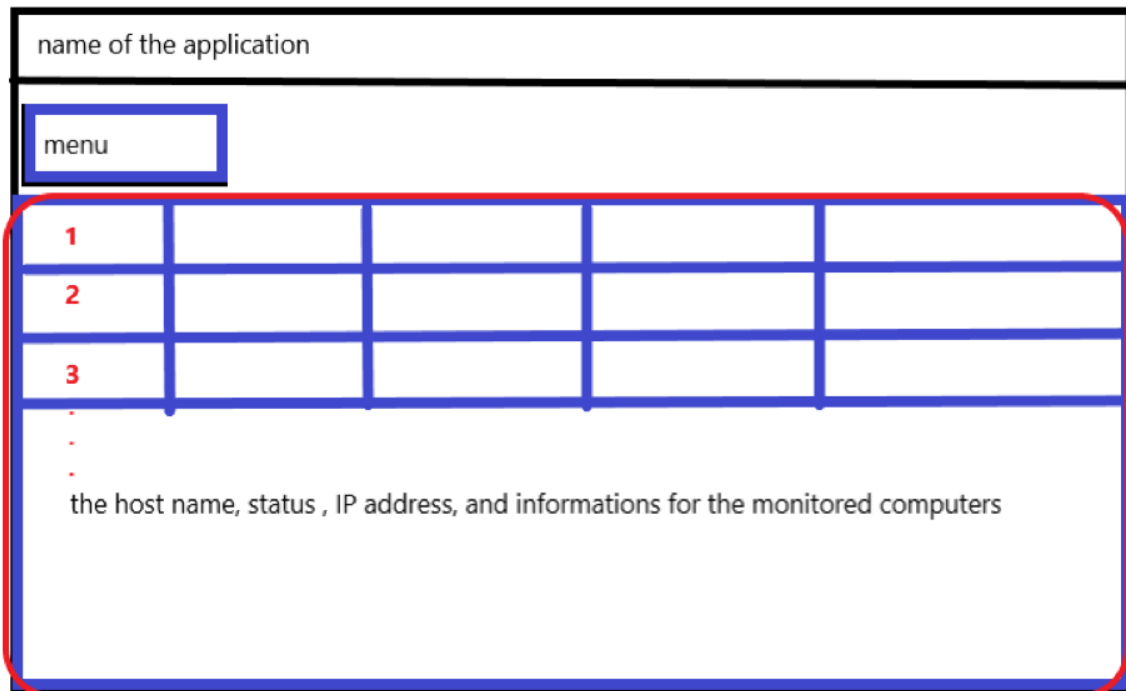
### **2.4.1 The Graphical Composition of the Dashboard**

The dashboard contains three windows.

1. The window shows the layout of the dashboard.
2. The window to add computers.
3. The window to manage the added computers.

### **2.4.2 The Graphical layout and the Functions of the Dashboard**

The first window is the main window; it shows the running status of the computers; please see *Figure 9 Layout of dashboard window 1* below. The main window will pop out right after launch the dashboard application by clicking the icon on the desktop.



**Figure 9 Layout of dashboard window 1**

The main window comprises three areas

1. The top area gives the name of the application.
2. The second area shows the menu, locates in the middle.
3. The third area settles at the bottom of the main window, where is reserved by the general information for the monitored computer, such as hostname, IP address, and status for each computer.

The computers can be remotely monitored by the application parallely and added to the third area by clicking the *add* button in the menu area. The information for each computer is plugged in the columns, row by row in the third area, which is the red part from *Figure 9 Layout of dashboard window 1*.

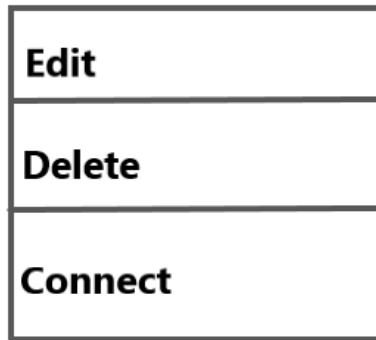
By clicking the button in the menu, for example, the *Add* button, it triggers an event and pops out the second window, which is the *add form*. Please see *Figure 10 Add PC* on the next page.

| Add PC                              |                                       |
|-------------------------------------|---------------------------------------|
| IP                                  |                                       |
| Username                            |                                       |
| Password                            |                                       |
| Port                                |                                       |
| <input type="button" value="Save"/> | <input type="button" value="Cancel"/> |

Figure 10 Add PC

When a remote PC is added to the dashboard, the third area displays the PC as the above description. The add form prompts for the IP, username, password, and Port for connecting with the remote computer. By default, the RDP (Remote Desktop Protocol) server listens on TCP port 3389. Remote Desktop Connection is Windows built-in application and designed for remote control.

The third window is for managing remote computers. There are three options, edit, delete and connect. Each option controlled by the event handler, clicking the associated option buttons, triggers the corresponding event handler. Please see the *Figure 11 Management window*.



**Figure 11 Management window**

The above management window pops out by right-clicking at any place of the entire row of the remote computer. When placing the mouse on the row of the remote computer, then left-clicking, the color of the row turns immediately to be blue background, which means the row is selected. Afterward, right-clicking the mouse shows the management window. The delete option is to remove the entire row by left-clicking the *Delete* button.

## **2.5 Summary of Chapter 2**

Chapter 2 explains why the programming language C# is appropriate to build the dashboard application by researching the features, touring the history of development, studying the scope for the usage of C#, and comparing it with the other programming language, Java.

The middle of chapter 2 briefly introduces the software development platform VS 2019 IDE. It extracts an example from the Microsoft Doc website to create a simple Windows forms application in VS 2019 IDE with C#.

The last part of the chapter describes each window with graphical layouts that illustrate how the dashboard looks and the functions for buttons.

## **Chapter 3: Implementation**

---

### **3.1 Introduction**

Chapter 3 brings a tour of how the dashboard application is knocked together in the software development platform VS 2019 IDE. When the software development environment is ready, create three source code files with the associated designers. Each file has the extension .cs. All of them are the partial class in the same namespace, *namespace Dashboard\_APP*.

1. The Form1.cs is the default and the most critical source code file. The following subsections present the functions and classes in the Form1.cs.
2. The AddPC.cs is the source code file to add the information of the remote PC.
3. The ConnForm.cs is the source code file to embed the Remote Desktop Connection app.

The code in the subsections presents statements, event handlers, methods, variables, and classes personalized for the project and are invoked in Form1.cs.

### **3.2 Create Form1.cs**

#### **3.2.1 Introduction of Form1.Designer.cs**

When a new project is added in VS 2019 IDE, rename the project to be *Dashboard\_APP*. A file named *Form1.Designer.cs* is generated automatically with the file *Form1.cs* once the new project is created, *Form1.Designer.cs* is also called Windows Forms Designer that provides many components in the Toolbox for rigging up Windows Forms applications.

A *Form1.Designer* is initially an empty form window; please see *Figure 13 An initial Form1.Designer*. It can be released by double-clicking Form1.cs or right-clicking Form1.cs, then select *view designer*. Please see *Figure 12 Form1.cs* below.



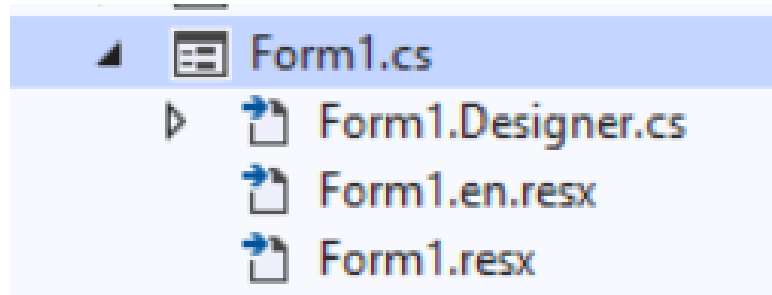


Figure 12 Form1.cs

From Figure 13 An initial Form1.Designer, there is a Toolbox window that flies out by clicking the button *Toolbox* located at the left of Form 1 designer, as Figure 14 *Toolbox* shows below.

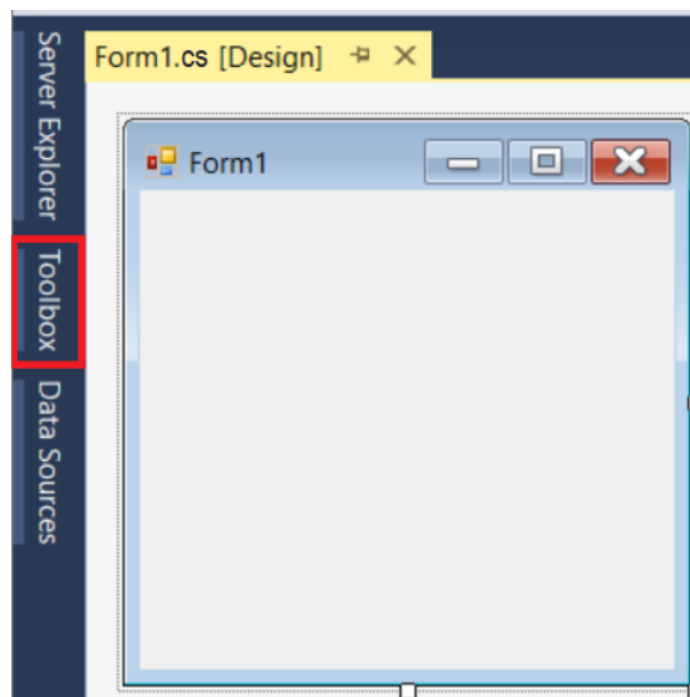


Figure 13 An initial Form1.Designer

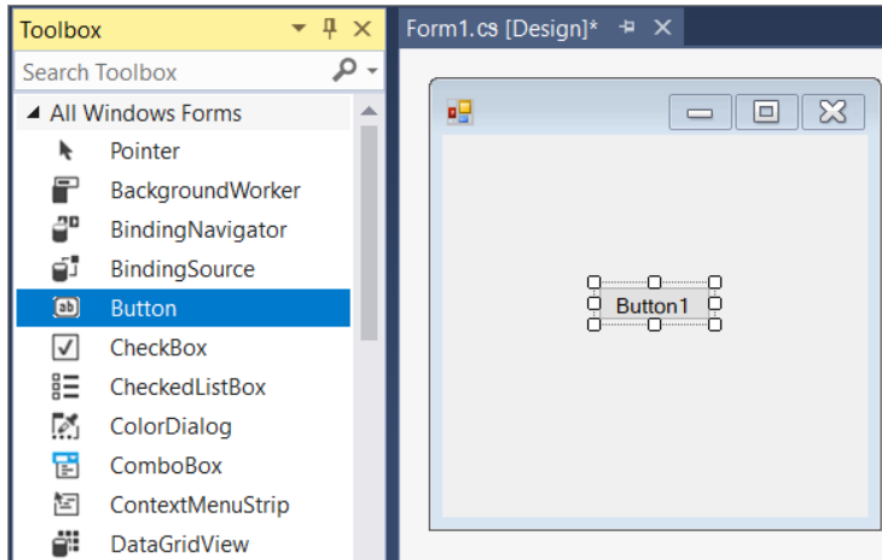


Figure 14 Toolbox

The Form1.cs can be designed by dragging the tools from the Toolbox to the form1 window. Here a button1, a panel1, and a dataGridView1 are placed on Form1 as Figure 15 Form1 designer for Dashboard App.

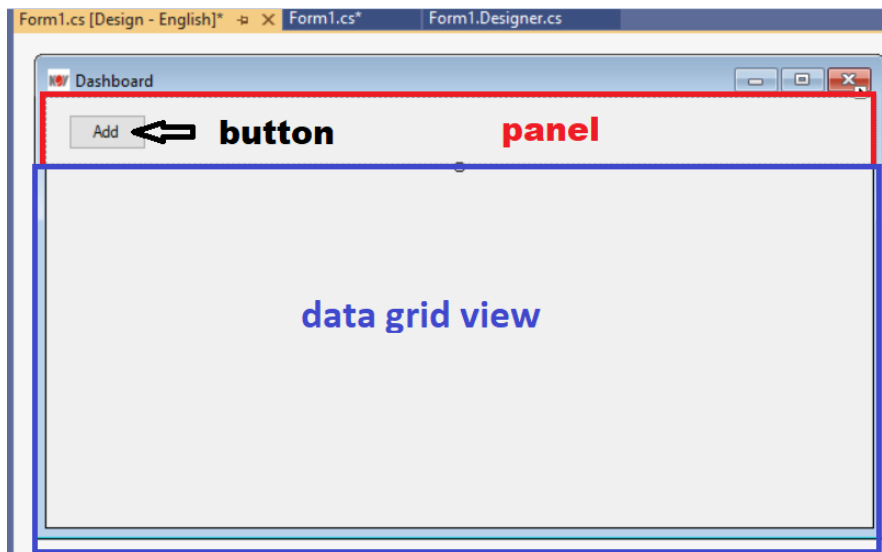


Figure 15 Form1 designer for Dashboard App

Form1 can be individualized from the Properties window; Figure 16 Properties window shows that the button, text, and icon are already changed to fit the project.

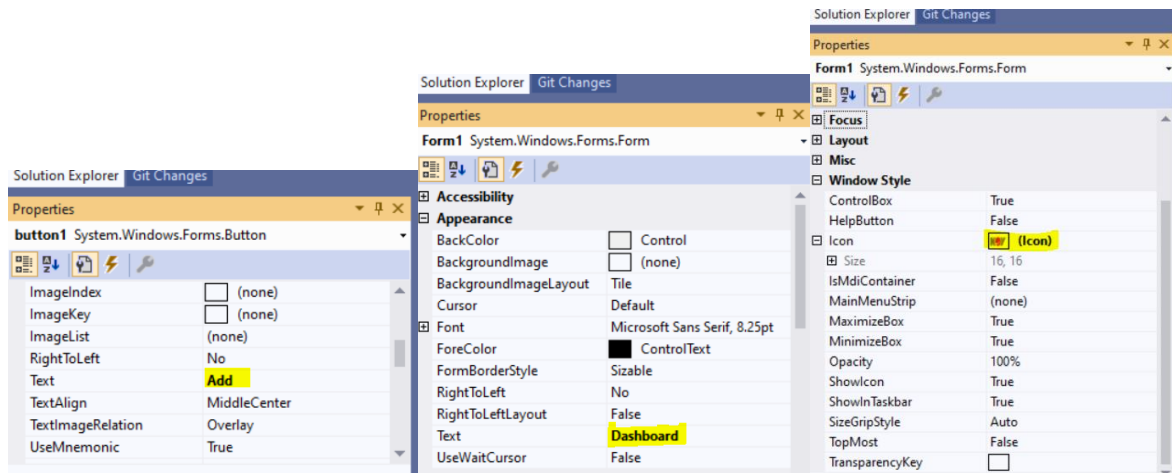


Figure 16 Properties window

The Windows Form Designer generates the code for Button1, Panel1, and dataGridView1. All of them are placed in a separate section in Form1.Designer.cs as Figure 17 Windows Form Designer generated code for Form1 shows.

```

48     // button1
49     //
50     resources.ApplyResources(this.button1, "button1");
51     this.button1.Name = "button1";
52     this.button1.UseVisualStyleBackColor = true;
53     this.button1.Click += new System.EventHandler(this.button1_Click);
54     //
55     // dataGridView1
56     //
57     resources.ApplyResources(this.dataGridView1, "dataGridView1");
58     this.dataGridView1.AllowUserToAddRows = false;
59     this.dataGridView1.AllowUserToDeleteRows = false;
60     this.dataGridView1.BackgroundColor = System.Drawing.SystemColors.Control;
61     this.dataGridView1.ColumnHeadersHeightSizeMode = System.Windows.Forms.DataGridViewColumnHeadersHeightSizeMode.AutoSize;
62     this.dataGridView1.Name = "dataGridView1";
63     this.dataGridView1.ReadOnly = true;
64     this.dataGridView1.RowTemplate.Height = 30;
65     this.dataGridView1.CellFormatting += new System.Windows.Forms.DataGridViewCellFormattingEventHandler(this.dataGridView1_CellFormatting);
66     this.dataGridView1.CellMouseUp += new System.Windows.Forms.DataGridViewCellMouseEventHandler(this.dataGridView1_CellMouseUp);
67     //
68     // panel1
69     //
70     resources.ApplyResources(this.panel1, "panel1");
71     this.panel1.Controls.Add(this.button1);
72     this.panel1.Name = "panel1";

```

Figure 17 Windows Form Designer generated code for Form1

### 3.2.2 Create a Form1 Class in Form1.cs

The class begins with *public partial class Form1 : Form*.

It is possible to split a Class into several source code files, as long as they are in the same Namespace. The *partial* keyword indicates that the class *Form1* is a part of the project. *Form1* is the name of the class. Every partial Class can contain several sections of methods, and all parts are combined to improve the application.

The colon `:` stands for the *class Form1* inherits the base class form's properties, which represents *System.Windows.Forms.Form*. It is inheriting to access the properties and methods of the base class.

Form1 is created by default whenever a Windows Forms App project starts in the software development platform VS 2019 IDE, and it always begins with the following code. The curly bracket should surround the body of Form1. Chapter 4 displays the implementation of the class Form in detail.

```
public partial class Form1 : Form
{
}
```

### 3.2.3 Create a Class PCInfo in Form1.cs

The class starts with *public class PCInfo*

The keyword *public* defines the access rules of the class named PCInfo. It indicates that the class *PCInfo* is accessible outside the assembly. Still, if the *public* keyword is not declared, the class type is defined as *internal* by default, which means that the class is only visible inside the same assembly, and the default access for the members is *private*.

The *PCInfo* class is intended for creating an object for each remote PC. The following code declares the *PCInfo* class.

```
public class PCInfo {
}
```

#### 3.2.3.1 The Constructor and Properties in PCInfo Class

The body of PCInfo class is defined between the curly brackets. It begins with a constructor. A constructor is a special method that achieves initializing objects. The advantage of a constructor is that it is called when creating an object of a class, and it can contain the initial values for fields (w3schools):

```
//Constructor
public PCInfo() { }
```

From the *Figure 15 Form1 designer for Dashboard App* in the previous section, there is an area called data grid view that locates at the bottom of the main window, where displays the

information of the remote PC object in rows. All the fields are declared right after the constructor (Note. The variables inside a class are called fields). Each PC object includes IP, hostname, username, password, port, isOnline, status, isConn, offline time, and the offline duration.

```
//create fields
private string _ip;
private string _hostName;
private string _username;
private string _password;
private string _port;
private bool _isOnline;
private string _status;
private bool _isConn;
private DateTime? _offlineTime;
private int _offlineDuration;
```

The mechanism is especially for binding the code and the data. It manipulates in the PCInfo class called Encapsulation. Encapsulation means wrapping up data under a single unit. It is a protective shield that prevents the data from being accessed by the code outside this shield. The meaning of Encapsulation is to make sure that the "sensitive" data or variables are hidden from any other class but can be accessed only through any member function of their own class in which they are declared (GeeksforGeeks).

There are two steps below to achieve the above mechanism in the *PCInfo* class:

- Declaring all the variables/fields as private, see the code of “//create fields.”
- Using the public *set()* and *get()* methods to set the value and get the values of variables, see the code below. (Note. The *get()* and *set()* methods are also known as property, which combines the private variables and the methods. By the existence of the public *get()* and *set()* methods in the class, the private variables can be called or assigned a value by Dot(.) method from the other class.)

```
/// get and set method. The property
public string ip
{
    set { _ip = value; }
    get { return _ip; }
}

public string hostName
{
    set { _hostName = value; }
    get { return _hostName; }
}

public string username
```

```
{
    set { _username = value; }
    get { return _username; }
}

public string password
{
    set { _password = value; }
    get { return _password; }
}

public string port
{
    set { _port = value; }
    get { return _port; }
}

public bool isOnline
{
    set { _isOnline = value; }
    get { return _isOnline; }
}

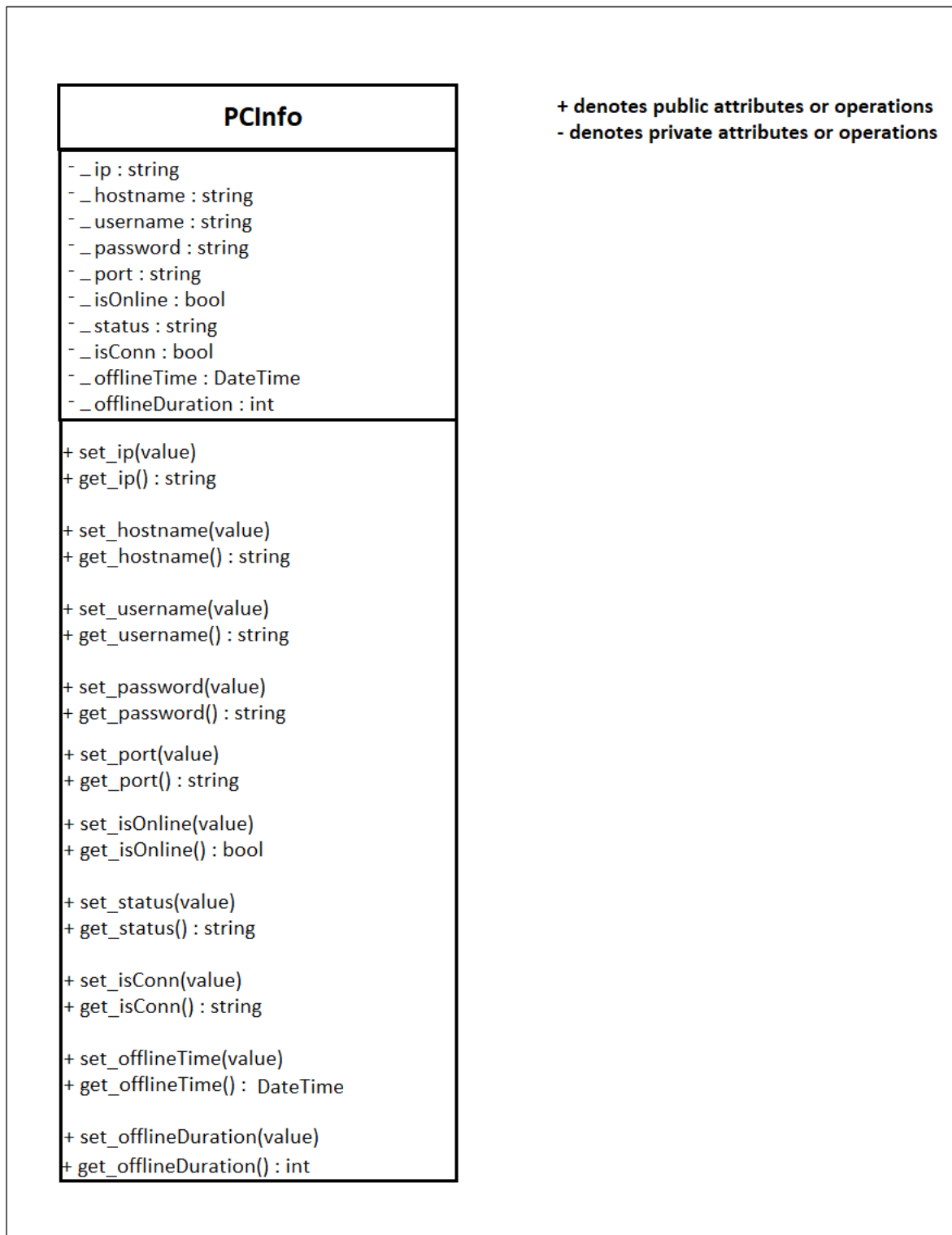
public string status
{
    set { _status = value; }
    get { return _status; }
}

public bool isConn
{
    set { _isConn = value; }
    get { return _isConn; }
}

public DateTime? offlineTime
{
    set { _offlineTime = value; }
    get { return _offlineTime; }
}

public int offlineDuration
{
    set { _offlineDuration = value; }
    get { return _offlineDuration; }
}
}
}
```

Please see the class diagram for PCInfo from *Figure 18 Class diagram for PCInfo* below.



*Figure 18 Class diagram for PCInfo*

Please see the completed code for *PCInfo* class below:

```
public class PCInfo
{
    //Constructor
    public PCInfo() { }
    //create fields
    private string _ip;
    private string _hostName;
    private string _username;
    private string _password;
    private string _port;
    private bool _isOnline;
    private string _status;
    private bool _isConn;
    private DateTime? _offlineTime;
    private int _offlineDuration;
    /// <summary>
    ///
    /// </summary>
    /// get and set method. The property
    public string ip
    {
        set { _ip = value; }
        get { return _ip; }
    }

    public string hostName
    {
        set { _hostName = value; }
        get { return _hostName; }
    }

    public string username
    {
        set { _username = value; }
        get { return _username; }
    }

    public string password
    {
        set { _password = value; }
        get { return _password; }
    }

    public string port
    {
        set { _port = value; }
        get { return _port; }
    }

    public bool isOnline
    {
        set { _isOnline = value; }
        get { return _isOnline; }
    }

    public string status
    {
        set { _status = value; }
        get { return _status; }
    }
}
```



```
public bool isConn
{
    set { _isConn = value; }
    get { return _isConn; }
}

public DateTime? offlineTime
{
    set { _offlineTime = value; }
    get { return _offlineTime; }
}

public int offlineDuration
{
    set { _offlineDuration = value; }
    get { return _offlineDuration; }
}
}
```

When the PCInfo class is completed, an object can be created by specify the class name *PCInfo*, followed by the object name *info*, and use the keyword *new*:

```
PCInfo info = new PCInfo();
```

### 3.3 Create AddPC.cs

A programming project usually assembles several Windows Forms. A new Windows form can be added by right-clicking the project's name and then click *Add*, selecting *New Item*. Then an Add New Item window shows up; please see *Figure 19 Add new Windows form*.

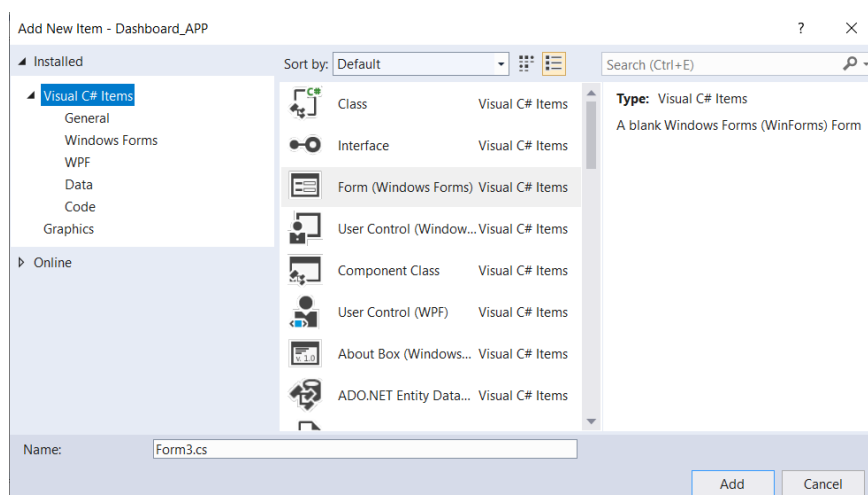



Figure 19 Add new Windows form

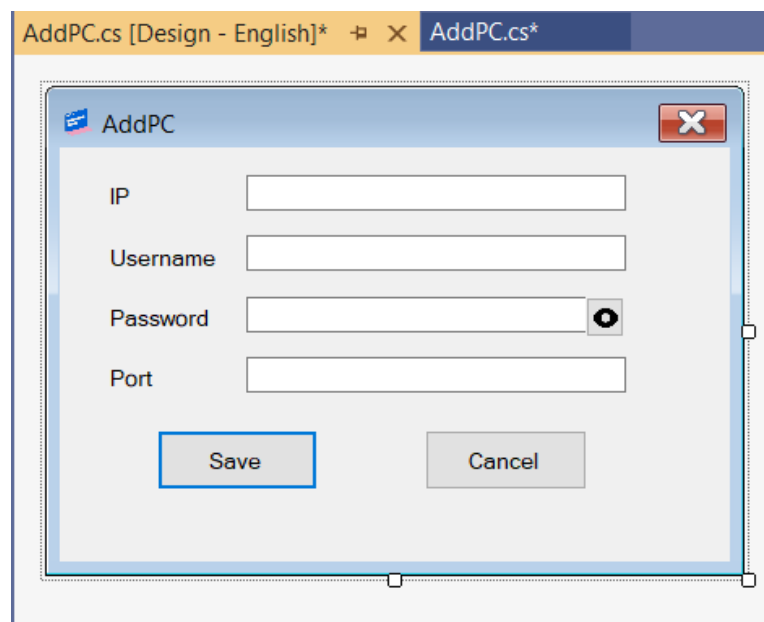
Selecting *Form* highlighted in the gray color from *Figure 19 Add new Windows form*, then clicking Add button to the end. The default name of the Windows Form is *Form2.cs*, renaming it to *AddPC.cs* by right-clicking on the term. The name of the auto-generated designer also alters to *AddPC.Designer.cs* accordingly.

### 3.3.1 Introduction of AddPC.Designer.cs

The *AddPC* form is designed according to the layout presented in section 2.4.3. The *AddPC* form consists of four labels, four text boxes, and three buttons. Please find the corresponding name for these components in *Table 3 AddPC*, and see the picture of the AddPC form in *Figure 20 AddPC Form*.

| Item       | Name                    | Item       | Name  |
|------------|-------------------------|------------|---|
| Label1     | IP                      | Text box 3 | Text field for Password   |
| Label2     | Username                | Text box 4 | Text field for Port   |
| Label3     | Password                | Button 1   |  |
| Label4     | Port                    | Button 2   | Save  |
| Text box 1 | Text field for IP       | Button 3   | Cancel  |
| Text box 2 | Text field for Username |            |   |

*Table 3 AddPC*



*Figure 20 AddPC Form*

Windows Form Designer also generates the following code for these components, and locates in *AddPC.Designer.cs*

```
private System.Windows.Forms.Label label1;  
private System.Windows.Forms.Label label2;  
private System.Windows.Forms.Label label3;  
private System.Windows.Forms.Label label4;  
private System.Windows.Forms.TextBox textBox1;  
private System.Windows.Forms.TextBox textBox2;  
private System.Windows.Forms.TextBox textBox3;  
private System.Windows.Forms.TextBox textBox4;  
private System.Windows.Forms.Button button2;  
private System.Windows.Forms.Button button3;  
private System.Windows.Forms.Button button1;
```

### 3.3.2 The AddPC.cs

*AddPC* class is defined as a partial class in the Namespaces *Dashboard\_APP* together with the *Form1* class. The *AddPC* class is designed to read the *PCInfo* object's information from the text box and add the *PCInfo* to the data grid view components by clicking *Save* button.

The *AddPC* class begins with a constructor that initializes objects, and it is called when an object of *AddPC* is created in the Class *Form1*. The *Form1* class will be introduced in chapter 4.

```
namespace Dashboard_APP  
{  
    public partial class AddPC : Form  
    {  
        // constructor  
        public AddPC()  
        {  
            InitializeComponent();  
        }  
    }  
}
```

Create a private field named *\_info*, with the type *PCInfo*, which indicates the *\_info* is an instance of *PCInfo* class. The public *PCInfo* class is specified in the previous section 3.2.3 *Create a Class PCInfo in The Source Code Form1.cs*.

```
private PCInfo _info;
```

Each *PCInfo* object has the following variables.

- IP
- Hostname
- Username

- Password
- port

The values of each variable need to be collected from the text boxes by using the *get()* method, and write it to each variable by using the *set()* method. Please see the following code.


```
// collecting information for PCInfo
public PCInfo info
{
    //read the information for PCInfo object
    get
    { //initialize the PCInfo object if there is no input.
        if (_info == null)
        {
            _info = new PCInfo();
        }
        //read IP from the input in the text box 1
        _info.ip = textBox1.Text;
        //read username from the input in the text box 2
        _info.username = textBox2.Text;
        // read password from the input in the text box 3
        _info.password = textBox3.Text;
        // read port from the input in the text box 4
        _info.port = textBox4.Text;
        // return
        return _info;
    }
    // write the value to the variables for each PCInfo object.
    set
    {
        //set value to PCInfo object named _info
        _info = value;
        // write IP
        textBox1.Text = _info.ip;
        // write username
        textBox2.Text = _info.username;
        // write password
        textBox3.Text = _info.password;
        // write port
        textBox4.Text = _info.port;
    }
}
```

### 3.3.2.1 Create an Event Handler for Button1

As mentioned in section 3.3.1 *Introduction of AddPC.Designer.cs*, there are three buttons added in the AddPC form. A button is a Button control, which processes the button click event. The Click event is raised whenever the Button control is clicked. A Click Event Handler can be declared beginning with a special format:

```
private void button_click(object sender, EventArgs e) {

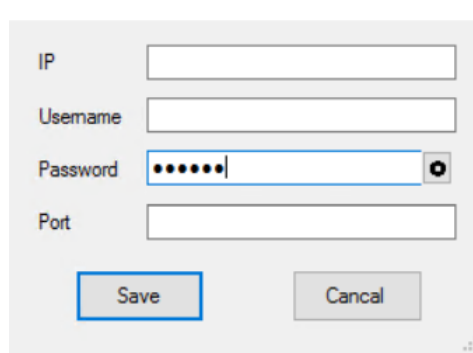
}
```

The following Click Event Handler sets the parameter for *TextBox.UseSystemPasswordChar Property* to *False*. When the GUI user clicks Button1  , the action enables the password visible in plain text.

```
// Add button1 click event handler
private void button1_Click(object sender, EventArgs e)
{
    // change the property value to False
    textBox3.UseSystemPasswordChar = !textBox3.UseSystemPasswordChar;
}
```

The *UseSystemPasswordChar Property* of a button determines whether user-supplied input should be displayed in the *MaskedTextBox* as multiple occurrences of a password character defined by the operating system. *UseSystemPasswordChar* functions use a programmer-supplied character for the prompt, and it also uses a prompt defined by the operating system (Doc, *MaskedTextBox.UseSystemPasswordChar Property*).

By default, the parameter for *TextBox.UseSystemPasswordChar Property* is *True*. If so, The password is displayed as it shows in *Figure 21 Password*.



*Figure 21 Password*

### 3.3.2.2 Create an Event Handler for Button2

Button2 is the *Save* button. A Boolean method is created to verify if the IP address is in the correct format before creating an Event Handler to Button2 control. An eligible IP address should be split into four cells with a dot (.) between the cells, e.g., 192.168.xxx.xxx, the Boolean method is named *IsIP()*, and it takes a string parameter. Please see the code below.

```
// create a Boolean method named IsIP to verify the IP address is in the correct format.
public bool IsIP(string IP)
{
    // split the IP parameter, and count the number of the elements.
    var iCount = IP.Split('.').Count();
    // if the number of the elements is not 4 return false
}
```

```
if (iCount != 4)
{
    return false;
}
// create an IP address object named ip
System.Net.IPAddress ip;
// if the string IP can be converted to integer ip successfully,
//return true. Otherwise return false.
if (System.Net.IPAddress.TryParse(IP, out ip))
{
    return true;
}
else
{
    return false;
}
}
```

A variable named *iCount* is declared at the beginning of the Boolean method *IsIP()* and assigned by the return of *Split()* method and *Count()* method. The *Split()* method and *Count()* method are called for splitting the string parameter IP with a dot (.) into cells and then return an integer number that represents the number of cells. The method *Split()* Returns a string array that contains the substrings. Moreover, the method *Count()* returns an *int* indicating the number of elements in that string array. A valid IP address should have four cells.

The flowing *if* statement with the condition *iCount != 4*, returns false if the IP address cannot be split into four cells. If so, it will skip the *if* statement and create an *IP Address* object by declaring the type *IPAddress*.

The IP should be converted from string type to *int* type by *TryParse()* method in the second *If* statement meanwhile, the second *If* statement returns True. Otherwise, it returns False.

When the Boolean method *IsIP()* is completed, it is time to add an Event Handler to Button2 control since the button2\_Click Event Handler *IsIP()* method needs to call *IsIP()* method. Here is the code for the button2\_Click Event Handler.

```
//Add button2 click Event Handler
private void button2_Click(object sender, EventArgs e)
{
    // examine the input of each text box is empty, or null.
    if (string.IsNullOrEmpty(textBox1.Text.Trim()) ||
        string.IsNullOrEmpty(textBox2.Text.Trim()) ||
        string.IsNullOrEmpty(textBox3.Text.Trim()) ||
        string.IsNullOrEmpty(textBox4.Text.Trim()))
    {
        //if any of the input is empty, show a message.
        MessageBox.Show("Cannot be empty!");
        return;
    }
    // call method IsIP to verify the IP
    if (!IsIP(textBox1.Text.Trim()))
    {
        // if it is not IP, show a message.
    }
}
```

```
    MessageBox.Show("Illegal IP!");  
    return;  
}  
// set the DialogResult is OK if any of the above "If" statements is triggered.  
this.DialogResult = DialogResult.OK;  
}
```

Once the Button2 (also known as the *Save* button) is clicked, the input of each text box needs to be verified if the information is null or empty by calling *String.IsNullOrEmpty(String)* method, if any of the input is empty or null, a message box will pop up showing the message “Cannot be empty”. The *String.IsNullOrEmpty(String)* method is a Boolean method, which indicates whether the specified string is null or an empty string. It returns *true* if the value parameter is null or an empty string; otherwise, it returns *false*.

The *IsIP()* method is called to verify if the input in text box 1 is in the correct IP address format or not. A message box will show up if the IP address is invalid (when the *IsIP()* method returns *false*). When the input in the text boxes is correct, setting the value for *DialogResult* property to “OK”.

### 3.3.2.3 Setup Button3

On Visual Studio 2019, a button property can be defined in the Properties window or defined by code in the source code file. Button 3 is the Cancel button; The dialog box return value is *Cancel* when the Cancel button is clicked. Please see *Figure 22 Button3 DialogResult property*. It shows that the *button3.DialogResult* property is set up to “Cancel”.

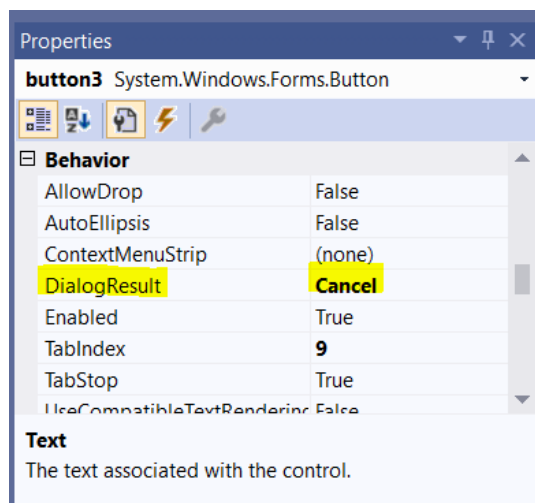
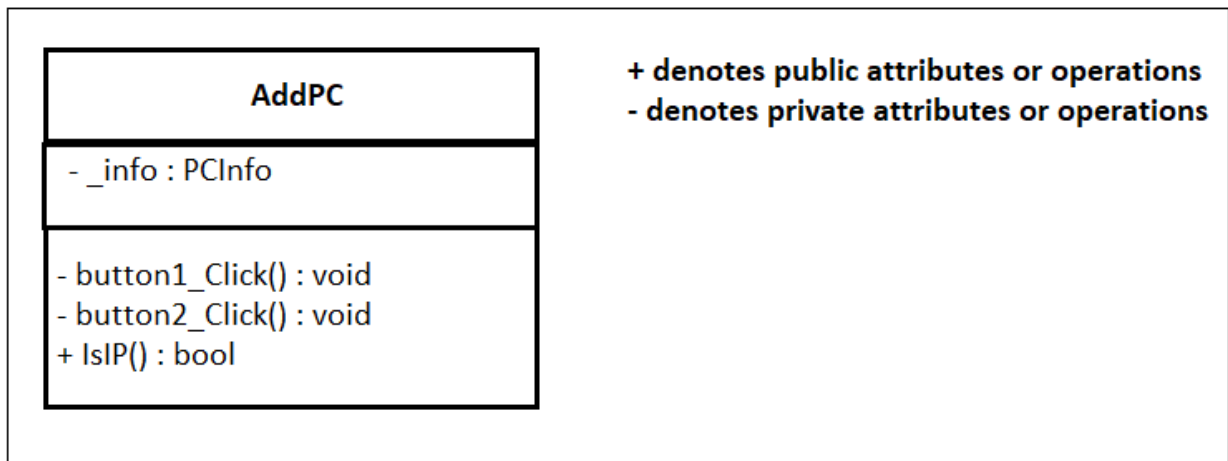


Figure 22 Button3 DialogResult property

Please see the class diagram for AddPC in *Figure 23 Class diagram for AddPC* below.



*Figure 23 Class diagram for AddPC*

Please see the completed code for class AddPC below.

```
using System;
using System.Linq;
using System.Windows.Forms;

namespace Dashboard_APP
{
    public partial class AddPC : Form
    {
        // constructor
        public AddPC()
        {
            InitializeComponent();
        }

        //define a private field named _info
        private PCInfo _info;

        // collecting information for PCInfo
        public PCInfo info
        {
            //read the information for PCInfo object
            get
            { //initialize the PCInfo object if there is no input.
                if (_info == null)
                {
                    _info = new PCInfo();
                }
                //read IP from the input in the text box 1
                _info.ip = textBox1.Text;
                //read username from the input in the text box 2
                _info.username = textBox2.Text;
                // read password from the input in the text box 3
                _info.password = textBox3.Text;
                // read port from the input in the text box 4
                _info.port = textBox4.Text;
                // return
                return _info;
            }
        }
    }
}
```



```
    }
    // write the value to the variables for each PCInfo object.
    set
    {
        //set value to PCInfo object named _info
        _info = value;
        // write IP
        textBox1.Text = _info.ip;
        // write username
        textBox2.Text = _info.username;
        // write password
        textBox3.Text = _info.password;
        // write port
        textBox4.Text = _info.port;
    }
}

//Add button2 click Event Handler
private void button2_Click(object sender, EventArgs e)
{
    // examine the input of each text box is string, empty, or null.
    if (string.IsNullOrEmpty(textBox1.Text.Trim()) ||
        string.IsNullOrEmpty(textBox2.Text.Trim()) ||
        string.IsNullOrEmpty(textBox3.Text.Trim()) ||
        string.IsNullOrEmpty(textBox4.Text.Trim()))
    {
        //if any of the input is empty, show a message.
        MessageBox.Show("Cannot be empty!");
        return;
    }
    // call method IsIP to verify the IP
    if (!IsIP(textBox1.Text.Trim()))
    {
        // if it is not IP, show a message.
        MessageBox.Show("Illegal IP!");
        return;
    }
    // set the DialogResult is OK if any of the above "If" statements is triggered.
    this.DialogResult = DialogResult.OK;
}

// create a boolean method named IsIP to verify the IP address is in the correct format.
public bool IsIP(string IP)
{
    // split the IP parameter, and count the number of the elements.
    var iCount = IP.Split('.').Count();
    // if the number of the elements is not 4 return false
    if (iCount != 4)
    {
        return false;
    }
    // create an IP address object named ip
    System.Net.IPAddress ip;
    // if the string IP can be converted to integer ip successfully,
    //return true. Otherwise return false.
    if (System.Net.IPAddress.TryParse(IP, out ip))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

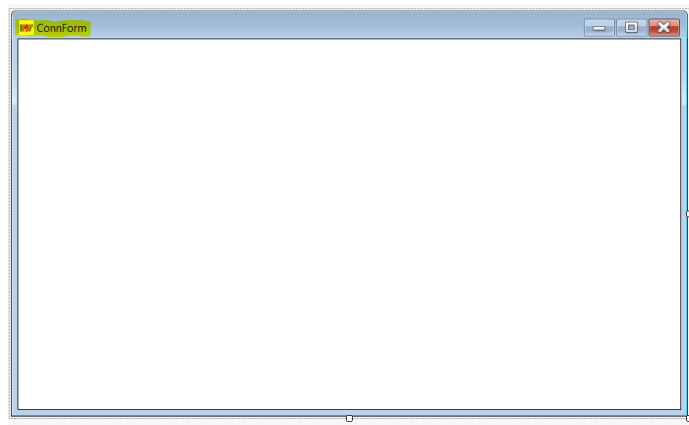
```
    }  
  
    // Add button1 click event handler  
    private void button1_Click(object sender, EventArgs e)  
    {  
        // change the property value to False  
        textBox3.UseSystemPasswordChar = !textBox3.UseSystemPasswordChar;  
    }  
}
```

### **3.4 Create ConnForm.cs**

A new Windows Form can be added by following the description at the beginning of section 3.3 *Create AddPC.cs*, and rename it to *ConnForm*. *MsRdpClient8*, *Ping*, and *PingReply* are the technologies/classes that are invoked to verify the condition of network communication between two computers. The *MsRdpClient8* class is one of the Remote Desktop ActiveX control classes.

The Designer has changed the name to *ConnForm* as long as the newly added Windows Form is renamed. The text and Icon can be modified by the designer, which can be personalized in the properties windows.

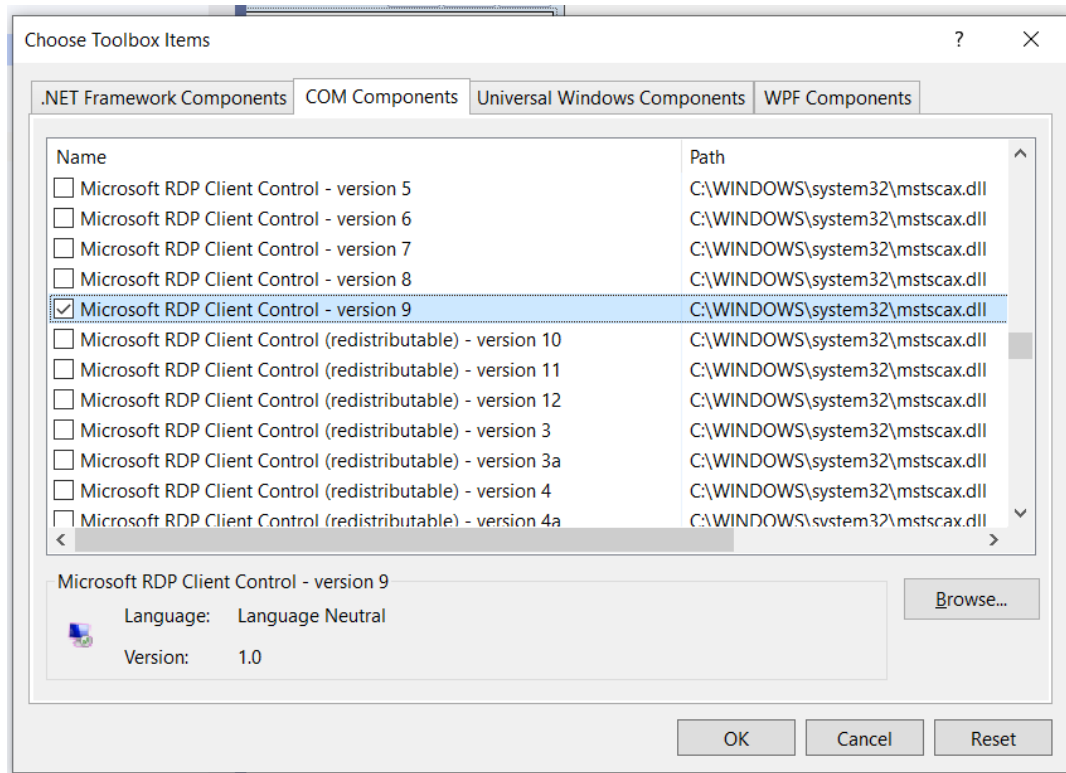
- Modify the text: Properties window -> Appearance -> Font -> Text
- Change the Icon: Properties window -> Windows Style -> Icon



***Figure 24 ConnForm designer***

Microsoft RDP Client Control ActiveX is designed for embedding a Remote Desktop Connection in the form at the blank area in *Figure 24 ConnForm designer*. Nevertheless, before that, the Microsoft RDP Client Control version 9 should be checked/added on Visual Studio by opening Toolbox -> Right-Click in the blank space-> select Choose Items -> Select COM

Components Tab -> Check Microsoft RDP Client Control version 9. Please see *Figure 25 Check Microsoft RDP Client Control - version 9* below.



*Figure 25 Check Microsoft RDP Client Control - version 9*

Now, the Microsoft RDP Client Control is available from ToolBox to create an *AxMSTSCLib.AxMsRdpClient8* object with the object name *axMsRdpClient81*. The code is generated from the Windows Form Designer located in the *ConnForm.Designer.cs*:

```
private AxMSTSCLib.AxMsRdpClient8 axMsRdpClient81;
```

### 3.4.1 The ConnForm.cs

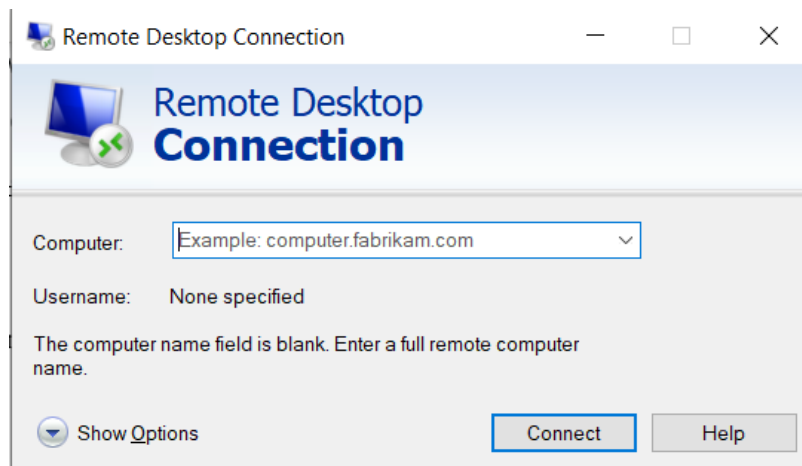
*ConnForm* (The *ConnForm* is short for Connection Form.) class is declared as the partial Class in the Namespace *Dashboard\_APP*, the same as the Source Code in *AddPC.cs* ,

```
namespace Dashboard_APP  
{  
    public partial class ConnForm : Form  
    {  
    }  
}
```

Afterward, create a constructor taking a *PCInfo* object as the parameter. When a constructor takes at least one parameter, it is called a parameterized constructor, and every instance of the class will be initialized with parameter (*PCInfo info*) values.

```
public ConnForm(PCInfo info)
{
    }
}
```

With Remote Desktop Terminal Services enabled, a single server can host multiple client sessions to establish a connection with remote computers from a local PC over a network connection. Please see the user interface for the Remote Desktop Terminal in *Figure 26 Remote Desktop Connection App*. The remote computer can be connected by typing in the IP address, the user name, and password for login to the remote computer.



**Figure 26 Remote Desktop Connection App**

*AxMsRdpClient8* class is a member of Remote Desktop ActiveX control classes. It is called to embed the Remote Desktop Connection App to the *ConnForm*.

Before establishing the remote connection with a remote computer, the remote computer needs to be confirmed online or offline. A *Ping* object and *PingReply* object need to create at the beginning of the constructor's body to achieve the goal. Both *Ping* and *PingReply* are in the namespace *System.Net.NetworkInformation*.

```
// create Ping object named pingsender
Ping pingsender = new Ping();
// create a PingReply object and confirm if the remote computer is online.
PingReply reply = pingsender.Send(info.ip);
```

Here the *Ping* class instance is created in provision for diagnosing whether a remote computer is reachable or not. Network topology can determine whether *Ping* can successfully contact a remote computer. A successful *Ping* indicates only that the remote computer can be reached on the network; the presence of higher-level services (such as a Web server) on the remote computer is not guaranteed (Doc, *Ping Class*). The presence and configuration of proxies, network address translation (NAT) equipment, or firewalls can prevent *Ping* from succeeding.

*Send(IPAddress)* method is called by *Ping* object named *pingsender*. It attempts to send an Internet Control Message Protocol (ICMP) echo message to the computer with the specified *IPAddress*. It receives a corresponding ICMP echo reply message from that computer and returns a *PingReply* instance.

An *If else* statement is compiled to set up the IP address, username, port number, and password of the remote computer that reads from the *PCInfo* info object to the embedded Remote Desktop Connection App when the *PingReply.status* returns *Success*. Otherwise, it will close the Remote Desktop Connection App and show a message in the message box.

```
if (reply.Status == IPStatus.Success)
{
    // setup IP
    axMsRdpClient81.Server = info.ip;
    // setup username
    axMsRdpClient81.UserName = info.username;
    //setup port number
    axMsRdpClient81.AdvancedSettings2.RDPPort = Convert.ToInt16(info.port);
    // setup size
    axMsRdpClient81.AdvancedSettings2.SmartSizing = true;
    // setup height
    axMsRdpClient81.DesktopHeight = this.Height;
    // setup width
    axMsRdpClient81.DesktopWidth = this.Width;
    // encryption is enabled
    axMsRdpClient81.AdvancedSettings9.NegotiateSecurityLayer = true;
    IMsTscNonScriptable securd = (IMsTscNonScriptable)axMsRdpClient81.GetOcx();
    // setup password
    securd.ClearTextPassword = info.password;
    axMsRdpClient81.AdvancedSettings5.ClearTextPassword = info.password;
    // setup color
    axMsRdpClient81.ColorDepth = 24;
    // establish the connection
    axMsRdpClient81.Connect();
}
//if there is no reply.
else
{
    // show the message box and close.
    MessageBox.Show("Unable to connect to the Server! ");
    this.Close();
}
```

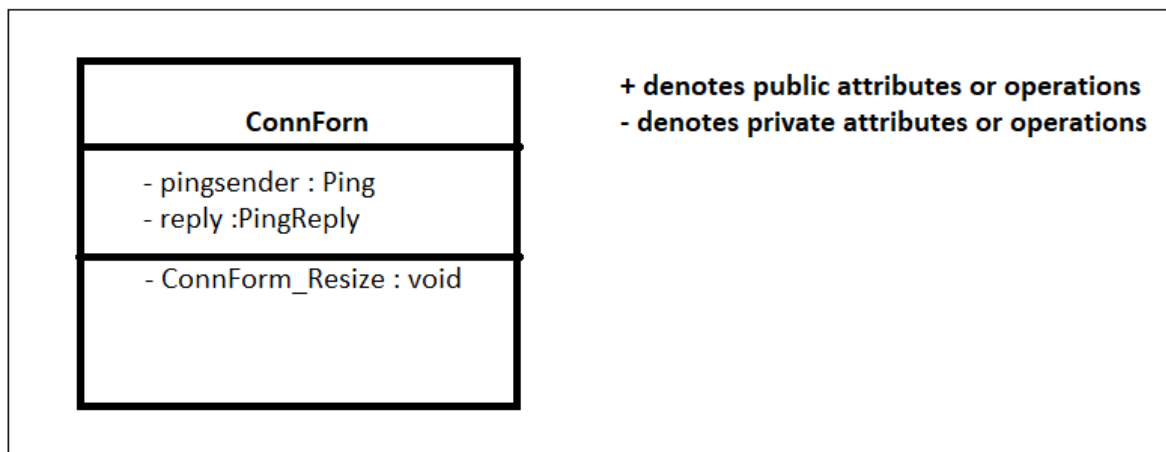
#### 3.4.1.1 Create an Event Handler for Setting Size

Add an event handler to reconnect the remote computer according to the customized height and width.

```
// resize event handler
private void ConnForm_Resize(object sender, EventArgs e)
{
    try
    {
        // reconnect the remote computer according to the height and width.
    }
}
```

```
        axMsRdpClient81.Reconnect((uint)this.Width, (uint)this.Height);
    }
    catch
    {
        throw;
    }
}
```

Please see the class diagram for ConnForm in *Figure 27 Class diagram for ConnForm* below.



*Figure 27 Class diagram for ConnForm*

Please see the completed code for class ConnForm below.

```
using MSTSCLib;
using System;
using System.Net.NetworkInformation;
using System.Windows.Forms;

namespace Dashboard_APP
{
    public partial class ConnForm : Form
    {
        // constructor
        public ConnForm(PCInfo info)
        {
            // initialize
            InitializeComponent();
            // create Ping object named pingsender
            Ping pingsender = new Ping();
            // create a PingReply object named reply and confirm if the remote computer is online.
            PingReply reply = pingsender.Send(info.ip);

            if (reply.Status == IPStatus.Success)
            {
                // setup IP
                axMsRdpClient81.Server = info.ip;
                // setup username
                axMsRdpClient81.UserName = info.username;
                //setup port number
                axMsRdpClient81.AdvancedSettings2.RDPPort = Convert.ToInt16(info.port);
                // setup size
                axMsRdpClient81.AdvancedSettings2.SmartSizing = true;
                // setup height
            }
        }
    }
}
```

```
axMsRdpClient81.DesktopHeight = this.Height;
// setup width
axMsRdpClient81.DesktopWidth = this.Width;
// encryption is enabled
axMsRdpClient81.AdvancedSettings9.NegotiateSecurityLayer = true;
IMsTscNonScriptable securd = (IMsTscNonScriptable)axMsRdpClient81.GetOcx();
// setup password
securd.ClearTextPassword = info.password;
axMsRdpClient81.AdvancedSettings5.ClearTextPassword = info.password;
// setup color
axMsRdpClient81.ColorDepth = 24;
// establish the connection
axMsRdpClient81.Connect();
}
//if there is no reply.
else
{
    // show the message box and close.
    MessageBox.Show("Unable to connect to the Server! ");
    this.Close();
}
}

// resize event handler
private void ConnForm_Resize(object sender, EventArgs e)
{
    try
    {
        // reconnect the remote computer according to the height and width.
        axMsRdpClient81.Reconnect((uint)this.Width, (uint)this.Height);
    }
    catch
    {
        throw;
    }
}
}
}
```

### **3.5 Summary of Chapter 3**

*Form1.Designer.cs* is the designer for the main form, and it is the most critical component for the dashboard application. It gives an introduction with illustrations of *Form1.Designer* at the beginning of the chapter and introduction of building the Windows form using the tool from ToolBox. The *Form1* form will undoubtedly need more tools. The associated event handlers are presented in chapter 4.

The *AddPC* form and *ConnForm* form with the associated source code files are put in place after the introduction of *Form1.Designer.cs*. These two forms improve the function of the dashboard application. The *AddPC* and *ConnForm* classes are designed to creating individual instances of objects called in the *Form1* class. In other words, it is in preparation for implementing object-oriented programming.

## Chapter 4: Implementation of Form1.cs

---

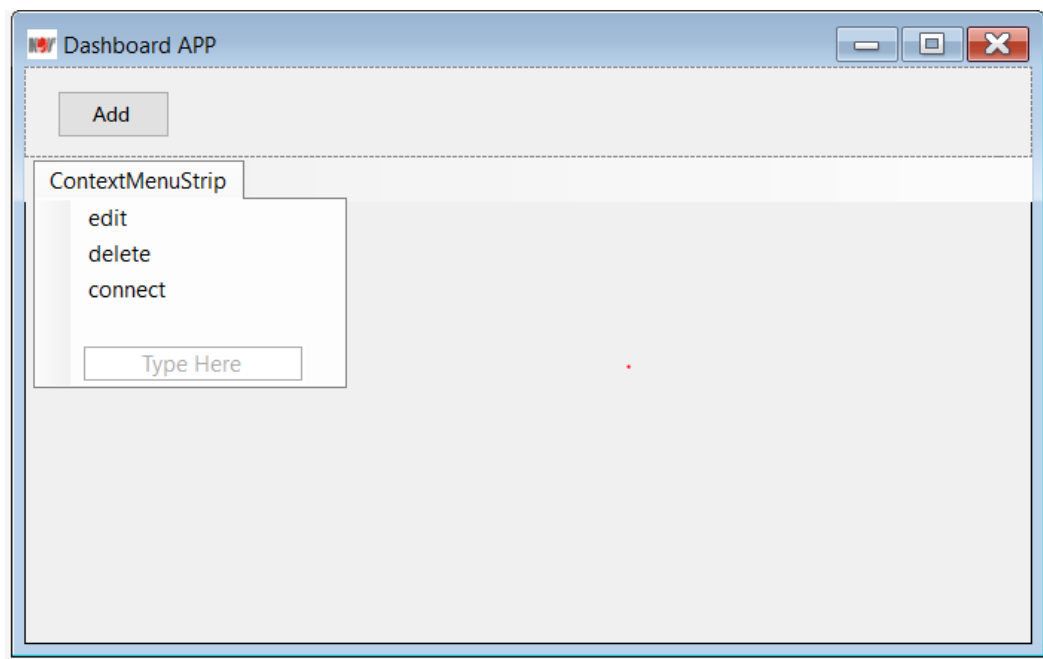
### 4.1 Introduction

As mentioned at the beginning of chapter 3, Form1.cs is the main form for the Dashboard\_APP project. This chapter puts forward how the application is implemented in Form1.cs by displaying the code. It will skip the introduction of Form1.Designer.cs that is already written in section 3.2.1.

### 4.2 Add More components to Form1 in Designer

#### 4.2.1.1 Add ContextMenuStrip Component to Form1

The dashboard application needs a cascading menu, which can be implemented by adding a *ContextMenuStrip* component from the ToolBox to the *Form1* form. *ContextMenuStrip* can be associated with any control, and a right mouse click automatically displays the shortcut menu. A *ContextMenuStrip* can be shown programmatically by using the *Show* method. *ContextMenuStrip* supports cancelable Opening and Closing events to handle the dynamic population and multiple-click scenarios. *ContextMenuStrip* supports images, menu-item check state, text, access keys, shortcuts, and cascading menus (Doc, ContextMenuStrip Class). The selections, e.g., Edit, Delete, and Connect, can be typed in by clicking the *Type Here* button as presented in *Figure 28 Design Form1*.



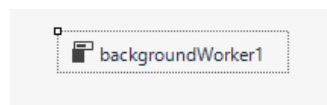
*Figure 28 Design Form1*



#### 4.2.1.2 Add BackgroundWorker Component to Form1

A *BackgroundWorker* returns an operation on a separate, dedicated thread. Time-consuming operations like downloads and database transactions can cause the user interface (UI) to seem like it has stopped responding while running. The *BackgroundWorker* class provides a convenient solution to implement a responsive UI (Doc, BackgroundWorker Class).

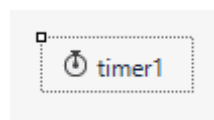
The *BackgroundWorker* component can be added by drag from the Toolbox to *Form1* form. A *BackgroundWorker* is visible in the Component Tray, and the properties are displayed in the Properties window. Please see Figure 29 BackgroundWorker



**Figure 29 BackgroundWorker**

#### 4.2.1.3 Add Timer Component to Form1

The Windows Forms Timer is a component that raises an event at regular intervals. This component is beneficial to a Windows Forms environment. Here a *timer1* is added to Form1 by dragging it from the Toolbox. It is also can be organized in the Component Tray.



**Figure 30 Timer**

### 4.3 Implementation of Form1 Class

There are two classes in Form1.cs. One is PCInfo class, which is already introduced in section 3.2.3, the other is Form1 Class, which begins with the code below.

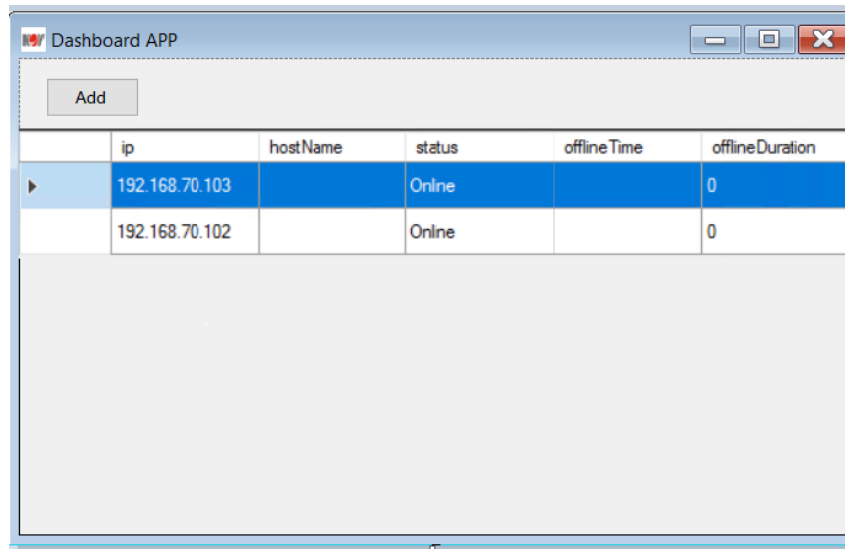
```
public partial class Form1 : Form
{
}
```

The Form1 class is compiled between the curly bracket.

#### 4.3.1.1 Create Constructor for Form1 Class

*Figure 31 Main form* shows that PC objects are located in rows on the main form of the DASHBOARD\_APP. The idea is to create two PC objects array; one is called *oldPCInfos*, the

other is called *PCInfos*. The PC objects are the elements in the array. The *oldPCInfos* array is updated by the *PCInfos* array whenever users type in a new PC object from the UI.



**Figure 31 Main form**

Two instances of the *BindingList<T>* class are declared at the beginning of the Form1 class, and the associate values are set in the constructor. The *BindingList<T>* class is a base class to create a two-way data-binding mechanism, which implements the *IBindingList* interface (Docs). Two-way data-binding refers to sharing data between a component class and its template. When there is a value changed in the input field from GUI, it will also reflate the value in a component class (Teacher). Besides, the selection and size mode needs to be set up in the constructor. The multi-selection is disabled by defining it as False.

```
// define a new PCInfo object list
private BindingList<PCInfo> PCInfos;
// define an old PC object list
private BindingList<PCInfo> oldPCInfos;

//1. constructor
public Form1()
{
    InitializeComponent();

    // setup the form
    dataGridView1.AutoSizeColumnsMode = DataGridViewAutoSizeColumnsMode.Fill;
    dataGridView1.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    dataGridView1.MultiSelect = false;

    // created an objet of PCInfo on the new PCInfo object list.
    PCInfos = new BindingList<PCInfo>();
    // created an objet of PCInfo on the old PCInfo object list.
    oldPCInfos = new BindingList<PCInfo>();

    // call the method to initialize the form
    initData();
}
```

```
}
```

#### 4.4 Initialize DataGridView1

*dataGridView1* is the area for the data grid view from *Figure 15 Form1 designer for Dashboard App*, where the PC objects will be displayed in rows. Therefore, the *BingdList* named *PCInfos* should be assigned to *dataGridView1*. The information of each PC object, Username, port, IP, etc., is displayed in each column. And all the initial information on *dataGridView1* is invisible by default. It is implemented by written *Visible=false* in the body area of the *initData()* method.

```
private void initData()
{
    // assign PCInfos to dataGridView1
    dataGridView1.DataSource = PCInfos;

    // set the initial information of PC object to be invisible
    dataGridView1.Columns["username"].Visible = false;
    dataGridView1.Columns["password"].Visible = false;
    dataGridView1.Columns["port"].Visible = false;
    dataGridView1.Columns["isOnline"].Visible = false;
    dataGridView1.Columns["isConn"].Visible = false;
}
```

#### 4.5 Create a Click Event Handler to Add Button

There is an Add button on the Form1 form, which is defined as Button1. When clicking the Button1, it triggers the event handler for Button. It displays the AddPC form from GUI, where users can input IP address, username, password, and port for adding a PC object.

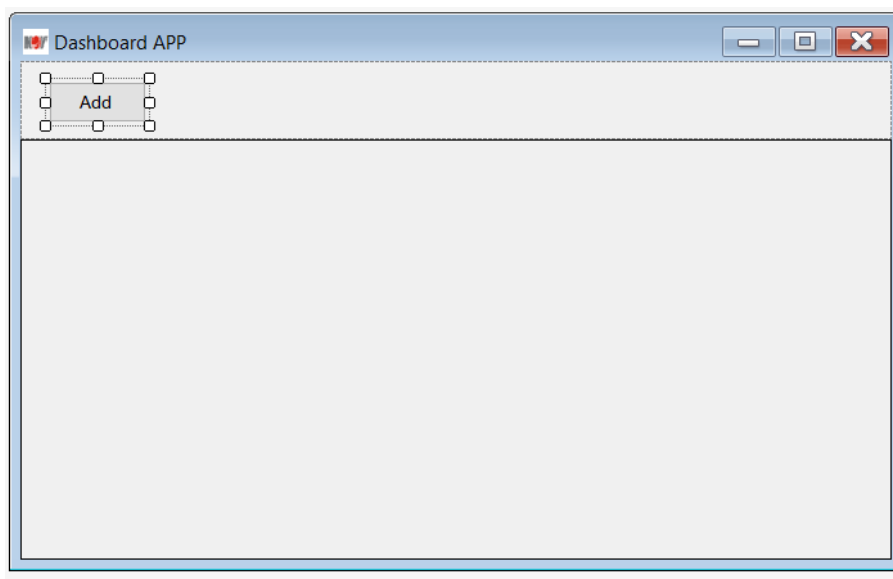


Figure 32 Form1 form

Please see the event handler named *button1\_click* below. An instance of *AddPC* class called *form* is created first, and the *AddPC form* should appear and cover the *Form1* form, and the *Form1* form is disabled until the user explicitly closes the *AddPC form*, which can be implemented by invoking the *form.ShowDialog()* method. In this manner, it shows the *AddPC form* as a modal dialog box. The *isOnline* and *isConn* status is implemented in the later section. Thus, they need to be set up as *false* by default so far.

```
private void button1_Click(object sender, EventArgs e)
{
    // create an instances of AddPC class.
    AddPC form = new AddPC();
    //shows the AddPC form as a modal dialog box.
    DialogResult res = form.ShowDialog();

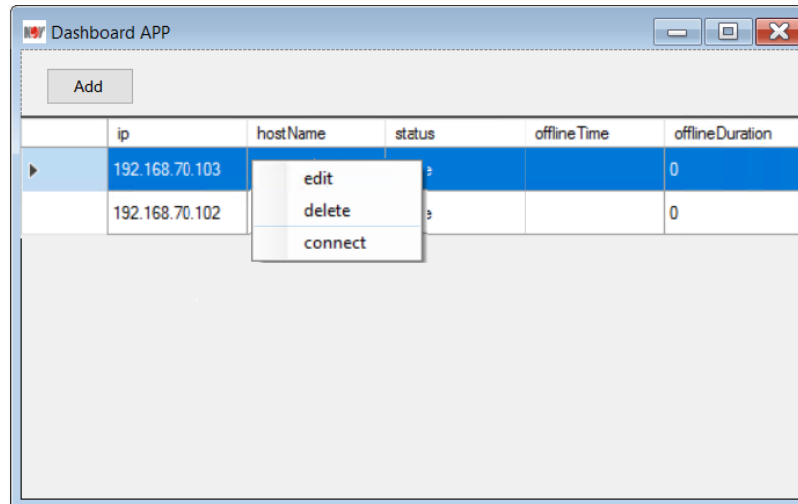
    // set Online status is false by default
    form.info.isOnline = false;
    // set the connection is false by default
    form.info.isConn = false;
    // verify if the input IP address already exists or not, when the DialogResult returns the value: OK.
    if (res == DialogResult.OK)
    {
        // if the IP address exists
        if (PCInfos.Any(index => index.ip == form.info.ip))
        {
            // show the message and return
            MessageBox.Show("IP already exists!");
            return;
        }
        // otherwise, assign the input information to PC object.
        PCInfo info = form.info;
        // and add the PC object to BindList PCInfos.
        PCInfos.Add(info);
    }
    // close AddPC form/window
    form.Dispose();
}
```

Each PC should have a unique IP address. Therefore, using an *If* statement to verify if the IP address already exists on the *BindList PCInfos*, when the *Save* button is clicked, returns the value *OK* to *DialogResult Enum*. Otherwise, assign the input information to the PC object, and add the PC object to *BindList PCInfos*. Since the *form.ShowDialog()* method is called, the *form.Dispose()* method will not be called automatically. It should be called manually for releasing the *Form1* form at the end of the *button1\_Click* event handler.

### 4.6 Add a Right-clicking Cascading Menu

So far, the dashboard application is functionalized for adding a remote computer by clicking Add button. When the information of a remote computer is finished inputting, it is visible in the area of *dataGridView1*. A cascading menu with *Edit*, *Delete*, and *Connect* options to the

remote computer is needed. Please look at *Figure 33 Cascading Menu for edit, delete and connect*.



**Figure 33 Cascading Menu for edit, delete and connect**

The event handler for the cascading menu is declared as *dataGridView1\_CellMouseUp* by calling the *ContextMenuStrip* from *Form1* Designer. Please see the code below. All the cells on the row of the remote computer are selected once right-clicking the mouse. Then call *Clearselection()* method to clear the current selection by unselecting all selected cells. The cascading menu should pop up at the position exactly where the right mouse clicked on the row of the remote computer.

```
private void dataGridView1_CellMouseUp(object sender, DataGridViewCellMouseEventArgs e)
{
    // if there is a right mouse clicking.
    if (e.Button == MouseButtons.Right)
    {
        // if the right mouse clicking is on the row of PC, not in the blank area.
        if (e.RowIndex >= 0 && e.ColumnIndex >= 0)
        {
            //call Clearselection() method to clear the current selection by unselecting all selected cells.
            dataGridView1.ClearSelection();
            // Get the selected row index
            dataGridView1.Rows[e.RowIndex].Selected = true;
            // Current grid
            dataGridView1.CurrentCell = dataGridView1.Rows[e.RowIndex].Cells[e.ColumnIndex];
            // show the cascading menu exactly in the position where the mouse clicking.
            contextMenuStrip1.Show(MousePosition.X, MousePosition.Y);
        }
    }
}
```

#### **4.7 Create a Click Event Handler to Edit Option**

The Event handler for *Edit* Option declares as *editToolStripMenuItem\_Click*. A variable of the selected row is declared as *var dataselect = this.dataGridView1.SelectedRows* if there is a need

to edit a remote computer on the dataGridView1. A new PCInfo instance named *info* and a new *AddPC form* instance named *frm* are created in a nested *If statement*. The idea is that to update the information/properties of the selected remote computer (*dataselect*) to the newly created PCInfo instance (*info*) first and then update the PCInfo instance (*info*) to the freshly made AddPC form instance (*frm*) in the outer *If statement*. Afterward, update the AddPC form instance (*frm*) to the selected row on the dataGridView1 (*dataselect*) in the inner *If statement*. The *AddPC form* will be closed, and *Form1* is released by calling the *Dispose()* method after the inner *If statement*. Otherwise, a message box "No data!" is displayed when the condition of the outer *If statement* is not matched, which means the selected row is empty. It can happen when the user right-clicking and select the edit option on an empty row.

```
private void editToolStripMenuItem_Click(object sender, EventArgs e)
{
    // declare a variable of the selected row,
    //and assign the information of PC object from the row on the
    // dataGridView1 to the variable.
    var dataselect = this.dataGridView1.SelectedRows;
    // if the PC object is selected.
    if (dataselect.Count > 0)
    {
        //create a new PC object of PCInfo class,
        PCInfo info = new PCInfo();
        //update the relevant information/property of PC to the new PC object.
        info.ip = dataselect[0].Cells["ip"].Value.ToString();
        info.username = dataselect[0].Cells["username"].Value.ToString();
        info.password = dataselect[0].Cells["password"].Value.ToString();
        info.port = dataselect[0].Cells["port"].Value.ToString();
        info.isOnline = Convert.ToBoolean(dataselect[0].Cells["isOnline"].Value.ToString());
        //create a new AddPC form to,
        AddPC frm = new AddPC();
        //update the value of the new PC object to AddPC form.
        frm.info = info;
        //show the AddPC form as a modal dialog.
        DialogResult res = frm.ShowDialog();
        //If the Save button is clicked, which indicates the value for the DialogResult is OK,
        if (res == DialogResult.OK)
        {
            //update the properties of PC object to dataGridView1.
            dataselect[0].Cells["ip"].Value = frm.info.ip;
            dataselect[0].Cells["username"].Value = frm.info.username;
            dataselect[0].Cells["password"].Value = frm.info.password;
            dataselect[0].Cells["port"].Value = frm.info.port;
            dataselect[0].Cells["isOnline"].Value = frm.info.isOnline;

            this.dataGridView1.Invalidate();
        }
        // close the AddPC form/window
        frm.Dispose();
    }
    // Otherwise, show the message if there is no PC object is selected.
    else
    {
        MessageBox.Show("No data!");
    }
}
```

## 4.8 Create a Click Event Handler to Delete Option

The event handler is declared with the name of *deleteToolStripMenuItem\_Click*. The purpose of this option is to delete the selected row of a computer. It can be implemented by iterating the row chosen collection and then delete the selected row if it is not the newly submitted row.

An instance of *DataGridViewRow* class named *dr* is created in the *foreach* statement, which is executed to iterate each element in the instances of the selected rows. Please see the code below.

```
private void deleteToolStripMenuItem_Click(object sender, EventArgs e)
{
    // declare a variable of the selected row,
    //and assign the information of PC object from the row on the dataGridView1 to the variable.
    var dataselect = this.dataGridView1.SelectedRows;
    // if the number of selected row is not 0
    if (dataselect.Count > 0)
    {
        // iterate the selected rows collection,
        foreach (DataGridViewRow dr in dataGridView1.SelectedRows)
        {
            //If it is not a submitted row, by default,
            //after adding a row of data successfully,
            //DataGridView will create a new row as the insertion location of the new data
            if (dr.IsNewRow == false)
            {
                // delete the row.
                dataGridView1.Rows.Remove(dr);
            }
        }
    }
    // otherwise, show "no data" message.
    else
    {
        MessageBox.Show("No data!");
    }
}
```

## 4.9 Create a Click Event Handler to Connect Option

The logic to implement the event handler for *Connect* option is similar to the logic for the *Edit* option. The event handler is declared with the name of *connectToolStripMenuItem\_Click*. A variable named *dataselect* is defined and assigned to the selected row on the *dataGridView1*, which is a *PCInfo* instance. Afterward, in the nested If statement, the properties of the selected *PCInfo* instance need to be assigned to *dataselect* in the outer *If* statement when there is a selected *PCInfo*. If not, show a message box with the message “No data!”. When the selected *PCInfo* instance is NOT online, trigger an inner *If* statement and display a message on the message box. Otherwise, declare an instance of *ConnForm*, which takes the *PCInfo* instance as

the parameter, and call the *show()* method to show the ConnForm form. Please see the code below.

```
private void connectToolStripMenuItem_Click(object sender, EventArgs e)
{
    // declare a variable of the selected row,
    //and assign the information of PC object from the row on the dataGridView1 to the variable.
    var dataselect = this.dataGridView1.SelectedRows;
    // if the number of selected row is not 0, which indicates there is a row/data selected.
    if (dataselect.Count > 0)
    {
        //create a new PC object of PCInfo class,
        PCInfo info = new PCInfo();
        //update the relevant information/property of PC to the new PC object by signing the data for the selected
row to info
        info.ip = dataselect[0].Cells["ip"].Value.ToString();
        info.username = dataselect[0].Cells["username"].Value.ToString();
        info.password = dataselect[0].Cells["password"].Value.ToString();
        info.port = dataselect[0].Cells["port"].Value.ToString();
        info.isOnline = Convert.ToBoolean(dataselect[0].Cells["isOnline"].Value.ToString());
        // if the selected row of computer is not Online.
        if (!info.isOnline)
        {
            // show the message box and return.
            MessageBox.Show("PC is offline!");
            return;
        }
        //Otherwise, create an instance of ConnForm and take info(which is the PCInfo instance) as the parameter.
        ConnForm form = new ConnForm(info);
        // call the show() method to show the ConnForm form.
        form.Show();
    }
    // otherwise, show a message box with the message "No data!"
    else
    {
        MessageBox.Show("No data!");
    }
}
```

#### **4.10 Create a Boolean Method to Return the Status of Remote Computer**

The Boolean method is named *StatusQuery*, which takes an IP address as a parameter. It returns True if the remote computer is reachable by *Ping.send()*. If the remote computer is not reachable by *Ping.send(ip)*, it returns false; meanwhile, an exception is caught and handled in a “try-catch-finally” block.

A Boolean type variable named *res*, an instance of Ping class, and an empty String type variable named *message* are declared before the “try-catch-finally” block. The *Ping.send(ip)* method is called in the *try* block, and it returns a value of *IPStatus*. An *If* statement assigns the value to the *message* variable while the *IPStatus* returns *Success*, and *True* will be released in the *finally* block. Otherwise, release a False result in the *finally* block, meanwhile raise an exception in the catch block.



The access for the method is defined as private. Since only the backgroundWorker will invoke it in the Form1 class, the *backgroundWorker* will be introduced in the next section.

```
private bool StatusQuery(string ip)
{
    // declare a bool type result named res.
    bool res;
    // the initial message is an empty string variable.
    string message = "";
    // create an instance of Ping
    Ping p = new Ping();
    // use "try-catch-finally" to raise an exception while the remote computer is not reachable by Ping
    try
    {
        // create an instance of PingReply class named r and
        // called Ping.send(ip) method to return a value for IPStatus.
        PingReply r = p.Send(ip);
        // if the return value from Ping.send() method is Success.
        if (r.Status == IPStatus.Success)
        {
            // assign the string "Success" to message.
            message = "Success";
        }
    }
    //deal with the exception in the catch block
    catch (Exception ex)
    {
        // raise the exception
        throw;
    }
    //release the result obtained in the try block
    finally
    {
        // if the message is string "Success"
        if (message == "Success")
        {
            // the result is true.
            res = true;
        }
        //otherwise
        else
        {
            // the result is false.
            res = false;
        }
    }
    // return the result
    return res;
}
```

### 4.11 Create a DoWork Event Handler to BackgroundWorker

Updating the real-time standby status for each remote computer is one of the functions of the dashboard application. Section 4.2.1.2 introduces how to add a BackgroundWorker to Form1. The *BackgroundWorker* listens for events that report the progress of the standby status of the remote computer and update the offline duration time when *StatusQuery()* is finished.

A *foreach* statement iterates the *PCInfo* instances in the *oldePCInfos* collection, and the *StatusQuery()* boolean method is called to get the status of the *PCInfo* instance at the beginning of the *backgroundWorker1\_DoWork* event handler. Two situations need to be considered in an *If statement*, when the *PCInfo* object is online and when the *PCInfo* object is offline.

When the *PCInfo* is online:

1. Create an instance of *IPHostEntry* from the specified IP address of the *PCInfo* instance.
2. Setup the hostname, status, offline time, and offline duration properties of the *PCInfo* instance.

When the *PCInfo* instance is offline:

1. Set the offline duration time as *DateTime.Now*.
2. If the *PCInfo* instance was online before, but the status is offline now.
  - a. Setup the status, offline time, and offline duration properties of the *PCInfo* instance.

The *dataGridView1.Invalidate()* method is called to repaint the *dataGridView*, and the *oldPCInfos* collection should be updated at the end of the event.

```
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
{
    // iterate the PCInfo instances in the oldePCInfos collection
    foreach (PCInfo item in oldPCInfos)
    {
        // call StatusQuery to get the status of each PCInfo instances
        item.isOnline = StatusQuery(item.ip);
        // if PCInfo instance is online
        if (item.isOnline)
        {
            // create the IPHostEntry instance form the ip address of PCInfo insatnce.
            IPHostEntry myScanHost = Dns.GetHostByAddress(item.ip);
            // assign the hostname
            item.hostName = myScanHost.HostName.ToString();
            // set ststus is online
            item.status = "Online";
            // set offtime is null
            item.offlineTime = null;
            // set the offline duration is 0.
            item.offlineDuration = 0;
        }
        // otherwise( when the PCInfo instance is offline)
        else
        {
            // check if the previous status of PCInfo instance is also offline
            if (item.status == "Offline")
            {
                //set the offline duration is
                //the current DateTime - offlineTime
                item.offlineDuration = Convert.ToInt32((DateTime.Now - item.offlineTime).Value.TotalSeconds);
            }
        }
    }
}
```

```

// otherwise( when the PCInfo instance was online)
else
{
    // change the ststus to offline.
    item.status = "Offline";
    // update the offline time to DateTime ow.
    item.offlineTime = DateTime.Now;
    // set the offline duration
    item.offlineDuration = 0;
}
}
// repaint the dataGridView
this.dataGridView1.Invalidate();
// update the oldPCInfos collection.
oldPCInfos = PCInfos;
}

```

## 4.12 Create a Tick Event Handler to Timer

The Timer.Tick event occurs when the *timer1* is enabled by setting *True* from the properties of the Timer1 component.

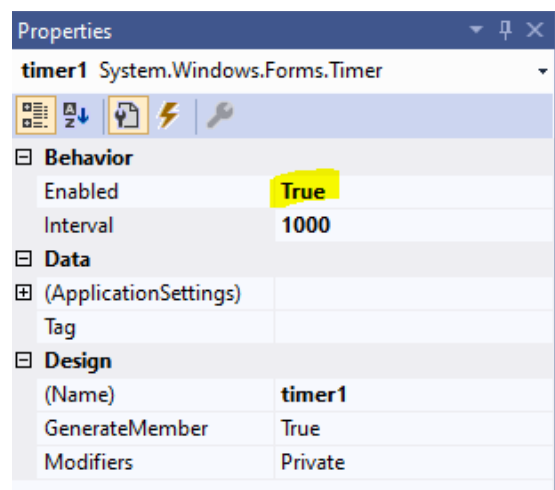


Figure 34 Properties of timer1

A *RunWorkerAsync()* method is called from the *timer1\_tick*. The *RunWorkerAsync()* method submits a request to start the operation running asynchronously. When the request is serviced, the *DoWork* event is raised, which starts executing the background operation—checking the *backgroundWorker1.IsBusy* property to see if the background task is running before calling the *RunWorkerAsync()* method. If so, it will return.

```

private void timer1_Tick(object sender, EventArgs e)
{
    // check IsBusy to see if the background task is running, and return
    if (backgroundWorker1.IsBusy)
    {
        return;
    }
}

```

```
// Start the operation in the background.  
backgroundWorker1.RunWorkerAsync();  
}
```

### 4.13 Create a CellFormatting Event Handler to DataGridView

A *CellFormatting* Event occurs when the contents of a cell need to be formatted for display in the *DataGridView*. The offline duration time is set to an equivalent 32-bit signed integer from the *backgroundWorker1\_DoWork* event handler.

By default, the *DataGridView* control displays the contents in String format. That is why a private *ConvertDayHourMinuteSencond(int duration)* method is created to convert the offline duration to String type.

The *ConvertDayHourMinuteSencond(int duration)* method takes an Integer type offline duration as a parameter, and it returns a String type. Please see the code below.

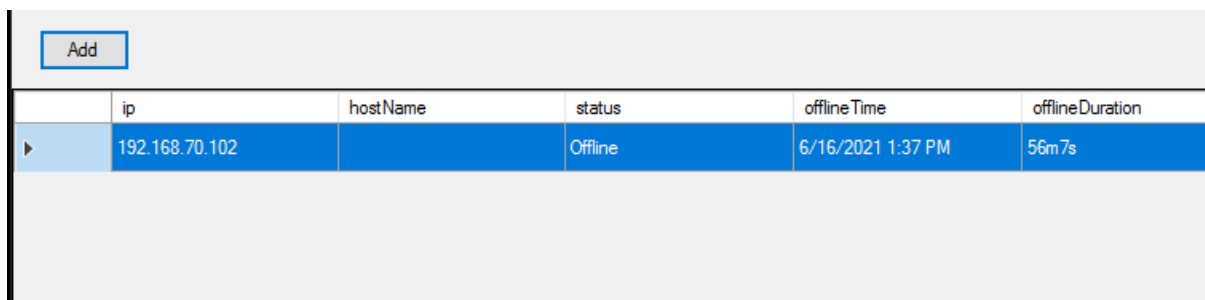
```
private string ConvertDayHourMinuteSencond(int duration)  
{  
    // get the time interval  
    TimeSpan ts = new TimeSpan(0, 0, duration);  
    // declare an empty string variable  
    string str = "";  
    // add day to string, if the offline duration time is more than 1 day.  
    if (ts.Days > 0)  
    {  
        str = ts.Days.ToString() + "d" + ts.Hours.ToString() + "h" + ts.Minutes.ToString() + "m" + ts.Seconds + "s";  
    }  
    //add hour to string if the offline duration time is more than 1 hour.  
    else if (ts.Hours > 0)  
    {  
        str = ts.Hours.ToString() + "h" + ts.Minutes.ToString() + "m" + ts.Seconds + "s";  
    }  
    // add minute to string if the offline duration time is more than 1 minute.  
    else if (ts.Minutes > 0)  
    {  
        str = ts.Minutes.ToString() + "m" + ts.Seconds + "s";  
    }  
    // add second to string if the offline duration time is more than 1 second.  
    else  
    {  
        str = ts.Seconds + "s";  
    }  
    // return string.  
    return str;  
}
```

Now is the time to declare the *CellFormatting* Event Handler. The event is implemented by three layers of nested *If* statements to meet the three conditions of aiming to the correct cell on the *dataGridView*. The outer *If* statement navigates the correct column, which is the column with the index 9. The column for the cell displays the offline duration time. The middle *If*

statement is triggered when the value of the cell is not Null. Moreover, the *ConvertDayHourMinuteSencond(int duration)* method is called to convert the Integer of offline duration time to String type when the value of the cell is not 0 converted by a *Convert.ToInt32* Method from *DateTime* format in the innermost *If* statement block.

```
private void dataGridView1_CellFormatting(object sender, DataGridViewCellFormattingEventArgs e)
{
    // if the index of column is 9
    if (e.ColumnIndex == 9)
    {
        // and the value is not null
        if (e.Value != null)
        {
            // if the value is not 0 covered from the integer
            if (Convert.ToInt32(e.Value) != 0)
            {
                // call the ConvertDayHourMinuteSencond(int duration) method
                //to convert the Integer to String
                e.Value = ConvertDayHourMinuteSencond(Convert.ToInt32(e.Value));
            }
        }
    }
}
```

The offline duration time is shown in *Figure 35 Offline duration* on the graphical interface of the dashboard application.



|   | ip             | hostName | status  | offlineTime       | offlineDuration |
|---|----------------|----------|---------|-------------------|-----------------|
| ▶ | 192.168.70.102 |          | Offline | 6/16/2021 1:37 PM | 56m 7s          |

*Figure 35 Offline duration*

#### **4.14 Create a Form.Load Event Handler to Form1**

The *Form.Load* Event performs tasks such as allocating resources used by the form, and the event occurs before a form is displayed for the first time. The event is declared by *private void Form1\_Load()*. When the dashboard application is launched, it reads data from a JavaScript Object Notation (JSON) file by calling *Readjson()* method. The JSON file is named “*config.json*”, which assembles the properties of the *PCInfo* object when the *FormClosing* event occurs.

```
private void Form1_Load(object sender, EventArgs e)
```

```
{
    // get the base directory, which is the config.json file
    string path = System.AppDomain.CurrentDomain.BaseDirectory + "config.json";
    // verify the config.json file exists or not.
    //if the config.json file exists, read the data.
    if (File.Exists(path))
        Readjson(path);
}
```

JSON provides an elementary database for each computer so that their information is stored for later use.

Please see the code for *Readjson()* method below. The method takes the path of the config.json file as a parameter. The logic for implementing the method is:

1. Create an instance of text reader named file and open an existing text file for reading.
2. Create an instance of *JSON text reader* which provides access to *JSON text data*.
3. Create a JSON array instance. *Read JSON* from config.json into JSON array
4. Iterate the JSON array
5. Get the current element from the JSON array.
6. Create a *PCInfo* instance. Update the current element from the JSON array to the *PCInfo* instance.
7. Add the *PCInfo* instance to the *PCInfos* collection.

```
private void Readjson(string path)
{
    // create a text reader insatnce of StreamReader Class
    using (System.IO.StreamReader file = System.IO.File.OpenText(path))
    {
        // create a instance of Json text reader
        using (JsonTextReader reader = new JsonTextReader(file))
        {
            // create a Json array, and assign the data read from Config.json to Json Array.
            JObject jarray = (JArray)JToken.ReadFrom(reader);
            // iterate the json Array
            for (int i=0; i<jarray.Count(); i++)
            {
                // get the current element in Json Array
                JObject temp = (JObject)jarray[i];
                // create a instance of PCInfo, named tempPCInfo
                //and update the properties of the current element to tempPCInfo.
                PCInfo tempPCInfo = new PCInfo
                {
                    ip = temp["ip"].ToString(),
                    hostName = temp["hostName"].ToString(),
                    username = temp["username"].ToString(),
                    password = temp["password"].ToString(),
                    port = temp["port"].ToString(),
                    isOnline = Convert.ToBoolean(temp["isOnline"].ToString()),
                    status = temp["status"].ToString(),
                    isConn = Convert.ToBoolean(temp["isConn"].ToString()),
                    offlineDuration = Convert.ToInt32(temp["offlineDuration"].ToString())
                };
            }
        }
    }
}
```

```
        // check the offline time
        //if the offline time is not empty.
        if (temp["offlineTime"].ToString() != "")
            // update the offline time of the current element to tempPCInfo
            tempPCInfo.offlineTime = Convert.ToDateTime(temp["offlineTime"].ToString());
        // add the tempPCInfo to PCInfos collection.
        this.PCInfos.Add(tempPCInfo);
    }
}
}
```

The advantage of using the C# using statement is that it defines a boundary for the object outside of which the object is automatically destroyed (Choksi).

The *Config.json* file is located in the C:\Program Files (x86)\Dashboard\_APP. The information of the remote PC is written to *Config.json* when the application is closed. The information will be read by the *Readjson()* method when the app is launched.

#### **4.15 Create a FormClosing Event Handler to Form1**

The *FormClosing* event occurs before the form is closed. When a form is closed, it will dispose and release all resources associated with the form. The *FormClosing* Event Handler is triggered whenever the dashboard application is closed, and a method is called to create and write the information for each *PCInfo* presented on the user interface to a JSON file.

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    // get the base directory, which is the config.json file
    string path = System.AppDomain.CurrentDomain.BaseDirectory + "config.json";
    // call writejson() method to create config.json in the directory.
    Writejson(path);
}
}
```

The *Writejson()* method also takes the path of the *config.json* file as a parameter. The logic for implementing the method is:

1. An encoded text file needs to be created/opened by calling *File.CreateText(String)* Method.
2. Create a *JsonTextWriter* instance to get access to write JSON.
3. Iterate the *PCInfos* collection, and update/write the information of each element from the *PCInfos* collection into the JSON file.

Please see the code for *Writejson()* method below.

```
private void Writejson(string path)
{
```

## Build a Dashboard Application for NOV's eVolve Automation System

```
// Creates or opens a file for writing encoded text. If the file already exists, its contents are overwritten.
using (System.IO.StreamWriter file = System.IO.File.CreateText(path))
{
    // create a JsonTextWriter instance
    using (JsonTextWriter writer = new JsonTextWriter(file))
    {
        // Writes the beginning of an array.
        writer.StartArray();
        // iterates the PCInfos collection,
        for (int i = 0; i < PCInfos.Count(); i++)
        {
            // get the current element in the PCInfos collection
            //write the properties of the current element into a JSON file.
            PCInfo temp = PCInfos[i];
            //Writes the beginning of a JSON object.
            writer.StartObject();

            writer.WritePropertyName("ip");
            writer.WriteValue(temp.ip);

            writer.WritePropertyName("hostName");
            writer.WriteValue(temp.hostName);

            writer.WritePropertyName("username");
            writer.WriteValue(temp.username);

            writer.WritePropertyName("password");
            writer.WriteValue(temp.password);

            writer.WritePropertyName("port");
            writer.WriteValue(temp.port);

            writer.WritePropertyName("isOnline");
            writer.WriteValue(temp.isOnline);

            writer.WritePropertyName("status");
            writer.WriteValue(temp.status);

            writer.WritePropertyName("isConn");
            writer.WriteValue(temp.isConn);

            writer.WritePropertyName("offlineTime");
            writer.WriteValue(temp.offlineTime);

            writer.WritePropertyName("offlineDuration");
            writer.WriteValue(temp.offlineDuration);
            //Writes the end of a JSON object.
            writer.EndObject();
        }
        //Writes the end of an array.
        writer.EndArray();
    }
}
```

Please see the complete code for the Form1 class in appendix C.



## **4.16 Summary of Chapter 4**

There are three more components added to Form1 designer, which are *contextMenuStrip1*, *timer1*, and *background worker1*. This chapter introduces the purpose and the functions of the added components at the beginning of the chapter.

The associated event handlers are created and implemented in the Form1 class, which handles the buttons, menu options, mouse-clicking events, timer, background worker, etc. These are introduced one after one by their running sequence. There are five functions and ten event handlers created in the Form1 class to functionalize the dashboard application in this chapter.

The technologies and instances for implementing the Form1 class,

1. *PCInfo* object contains the data and information of the remote computer. A *PCInfos* collection shuffles a list of *PCInfo* instances while some information needs to be updated.
2. *ConnForm* object is created for presenting the connection window.
3. *Ping* and *PingReply* query the status of the remote computer, which is invoked in the Boolean method *StatusQuery()*.
4. *IPHostEntry* instance is created from the IP address of the *PCInfo* instance, which obtains the hostname of the remote computer.
5. *TimeSpan* instance is declared to set up the time interval for the offline duration.
6. JSON file stores the information of *PCInfo* instances for providing the information to the dashboard application whenever it launches. It acts like a database of the dashboard application.
7. *StreamReader*, *JsonTextReader*, *StreamWriter*, and *JsonTextWriter* facilitate the reading and writing function of the JSON file when the *Form.Loading* event and *Form.Closing* event are triggered.

## **Chapter 5: Demonstration**

---

### **5.1 Introduction**

Chapter 5 introduces a procedure for creating an MSI installer in VS 2019 and presents a demonstration of the dashboard application in a video. The URL for that video is posted on youtube.com. The GitHub repository for the source code files of the dashboard application is also attached in one of the subsections. Two comments from NOV are also listed in the later subsection and the corresponding solutions from the author.

### **5.2 Create an MSI Installer for Dashboard Application.**

The following step creates the MSI:

1. The Microsoft Visual Studio Installer Projects should be installed as a prerequisite.
2. Right-click the Solution on VS 2019 and select Add new project.
3. Search the Setup project in the Add a new project window, rename the Setup project to *Demon*, and click create.
4. Add the Project Output to Applications Folder.
5. Add a shortcut to both Application Folder and User's Desktop.
6. To the end, it generates an installer file named *demon.msi*.

The reference learning tutorial about how to create an MSI installer in VS 2019, please see the video: <https://www.youtube.com/watch?v=fehVTLNQorQ>

### **5.3 The Video for Demonstration**

**Link:** [https://youtu.be/EP\\_kAZPlxKM](https://youtu.be/EP_kAZPlxKM)

The author makes the demonstration, which begins with installing the dashboard application on a WDP terminal Windows computer. The author expounds on how the dashboard application works step by step. Watching the video offers readers a deep understanding and gives an intuitive impression of the dashboard application.

#### **5.3.1 Modification According to The Comments**

The comments from NOV:

1. Modify the production name to NOV Dashboard and manufacturer to NOV.

2. The shortcut icon should be the NOV logo.

The solution from the author:

1. The production name and manufacturer can be modified in the properties window of the **Demon** project on VS 2019.

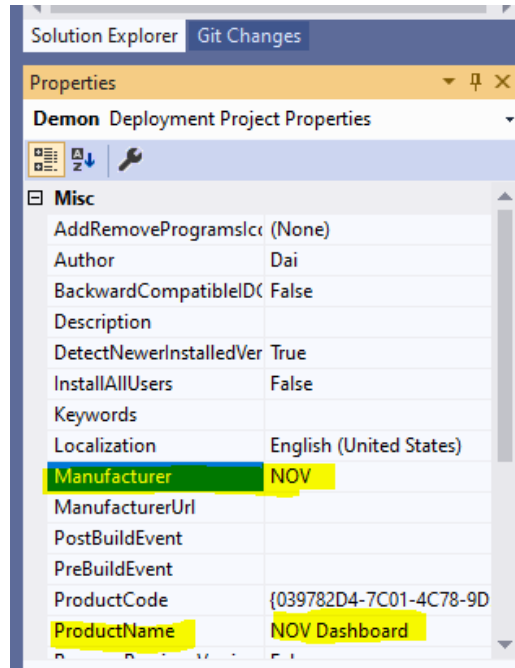


Figure 36 Modify the production name and manufacturer

2. Add the NOV.ico to the File System of the **Demon project**. And then change the icon to NOV.ico in the properties window of shortcut.

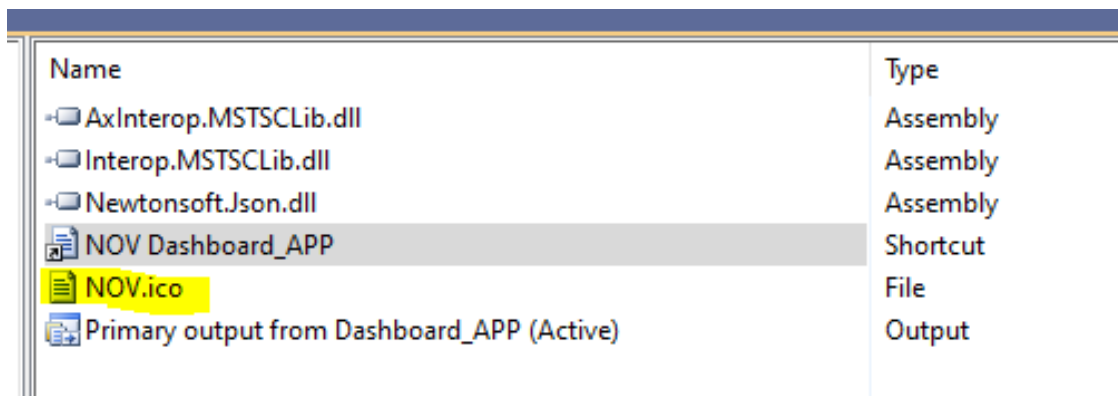


Figure 37 Add NOV.ico

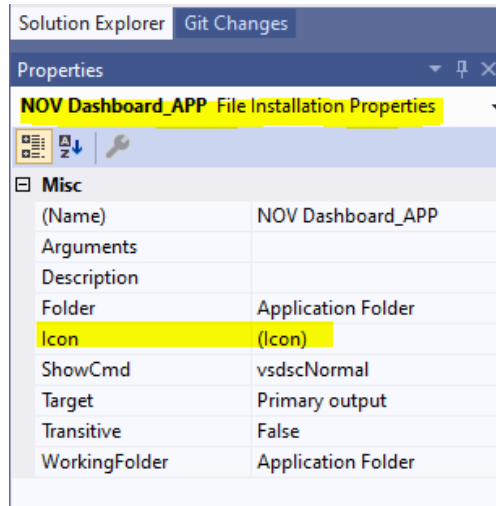


Figure 38 Modify icon of the shortcut

3. Rebuild the *Demon project*, and install the newly built MSI installer on the WDP terminal virtual machine.

Verification after installation:

1. The production name is NOV Dashboard, and the publisher is NOV

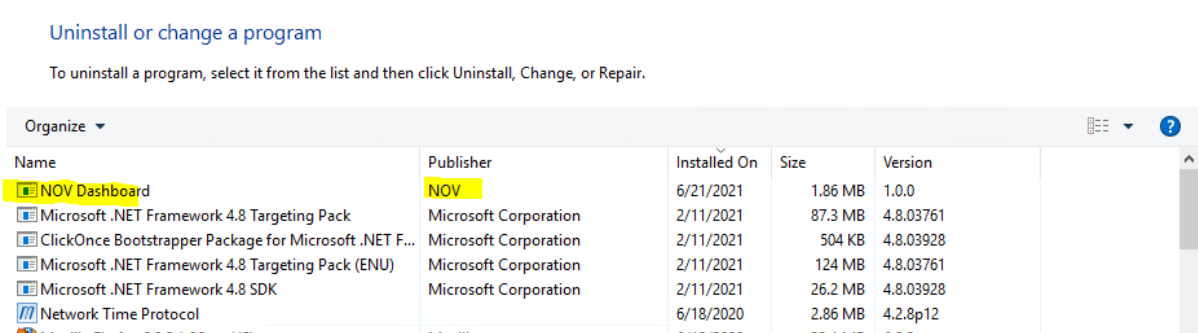


Figure 39 After installation 1

2. The shortcut on the desktop is with the NOV logo as an icon.

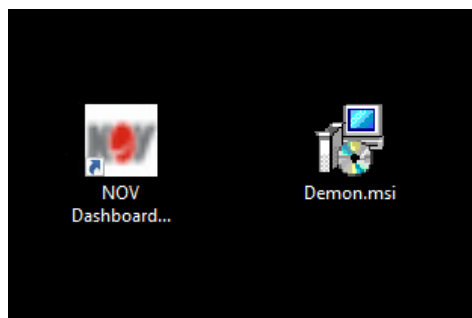


Figure 40 After installation 2

## **5.4 The GitHub Repository for the Project**

The author wrote all of the source code on her own, pushed it to the GitHub repository. The GitHub repository: [https://github.com/Daisynygaard/Dashboard\\_APP](https://github.com/Daisynygaard/Dashboard_APP). The source code files are also attached in the appendix.

## **5.5 Summary of Chapter 5**

Chapter 5 shows the steps of building an MSI installer on VS 2019 in the second subsection. Moreover, the author records a video of using the NOV Dashboard with a link attached. Furthermore, the source code files are uploaded to the GitHub repository. The relative modifications are done according to the comments from the NOV.

## **Chapter 6: Conclusion**

---

The entire bachelor program project includes building an NOV Dashboard application and writing a bachelor thesis. The author has finished the NOV Dashboard application by C# object-oriented programming and completed the thesis under her own steam. There are four classes created in the C# programming language with fourteen pages of source code, which include seven functions and thirteen event handlers, such as IsIP(), StatusQuery(), ConvertDayHourMinuteSencond(), Readjson(), Writejson(), Button\_click(), Form\_load(), dataGridView1\_CellFormatting(), Button\_click(), Background\_Doworker(), Timer\_Tick(), and so on. Furthermore, it also invoked massive C# classes and objects from the library.

### **6.1 Restate the Thesis**

Chapter 1 introduces a lot of hardware and software that interact in the NOV eVolve Automation System. The system integrates between the downhole tools and the surface automation systems to monitor and capture the downhole data while drilling. The downhole information is sent to NOV devices (e.g., Stream TV, WBC, RigSense, etc.) through the Intelliserve wired drill pipe. NOV employees can remotely control a Windows system device called the WDP terminal to access any of the mentioned NOV devices via Remote Desktop Connection.

High demand for a Windows-formed dashboard application with an overview of the devices as mentioned earlier is put on the table. The project aims to make a simple dashboard with the essential statuses of the devices as discussed above, making it easier to identify the issues delivered from the crew on an offshore rig to the NOV system engineer department.

Therefore, the author has built such a dashboard application, and it meets all the above demands. In addition, it realizes to establish remote control with any of the devices above. It shows the offline time and the duration of the devices' downtime and stores information in a JSON file for later investigation.

The dashboard application is implemented by doing the C# object-oriented programming on the Visual Studio 2019 software development platform. The author has studied the C# programming language for the project. The features and advantages of the C# are extracted in Chapter 2. Chapter 2 also presents the theoretical concept with the graphical explanation for

the dashboard application design and illustrates how to create a Windows-formed project on Visual Studio 2019.

The dashboard application consists of three source code files, *AddPC.cs*, *ConnForm.cs*, and *Form1.cs*. Chapter 3 shows the code of two partial classes in the *Dashboard\_APP* namespace, one is *AddPC* class located in the *AddPC.cs*, and the other is *ConnForm* class situated in the *ConnForm.cs*. A public *PCInfo* class from *Form1.cs* is also introduced in chapter 3. The *button\_Click event handler* invokes the *AddPC* class in the *Form1* class for creating an *AddPC* object to add a new PC to the dashboard application when a user clicks the Add button.

The *ConnForm* is short for Connection Form. When the Connection option is selected from a cascading menu, a *ConnForm* object is created to launch a window of the embedded Remote Desktop Connection App to remote connect with the related *PCInfo* object. Each *PCInfo* object stands for one remote computer.

The *AddPC* class, the *ConnForm* class, and the *PCInfo* class are invoked to create the corresponding object in the event handlers located in the *Form1* class. They are the foundation to implement the dashboard application with object-oriented programming.

Another partial class is named *Form1* class locates in the *Form1.cs*, which is the most critical partial class for the dashboard application project. All of the required functions of the dashboard application are implemented by compiling the event handlers in the *Form1* class. There are ten event handlers and five functions created in the *Form1* class. Chapter 4 introduces all of the event handlers with the associated code.

The process of generating an MSI installer for the dashboard application project on Visual Studio 2019 is introduced step by step at the beginning of chapter 5. A video for the demonstration of the completed dashboard application is recorded and uploaded to youtube.com. The three source code files are pushed to the GitHub repository. Please see the links below:

- The video for the demonstration: [https://youtu.be/EP\\_kAZPlxKM](https://youtu.be/EP_kAZPlxKM)
- GitHub repository for code: [https://github.com/Daisynygaard/Dashboard\\_APP](https://github.com/Daisynygaard/Dashboard_APP)

Chapter 5 also lists some feedback from the NOV. Based on the feedback, the author has given the solution and improved the dashboard application.

## **6.2 The Completed Dashboard Application**

After studying the C# 6.0 programming with Visual Studio 2019, the author has completed a Windows-formed dashboard application after accomplishing all the study as the thesis presents. The completed dashboard application is renamed to NOV Dashboard\_APP according to the comments from NOV.

After installing the NOV Dashboard\_APP on the WDP terminal Windows 10 computer, the program creates a shortcut on the desktop with the NOV logo as an icon. The first thing that catches the eye is a user interface with an *Add* button on the top-left when the NOV Dashboard\_APP is launched by double mouse-click the shortcut.

The NOV Dashboard\_APP can add the remote computer by typing in the information on the *AddPC* form. The action of clicking the Save button triggers a background worker event handler to call the *StatusQuery(string ip)* method to verify the status of the remote computer by invoking the *Ping* and *PingReply* class.

With the interaction between the Background Worker and the Timer component, the NOV Dashboard\_APP detects the real-time running status of the remote computer successfully. Furthermore, it displays the information on the NOV Dashboard\_APP user interface. In addition, the NOV Dashboard\_APP will also show offline time in the “*mm/dd/yyyy, hh: mm*” format if the remote computer powers off. The downtime interval increases every 5 seconds.

In addition, the NOV Dashboard\_APP has embedded a Remote Desktop Connection APP by creating an instance of the *axMsRdpClient8* class in the *ConnForm.cs* to accomplish the remote control over the network connection. In that case, the remote computers can be connected and controlled from NOV Dashboard\_APP when the *Connect* option is selected from the cascading menu.

Besides, the NOV Dashboard\_APP creates a JASON file database for storing the data inputting from the user interface and the information detected by the NOV Dashboard\_APP when the user interface closes. Then, the NOV Dashboard\_APP reloads the JSON file for the next launch.



### **6.3 The Advice for the Further Development**

The WellBore Connect (WBC) device is a critical component in the NOV eVolve Automation System, with many I/O modules running on it. It receives the oil well data while the offshore rig starts drilling. The dashboard application can now detect the running status of the WBC device and remotely control it with the built-in Remote Desktop Connection. Nevertheless, if the status of the signals on I/O modules for receiving oil well data can also be seen and displayed on the dashboard, it would give much more detailed information to identify the issues related to losing signals and shorten the time for resolving the problems.

## References

---

Arora, S. K. (2020). C# vs Java: Differences you should Know. Hentet fra

<https://hackr.io/blog/c-sharp-vs-java>

Choksi, D. (u.d.). C# using statement. Hentet fra <https://www.c-sharpcorner.com/article/the-using-statement-in-C-Sharp/>

Doc, M. (u.d.). BackgroundWorker Class. Hentet fra <https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.backgroundworker?view=net-5.0>

Doc, M. (u.d.). ContextMenuStrip Class. Hentet fra <https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.contextmenustrip?view=net-5.0>

Doc, M. (u.d.). Create a Windows Forms app in Visual Studio with C#. Hentet fra <https://docs.microsoft.com/en-us/visualstudio/ide/create-csharp-winform-visual-studio?view=vs-2019>

Doc, M. (u.d.). MaskedTextBox.UseSystemPasswordChar Property. Hentet fra <https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.maskedtextbox.usesystempasswordchar?view=net-5.0>

Doc, M. (u.d.). Ping Class. Hentet fra <https://docs.microsoft.com/en-us/dotnet/api/system.net.networkinformation.ping?view=net-5.0>

Doc, M. (u.d.). System Namespace. Hentet fra <https://docs.microsoft.com/en-us/dotnet/api/system?view=net-5.0#:~:text=Contains%20fundamental%20classes%20and%20base,%2C%20attributes%2C%20and%20processing%20exceptions.>

Docs, M. (u.d.). BindingList<T> Class. Hentet fra <https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.bindinglist-1?view=net-5.0>

GeeksforGeeks. (u.d.). C# Encapsulation. Hentet fra <https://www.geeksforgeeks.org/c-sharp-encapsulation/#:~:text=Encapsulation%20is%20defined%20as%20the,and%20the%20data%20it%20manipulates.&text=Encapsulation%20can%20be%20achieved%20by,get%20the%20values%20of%20variables.>

## *Build a Dashboard Application for NOVs eVolve Automation System*

International, E. (2017). C# Language Specification. Hentet fra <https://www.ecma-international.org/publications-and-standards/standards/ecma-334/>

Rashedul.Rubel. (u.d.). Form1.cs, Form1.designer.cs and Program.cs in c#. Hentet fra <https://stackoverflow.com/questions/21003049/difference-between-form1-cs-form1-designer-cs-and-program-cs-in-c-sharp>

Teacher, T. (u.d.). Two-way data binding. Hentet fra <https://www.tutorialsteacher.com/angular/two-way-data-binding>

w3schools. (u.d.). C# Constructors. Hentet fra [https://www.w3schools.com/cs/cs\\_constructors.asp](https://www.w3schools.com/cs/cs_constructors.asp)

w3schools.com. (u.d.). C# Introduction. Hentet fra [https://www.w3schools.com/cs/cs\\_intro.asp](https://www.w3schools.com/cs/cs_intro.asp)

wikipedia.org. (u.d.). Comparison of C Sharp and Java. Hentet fra [https://en.wikipedia.org/wiki/Comparison\\_of\\_C\\_Sharp\\_and\\_Java](https://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Java)

## Appendix A: AddPC.cs

---

```
using System;
using System.Linq;
using System.Windows.Forms;

namespace Dashboard_APP
{
    public partial class AddPC : Form
    {
        // constructor
        public AddPC()
        {
            InitializeComponent();
        }

        //define a private field named _info
        private PCInfo _info;

        // collecting information for PCInfo
        public PCInfo info
        {
            //read the information for PCInfo object
            get
            { //initialize the PCInfo object if there is no input.
              if (_info == null)
              {
                  _info = new PCInfo();
              }
              //read IP from the input in the text box 1
              _info.ip = textBox1.Text;
              //read username from the input in the text box 2
              _info.username = textBox2.Text;
              // read password from the input in the text box 3
              _info.password = textBox3.Text;
              // read port from the input in the text box 4
              _info.port = textBox4.Text;
              // return
              return _info;
            }
            // write the value to the variables for each PCInfo object.
            set
            {
                //set value to PCInfo object named _info
                _info = value;
                // write IP
                textBox1.Text = _info.ip;
                // write username
                textBox2.Text = _info.username;
                // write password
                textBox3.Text = _info.password;
                // write port
                textBox4.Text = _info.port;
            }
        }

        //Add button2 click Event Handler
        private void button2_Click(object sender, EventArgs e)
        {
            // examine the input of each text box is string, empty, or null.
            if (string.IsNullOrEmpty(textBox1.Text.Trim()) ||
```

```
string.IsNullOrEmpty(textBox2.Text.Trim()) ||
string.IsNullOrEmpty(textBox3.Text.Trim()) ||
string.IsNullOrEmpty(textBox4.Text.Trim()))
{
    //if any of the input is empty, show a message.
    MessageBox.Show("Cannot be empty!");
    return;
}
// call method IsIP to verify the IP
if (!IsIP(textBox1.Text.Trim()))
{
    // if it is not IP, show a message.
    MessageBox.Show("Illegal IP!");
    return;
}
// set the DialogResult is OK if any of the above "If" statements is triggered.
this.DialogResult = DialogResult.OK;
}

// create a boolean method named IsIP to verify the IP address is in the correct format.
public bool IsIP(string IP)
{
    // split the IP parameter, and count the number of the elements.
    var iCount = IP.Split('.').Count();
    // if the number of the elements is not 4 return false
    if (iCount != 4)
    {
        return false;
    }
    // create an IP address object named ip
    System.Net.IPAddress ip;
    // if the string IP can be converted to integer ip successfully,
    //return true. Otherwise return false.
    if (System.Net.IPAddress.TryParse(IP, out ip))
    {
        return true;
    }
    else
    {
        return false;
    }
}

// Add button1 click event handler
private void button1_Click(object sender, EventArgs e)
{
    // change the property value to False
    textBox3.UseSystemPasswordChar = !textBox3.UseSystemPasswordChar;
}
}
```

## Appendix B: ConnForm.cs

---

```
using MSTSCLib;
using System;
using System.Net.NetworkInformation;
using System.Windows.Forms;

namespace Dashboard_APP
{
    public partial class ConnForm : Form
    {
        // constructor
        public ConnForm(PCInfo info)
        {
            // initialize
            InitializeComponent();
            // create Ping object named pingsender
            Ping pingsender = new Ping();
            // create a PingReply object named reply and confirm if the remote computer is online.
            PingReply reply = pingsender.Send(info.ip);

            if (reply.Status == IPStatus.Success)
            {
                // setup IP
                axMsRdpClient81.Server = info.ip;
                // setup username
                axMsRdpClient81.UserName = info.username;
                //setup port number
                axMsRdpClient81.AdvancedSettings2.RDPPort = Convert.ToInt16(info.port);
                // setup size
                axMsRdpClient81.AdvancedSettings2.SmartSizing = true;
                // setup height
                axMsRdpClient81.DesktopHeight = this.Height;
                // setup width
                axMsRdpClient81.DesktopWidth = this.Width;
                // encryption is enabled
                axMsRdpClient81.AdvancedSettings9.NegotiateSecurityLayer = true;
                IMsTscNonScriptable securd = (IMsTscNonScriptable)axMsRdpClient81.GetOcx();
                // setup password
                securd.ClearTextPassword = info.password;
                axMsRdpClient81.AdvancedSettings5.ClearTextPassword = info.password;
                // setup color
                axMsRdpClient81.ColorDepth = 24;
                // establish the connection
                axMsRdpClient81.Connect();
            }
            //if there is no reply.
            else
            {
                // show the message box and close.
                MessageBox.Show("Unable to connect to the Server! ");
                this.Close();
            }
        }

        // resize event handler
        private void ConnForm_Resize(object sender, EventArgs e)
        {
            try
            {

```

## *Build a Dashboard Application for NOVs eVolve Automation System*

```
        // reconnect the remote computer according to the height and width.  
        axMsRdpClient81.Reconnect((uint)this.Width, (uint)this.Height);  
    }  
    catch  
    {  
        throw;  
    }  
}  
}
```

## Appendix C: Form1.cs

---

```
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using System;
using System.ComponentModel;
using System.IO;
using System.Linq;
using System.Net;
using System.Net.NetworkInformation;
using System.Windows.Forms;

namespace Dashboard_APP
{
    public partial class Form1 : Form
    {
        // define a new PCInfo object list
        private BindingList<PCInfo> PCInfos;
        // define an old PC object list
        private BindingList<PCInfo> oldPCInfos;

        //1. constructor
        public Form1()
        {
            InitializeComponent();

            // setup the form
            dataGridView1.AutoSizeColumnsMode = DataGridViewAutoSizeColumnsMode.Fill;
            dataGridView1.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
            dataGridView1.MultiSelect = false;

            // created an objet of PCInfo on the new PCInfo object list.
            PCInfos = new BindingList<PCInfo>();
            // created an objet of PCInfo on the old PCInfo object list.
            oldPCInfos = new BindingList<PCInfo>();

            // call the method to initialize the form
            initData();
        }

        // 2. Create a method named initData() to initialize the form
        private void initData()
        {
            // assign PCInfos to dataGridView1
            dataGridView1.DataSource = PCInfos;

            // set the initial information of PC object to be invisible
            dataGridView1.Columns["username"].Visible = false;
            dataGridView1.Columns["password"].Visible = false;
            dataGridView1.Columns["port"].Visible = false;
            dataGridView1.Columns["isOnline"].Visible = false;
            dataGridView1.Columns["isConn"].Visible = false;
        }

        // 3. add the event handler to button1/Add button
        private void button1_Click(object sender, EventArgs e)
        {
            // create an instances of AddPC class.
            AddPC form = new AddPC();
            // shows the AddPC form as a modal dialog box.
        }
    }
}
```



```
DialogResult res = form.ShowDialog();

// set Online status is false by default
form.info.isOnline = false;
// set the connection is false by default
form.info.isConn = false;
// verify if the input IP address already exists or not, when the DialogResult returns the value: OK.
if (res == DialogResult.OK)
{
    // if the IP address exists
    if (PCInfos.Any(index => index.ip == form.info.ip))
    {
        // show the message and return
        MessageBox.Show("IP already exists!");
        return;
    }
    // otherwise, assign the input information to PC object.
    PCInfo info = form.info;
    // and add the PC object to BindList PCInfos.
    PCInfos.Add(info);
}
// close AddPC form/window
form.Dispose();
}
```

#### // 4. add event handler for edit option

```
private void editToolStripMenuItem_Click(object sender, EventArgs e)
{
    // declare a variable of the selected row,
    //and assign the information of PC object from the row on the dataGridView1 to the variable.
    var dataselect = this.dataGridView1.SelectedRows;
    // if the PC object is selected.
    if (dataselect.Count > 0)
    {
        //create a new PC object of PCInfo class,
        PCInfo info = new PCInfo();
        //update the relevant information/property of PC to the new PC object.
        info.ip = dataselect[0].Cells["ip"].Value.ToString();
        info.username = dataselect[0].Cells["username"].Value.ToString();
        info.password = dataselect[0].Cells["password"].Value.ToString();
        info.port = dataselect[0].Cells["port"].Value.ToString();
        info.isOnline = Convert.ToBoolean(dataselect[0].Cells["isOnline"].Value.ToString());
        //create a new AddPC form to,
        AddPC frm = new AddPC();
        //update the value of the new PC object to AddPC form.
        frm.info = info;
        //show the AddPC form as a modal dialog.
        DialogResult res = frm.ShowDialog();
        //If the Save button is clicked, which indicates the value for the DialogResult is OK,
        if (res == DialogResult.OK)
        {
            //update the properties of PC object to dataGridView1.
            dataselect[0].Cells["ip"].Value = frm.info.ip;
            dataselect[0].Cells["username"].Value = frm.info.username;
            dataselect[0].Cells["password"].Value = frm.info.password;
            dataselect[0].Cells["port"].Value = frm.info.port;
            dataselect[0].Cells["isOnline"].Value = frm.info.isOnline;

            this.dataGridView1.Invalidate();
        }
        // close the AddPC form/window
        frm.Dispose();
    }
}
```

```
// Otherwise, show the message if there is no PC object is selected.
else
{
    MessageBox.Show("No data!");
}
}

// 5. add event handler for delete option
private void deleteToolStripMenuItem_Click(object sender, EventArgs e)
{
    // declare a variable of the selected row,
    //and assign the information of PC object from the row on the dataGridView1 to the variable.
    var dataselect = this.dataGridView1.SelectedRows;
    // if the number of selected row is not 0, which indicates there is a row/data selected.
    if (dataselect.Count > 0)
    {
        // iterate the selected rows collection,
        foreach (DataGridViewRow dr in dataGridView1.SelectedRows)
        {
            //If it is not a submitted row, by default,
            //after adding a row of data successfully,
            //DataGridView will create a new row as the insertion location of the new data
            if (dr.IsNewRow == false)
            {
                // delete the row.
                dataGridView1.Rows.Remove(dr);
            }
        }
    }
    // otherwise, show "no data" message.
    else
    {
        MessageBox.Show("No data!");
    }
}

//6. display the cascading menu, which is the contextMenuStrip from Form1 Designer.
private void dataGridView1_CellMouseUp(object sender, DataGridViewCellEventArgs e)
{
    // if there is a right mouse clicking.
    if (e.Button == MouseButtons.Right)
    {
        // if the right mouse clicking is on the row of PC, not in the blank area.
        if (e.RowIndex >= 0 && e.ColumnIndex >= 0)
        {
            //call Clearselection() method to clear the current selection by unselecting all selected cells.
            dataGridView1.ClearSelection();
            // Get the selected row index
            dataGridView1.Rows[e.RowIndex].Selected = true;
            // Current grid
            dataGridView1.CurrentCell = dataGridView1.Rows[e.RowIndex].Cells[e.ColumnIndex];
            // show the cascading menu exactly in the position where the mouse clicking.
            contextMenuStrip1.Show(MousePosition.X, MousePosition.Y);
        }
    }
}

// 7. add event handler for connect option
private void connectToolStripMenuItem_Click(object sender, EventArgs e)
{
    // declare a variable of the selected row,
    //and assign the information of PC object from the row on the dataGridView1 to the variable.
    var dataselect = this.dataGridView1.SelectedRows;
```

## Build a Dashboard Application for NOV's eVolve Automation System

```
// if the number of selected row is not 0, which indicates there is a row/data selected.
if (dataselect.Count > 0)
{
    //create a new PC object of PCInfo class,
    PCInfo info = new PCInfo();
    //update the relevant information/property of PC to the new PC object bu signing the data for the selected
row to info
    info.ip = dataselect[0].Cells["ip"].Value.ToString();
    info.username = dataselect[0].Cells["username"].Value.ToString();
    info.password = dataselect[0].Cells["password"].Value.ToString();
    info.port = dataselect[0].Cells["port"].Value.ToString();
    info.isOnline = Convert.ToBoolean(dataselect[0].Cells["isOnline"].Value.ToString());
    // if the selected row of computer is not Online.
    if (!info.isOnline)
    {
        // show the message box and return.
        MessageBox.Show("PC is offline!");
        return;
    }
    //Otherwise, create an instance of ConnForm and take info(which is the PCInfo instance) as the parameter.
    ConnForm form = new ConnForm(info);
    // call the show() method to show the ConnForm form.
    form.Show();
}
// otherwise, show a message box with the message "No data!"
else
{
    MessageBox.Show("No data!");
}
}
```

**//8.write a boolean method to verify the status of the remote computer is online,  
//and catch an exception if it is not online.**

```
private bool StatusQuery(string ip)
{
    // declare a bool type result named res.
    bool res;
    // the initial message is an empty string variable.
    string message = "";
    // create an instance of Ping
    Ping p = new Ping();
    // use "try-catch-finally" to raise an exception while the remote computer is not reachable by Ping
    try
    {
        // create an instance of PingReply class named r and
        //called Ping.send(ip) method to return a value for IPStatus.
        PingReply r = p.Send(ip);
        // if the return value from Ping.send() method is Success.
        if (r.Status == IPStatus.Success)
        {
            // assign the string "Success" to message.
            message = "Success";
        }
    }
    //deal with the exception in the catch block
    catch (Exception ex)
    {
        // raise the exception
        throw;
    }
    //release the result obtained in the try block
    finally
    {

```

```
// if the message is string "Success"
if (message == "Success")
{
    // the result is true.
    res = true;
}
//otherwise
else
{
    // the result is false.
    res = false;
}
}
// return the result
return res;
}
```

**// 9. event handler for timer**

```
private void timer1_Tick(object sender, EventArgs e)
{
    // check IsBusy to see if the background task is running, and return
    if (backgroundWorker1.IsBusy)
    {
        return;
    }
    // Start the operation in the background.
    backgroundWorker1.RunWorkerAsync();
}
```

**// 10. create an event handler to backgroundWorker**

```
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
{
    // iterate the PCInfo instances in the oldPCInfos collection
    foreach (PCInfo item in oldPCInfos)
    {
        // call StatusQuery to get the status of each PCInfo instances
        item.isOnline = StatusQuery(item.ip);
        // if PCInfo instance is online
        if (item.isOnline)
        {
            // create the IPHostEntry instance from the ip address of PCInfo instance.
            IPHostEntry myScanHost = Dns.GetHostByAddress(item.ip);
            // assign the hostname
            item.hostName = myScanHost.HostName.ToString();
            // set status is online
            item.status = "Online";
            // set offlineTime is null
            item.offlineTime = null;
            // set the offline duration is 0.
            item.offlineDuration = 0;
        }
        // otherwise( when the PCInfo instance is offline)
        else
        {
            // check if the previous status of PCInfo instance is also offline
            if (item.status == "Offline")
            {
                // set the offline duration is the current DateTime - offlineTime
                item.offlineDuration = Convert.ToInt32((DateTime.Now - item.offlineTime).Value.TotalSeconds);
            }
            // otherwise( when the PCInfo instance was online)
            else
            {
                {

```

```
        // change the status to offline.
        item.status = "Offline";
        // update the offline time to DateTime now.
        item.offlineTime = DateTime.Now;
        // set the offline duration
        item.offlineDuration = 0;
    }
}
// repaint the dataGridView
this.dataGridView1.Invalidate();
// update the oldPCInfos collection.
oldPCInfos = PCInfos;
}
```

**//11. Create a CellFormatting Event Handler to DataGridView**

```
private void dataGridView1_CellFormatting(object sender, DataGridViewCellFormattingEventArgs e)
{
    // if the index of column is 9
    if (e.ColumnIndex == 9)
    {
        // and the value is not null
        if (e.Value != null)
        {
            // if the value is not 0 converted from the integer
            if (Convert.ToInt32(e.Value) != 0)
            {
                // call the ConvertDayHourMinuteSecond(int duration) method to convert the Integer to String
                e.Value = ConvertDayHourMinuteSecond(Convert.ToInt32(e.Value));
            }
        }
    }
}
```

**//12. create a method to convert the offline duration time from integer to string.**

```
private string ConvertDayHourMinuteSecond(int duration)
{
    // get the time interval
    TimeSpan ts = new TimeSpan(0, 0, duration);
    // declare an empty string variable
    string str = "";
    // add day to string, if the offline duration time is more than 1 day.
    if (ts.Days > 0)
    {
        str = ts.Days.ToString() + "d" + ts.Hours.ToString() + "h" + ts.Minutes.ToString() + "m" + ts.Seconds + "s";
    }
    // add hour to string if the offline duration time is more than 1 hour.
    else if (ts.Hours > 0)
    {
        str = ts.Hours.ToString() + "h" + ts.Minutes.ToString() + "m" + ts.Seconds + "s";
    }
    // add minute to string if the offline duration time is more than 1 minute.
    else if (ts.Minutes > 0)
    {
        str = ts.Minutes.ToString() + "m" + ts.Seconds + "s";
    }
    // add second to string if the offline duration time is more than 1 second.
    else
    {
        str = ts.Seconds + "s";
    }
    // return string.
}
```

```
    return str;
}

// 13.create Form.Load event handler
private void Form1_Load(object sender, EventArgs e)
{
    // get the base directory, which is the config.json file
    string path = System.AppDomain.CurrentDomain.BaseDirectory + "config.json";
    // verify the config.json file exists or not.
    //if the config.json file exists, read the data.
    if (File.Exists(path))
        Readjson(path);
}

// 14. read data from config.json file
private void Readjson(string path)
{
    // create a text reader insatnce of StreamReader Class
    using (System.IO.StreamReader file = System.IO.File.OpenText(path))
    {
        // create a instance of Json text reader
        using (JsonTextReader reader = new JsonTextReader(file))
        {
            // create a Json array, and assign the data read from Config.json to Json Array.
            JArray jarray = (JArray)JToken.ReadFrom(reader);
            // iterate the json Array
            for (int i=0; i<jarray.Count(); i++)
            {
                // get the current element in Json Array
                JObject temp = (JObject)jarray[i];
                // create a instance of PCInfo, named tempPCInfo
                //and update the properties of the current element to tempPCInfo.
                PCInfo tempPCInfo = new PCInfo
                {
                    ip = temp["ip"].ToString(),
                    hostName = temp["hostName"].ToString(),
                    username = temp["username"].ToString(),
                    password = temp["password"].ToString(),
                    port = temp["port"].ToString(),
                    isOnline = Convert.ToBoolean(temp["isOnline"].ToString()),
                    status = temp["status"].ToString(),
                    isConn = Convert.ToBoolean(temp["isConn"].ToString()),
                    offlineDuration = Convert.ToInt32(temp["offlineDuration"].ToString())
                };
                // check the offline time
                //if the offline time is not empty.
                if (temp["offlineTime"].ToString() != "")
                    // update the offline time of the current element to tempPCInfo
                    tempPCInfo.offlineTime = Convert.ToDateTime(temp["offlineTime"].ToString());
                // add the tempPCInfo to PCInfos collection.
                this.PCInfos.Add(tempPCInfo);
            }
        }
    }
}

//15. Write JSON file.
private void Writejson(string path)
{
    // Creates or opens a file for writing encoded text. If the file already exists, its contents are overwritten.
    using (System.IO.StreamWriter file = System.IO.File.CreateText(path))
    {
        // create a JsonTextWriter instance
    }
}
```

```
using (JsonTextWriter writer = new JsonTextWriter(file))
{
    // Writes the beginning of an array.
    writer.WriteStartArray();
    // iterates the PCInfos collection,
    for (int i = 0; i < PCInfos.Count(); i++)
    {
        // get the current element in the PCInfos collection
        //write the properties of the current element into a JSON file.
        PCInfo temp = PCInfos[i];
        //Writes the beginning of a JSON object.
        writer.WriteStartObject();

        writer.WritePropertyName("ip");
        writer.WriteValue(temp.ip);

        writer.WritePropertyName("hostName");
        writer.WriteValue(temp.hostName);

        writer.WritePropertyName("username");
        writer.WriteValue(temp.username);

        writer.WritePropertyName("password");
        writer.WriteValue(temp.password);

        writer.WritePropertyName("port");
        writer.WriteValue(temp.port);

        writer.WritePropertyName("isOnline");
        writer.WriteValue(temp.isOnline);

        writer.WritePropertyName("status");
        writer.WriteValue(temp.status);

        writer.WritePropertyName("isConn");
        writer.WriteValue(temp.isConn);

        writer.WritePropertyName("offlineTime");
        writer.WriteValue(temp.offlineTime);

        writer.WritePropertyName("offlineDuration");
        writer.WriteValue(temp.offlineDuration);
        //Writes the end of a JSON object.
        writer.WriteEndObject();
    }
    //Writes the end of an array.
    writer.WriteEndArray();
}
}
}

// 16.create FormClosing event handler
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    // get the base directory, which is the config.json file
    string path = System.AppDomain.CurrentDomain.BaseDirectory + "config.json";
    // call writejson() method to create config.json in the directory.
    Writejson(path);
}
}

// PC information class
public class PCInfo
```

```
{
  //Constructor
  public PCInfo() { }
  //create fields
  private string _ip;
  private string _hostName;
  private string _username;
  private string _password;
  private string _port;
  private bool _isOnline;
  private string _status;
  private bool _isConn;
  private DateTime? _offlineTime;
  private int _offlineDuration;
  /// <summary>
  ///
  /// </summary>
  /// get and set method. The property
  public string ip
  {
    set { _ip = value; }
    get { return _ip; }
  }

  public string hostName
  {
    set { _hostName = value; }
    get { return _hostName; }
  }

  public string username
  {
    set { _username = value; }
    get { return _username; }
  }

  public string password
  {
    set { _password = value; }
    get { return _password; }
  }

  public string port
  {
    set { _port = value; }
    get { return _port; }
  }

  public bool isOnline
  {
    set { _isOnline = value; }
    get { return _isOnline; }
  }

  public string status
  {
    set { _status = value; }
    get { return _status; }
  }

  public bool isConn
  {
    set { _isConn = value; }
  }
}
```



## *Build a Dashboard Application for NOVs eVolve Automation System*

```
        get { return _isConn; }
    }

    public DateTime? offlineTime
    {
        set { _offlineTime = value; }
        get { return _offlineTime; }
    }

    public int offlineDuration
    {
        set { _offlineDuration = value; }
        get { return _offlineDuration; }
    }
}
}
```